# Design and Analysis of Algorithm

## Advanced Data Structure (Fibonacci Heap)

LECTURE 45 - 48

## Overview

- This section present data structures known as mergeable heaps, which support the following seven operations.
    - MAKE-FIBO-HEAP()
    - FIB-HEAP-INSERT()
    - FIB-HEAP-MINIMUM()
    - FIB-HEAP-EXTRACT-MIN()
    - FIB-HEAP-UNION()
    - FIB-HEAP DECREASE-KEY()
    - FIB-HEAP-DELETE()

## Fibonacci Heap

- Fibonacci heap is designed and developed by Fredman and Tarjan in the year 1986
- Ingenious data structure and analysis.
- Like a binomial heap, a Fibonacci heap is a collection of trees with looser structure.
- Fibonacci heaps are called lazy data structures because they delay work as long as possible using the field mark (i.e. black node).

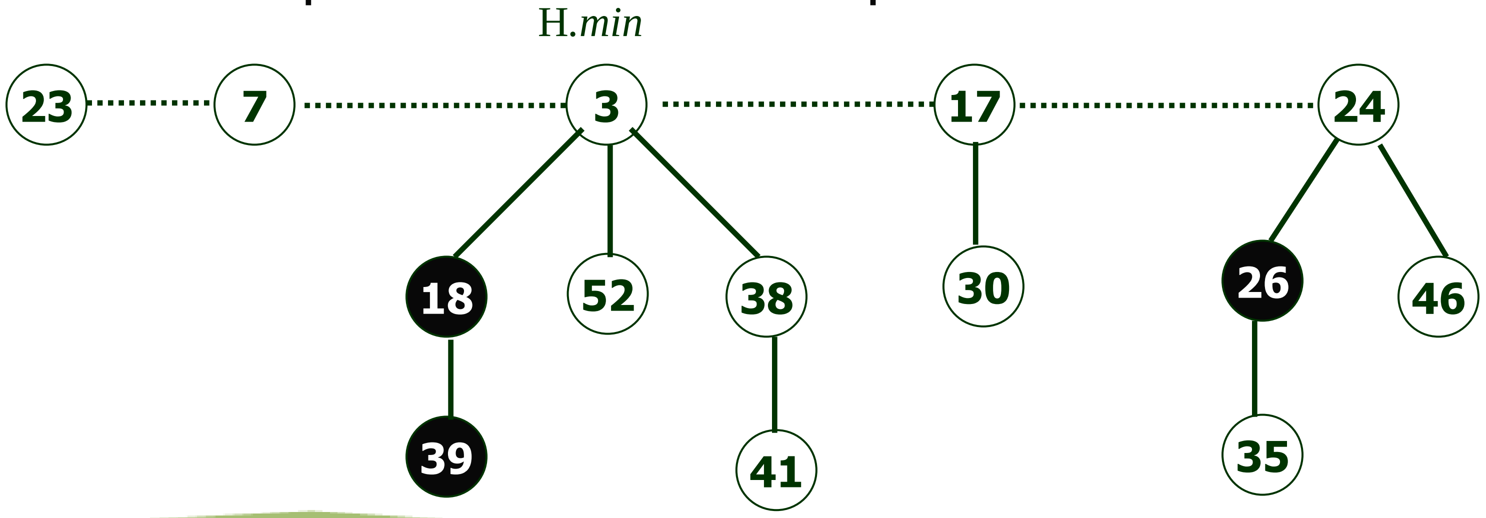
# Fibonacci Heap

Properties:

- Unlike Binomial Heap, Fibonacci Heap can have many tree of same degree and tree does not have exactly $2^k$ nodes.
- Tree in Fibonacci heap are rooted but unordered.
- Roots and Sibling are circular doubly link list.
- Each node have its degree. The number of children of x has degree[x].
- Each node mark [x] either true (i.e. black node) or false. Newly created node is always unmarked.

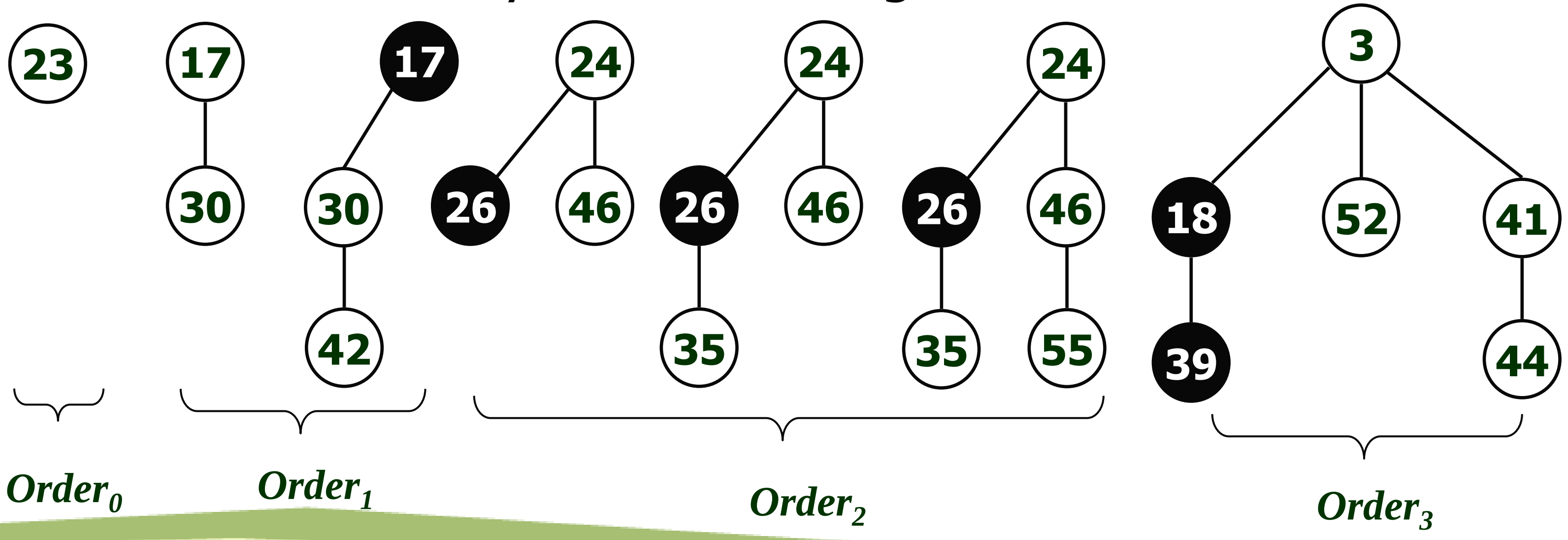
## Fibonacci Heap(Structure)

- Structure of Fibonacci heaps
  - Like a binomial heap, a Fibonacci heap is a collection of min-heap-ordered trees. The trees in a Fibonacci heap are not constrained to be binomial trees. The picture shows below is an example of a Fibonacci heap.



# Fibonacci Heap(Structure)

- Nodes within a Fibonacci heap can be removed from their tree without restructuring them.
- So the order does not necessarily indicate the maximum height of the tree or number of nodes it contain. Some examples of order 0, 1, 2 are given below for easy understanding.



## Fibonacci Heap (Structure)

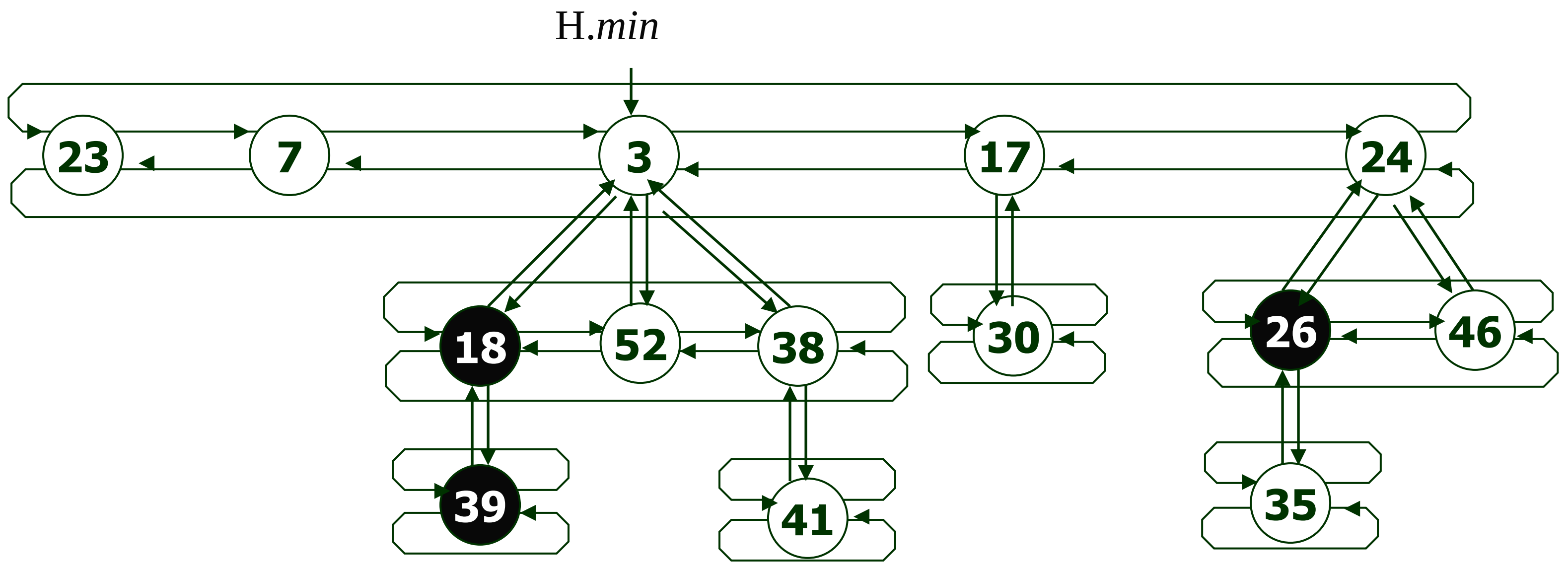
- Each node x contain the following fields.
  - x.p → points to its parent
  - x.child → points to any one of its children and children of x are linked together in a circular doubly linked list.
  - x.left, x.right → points to its left and right siblings.
  - x.degree → number of children in the child list of x

## Fibonacci Heap(Structure)

- x.mark → indicate whether node x has lost a child since the last time x was mode the child of another node
- H.min → points to the root of the tree containing a minimum key
- H.n → number of nodes in H

# Fibonacci Heap(Structure)

- An example of Fibonacci Heap with the help of doubly circular link list is given below.



## Fibonacci Heap

- The Fibonacci heap data structure serves a dual purpose.
  - First, it supports a set of operations that constitutes what is known as a “mergeable heap.”
  - Second, several Fibonacci-heap operations run in constant amortized time, which makes this data structure well suited for applications that invoke these operations frequently.

## Fibonacci Heap

- A mergeable heap is any data structure that supports the following seven operations, in which each element has a key:
  1. **MAKE-HEAP()** creates and returns a new heap containing no elements.
  2. **MINIMUM(H)** returns a pointer to the node in heap H whose key is minimum.
  3. **UNION(H1, H2)** creates and returns a new heap that contains all the nodes of heaps H1 and H2. Heaps H1 and H2 are "destroyed" by this operation.
  4. **EXTRACT-MIN(H)** deletes the node from heap H whose key is minimum, returning a pointer to the node.

# Fibonacci Heap

5. **INSERT(H, x)** inserts node x, whose key field has already been filled in, into heap H.
6. **DECREASE-KEY(H, x, k)** assigns to node x within heap H the new key value k, which is assumed to be no greater than its current key value.[1]
7. **DELETE(H, x)** deletes node x from heap H

# Fibonacci Heap

| Heaps Analysis | | | |
|---|---|---|---|
| Operation | Binary | Binomial | Fibonacci † |
| make-heap | 1 | 1 | 1 |
| insert | log N | log N | 1 |
| find-min | 1 | log N | 1 |
| delete-min | log N | log N | log N |
| union | N | log N | 1 |
| decrease-key | log N | log N | 1 |
| delete | log N | log N | log N |
| is-empty | 1 | 1 | 1 |

Note: † amortized

# Fibonacci Heap

- Amortized Analysis
  - Analyze a sequence of operations on a data structure.
  - It shows that although some individual operations may be expensive, on average the cost per operation is small.
- Average in this context does not mean that we're averaging over a distribution of inputs.
  - No probability is involved.
  - We're talking about average cost in the worst case.

## Fibonacci Heap

- The Three most common techniques used are
  - Aggregate Analysis
  - Accounting Method
  - Potential Method

## Fibonacci Heap

- The Three most common techniques used are
  - Aggregate Analysis
  - Accounting Method
  - Potential Method

- The potential method is used to analyze the performance of Fibonacci heap operations.

# Fibonacci Heap

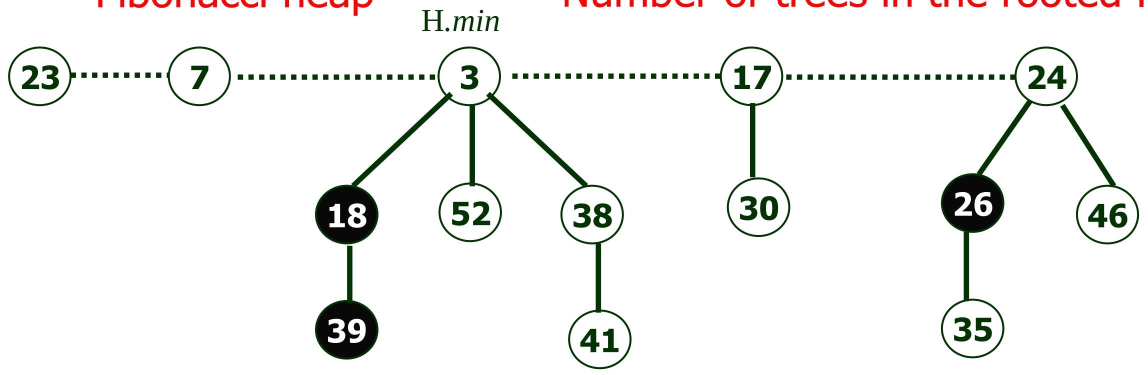
- For a given Fibonacci heap $\mathrm{H}$
  - $\mathrm{t(H)}$ the number of trees in the root list of $\mathrm{H}$.
  - $\mathrm{m(H)}$ the number of marked nodes in $\mathrm{H}$.
- The potential of Fibonacci heap H is then defined by

marked nodes in H

$$\Phi(\mathrm{H}) = \mathrm{t(H)} + 2.\,\mathrm{m(H)}$$

Fibonacci heap

Number of trees in the rooted list of H



In the above example $\mathrm{t(H)} = 5$ and $\mathrm{m(H)} = 3$, Hence

$$\Phi(\mathrm{H}) = 5 + 2.3 = 11$$

## Fibonacci Heap

- Fibonacci heap application begins with no heaps. Hence
  - The initial potential is 0(zero) and the potential is nonnegative at all subsequent times.
  - An upper bound on the total amortized cost is thus an upper bound on the total actual cost for the sequence of operations.
- Maximum degree
  - The amortized analyses was performed with a known upper bound D(n) on the maximum degree of any node in an n-node Fibonacci heap. (i.e. $D(n) \leq \lfloor \lg n \rfloor$).

# Fibonacci Heap (Operations)

**Creating a new Fibonacci Heap:**

- To make an empty Fibonacci heap, the MAKE-FIB-HEAP procedure allocates and returns the Fibonacci heap object H ,
- where $\mathrm{n[H]} = 0$ and $\mathrm{min[H]} = \mathrm{NIL}$; there are no trees in H .
- Because $\mathrm{t(H)} = 0$ and $\mathrm{m(H)} = 0$ , the potential of the empty Fibonacci heap is
  - $\Phi(H) = 0.$
- The amortized cost of MAKE-FIB-HEAP is thus equal to its $\mathrm{O}(1)$ actual cost.

## Fibonacci Heap (Operations)

**Inserting a node:**

- To insert a node in Fibonacci heap the following steps will be used.

  Step1 :Create a new singleton tree.

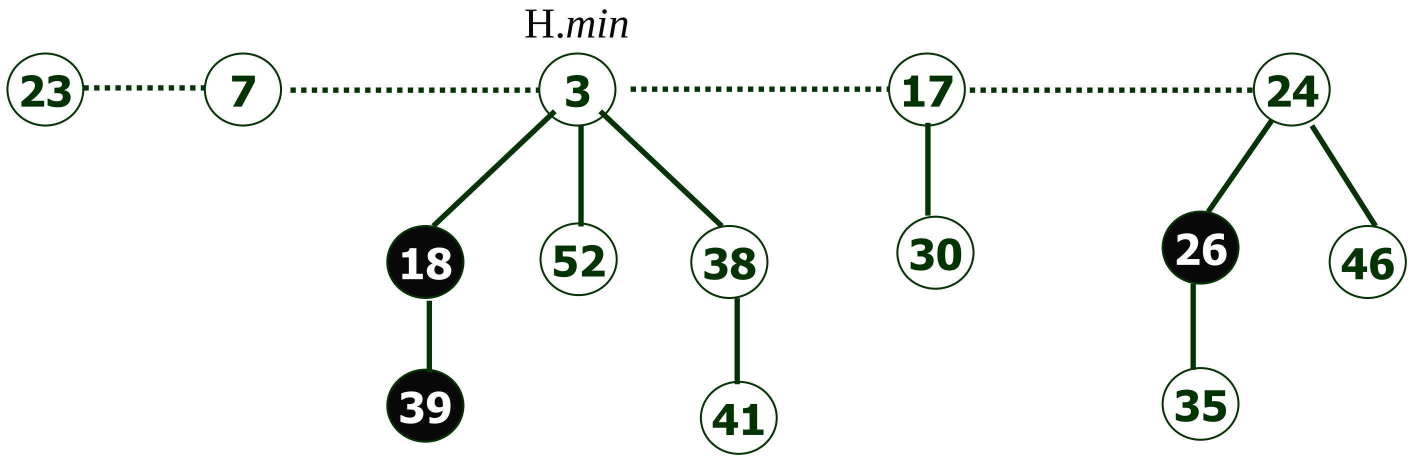
  Step2 : Add to left of min pointer.

  Step3 : Update min pointer.

## Fibonacci Heap (Operations)

**Inserting a node:**

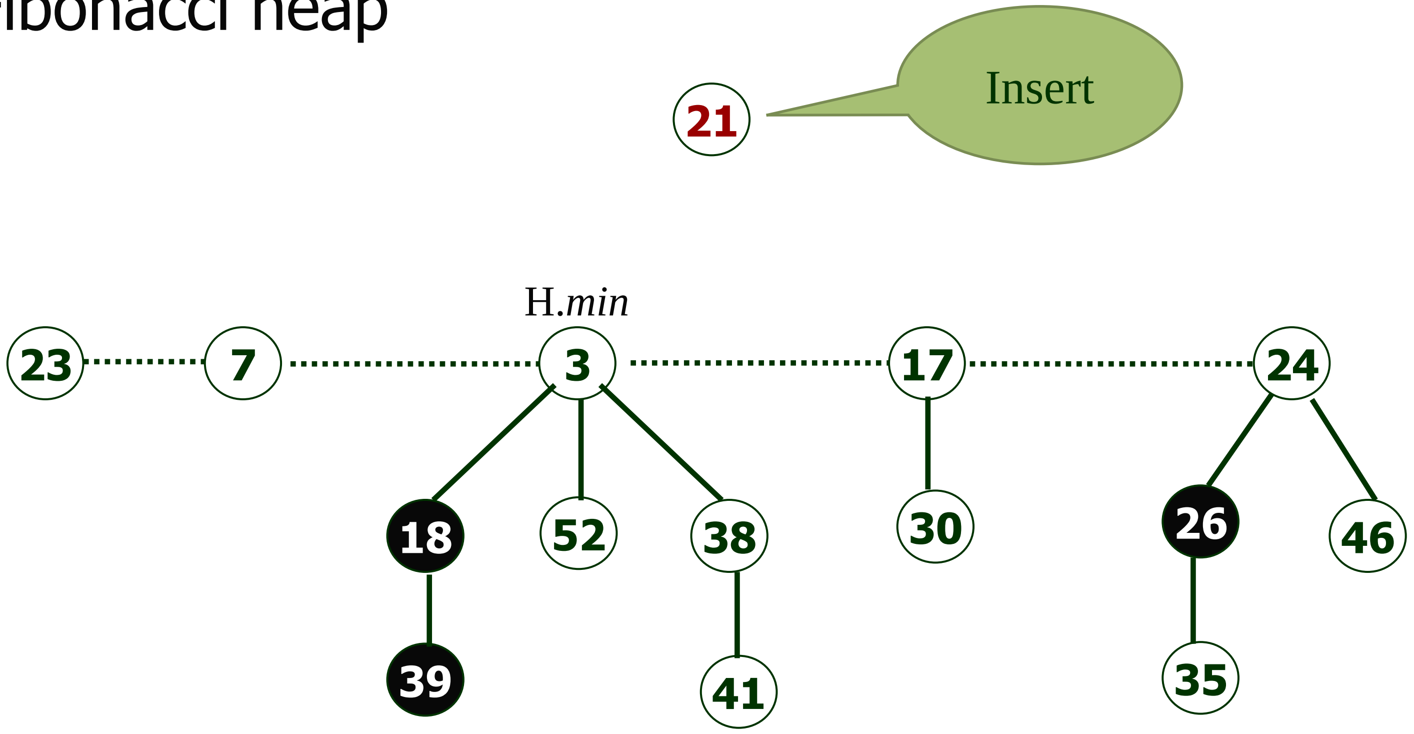
- Example: Insert the node 21 in following Fibonacci heap



## Fibonacci Heap (Operations)

**Inserting a node:**

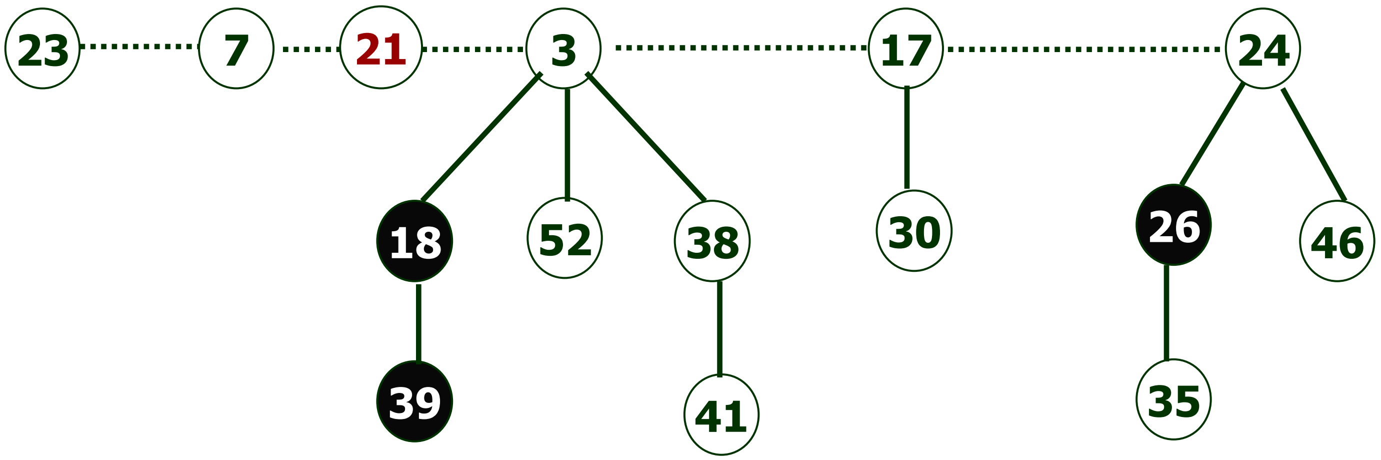
- Example: Insert the node 21 in following Fibonacci heap



# Fibonacci Heap (Operations)

**Inserting a node:**

- Example: Insert the node 21 in following Fibonacci heap



## Fibonacci Heap (Operations)

### Inserting a node:

- The following procedure inserts node x into Fibonacci heap H , assuming that the node has already been allocated and that key[x] has already been filled in.

```
FIB-HEAP-INSERT(H, x)
 1  degree[x] ← 0
 2  p[x] ← NIL
 3  child[x] ← NIL
 4  left[x] ← x
 5  right[x] ← x
 6  mark[x] ← FALSE
 7  concatenate the root list containing x with root list H
 8  if min[H] = NIL or key[x] < key[min[H]]
 9     then min[H] ← x
10     n[H] ← n[H] + 1
```

# Fibonacci Heap (Operations)

## Inserting a node (Analysis)

To determine the amortized cost of FIB-HEAP-INSERT,

let

$H$ be the input Fibonacci heap and

$H'$ be the resulting Fibonacci heap.

Then,

$t(H') = t(H) + 1$ and

$m(H') = m(H)$,

and the difference in potential cost is=

$$((t(H) + 1) + 2\,m(H)) - (t(H) + 2\,m(H)) = 1.$$

Since the actual cost is $O(1)$,

the amortized cost = Actual cost+ Difference in potential cost $(\Delta(\Phi))$

$$= O(1) + 1 = O(1).$$

## Fibonacci Heap (Operations)

### Finding the minimum node

- The minimum node of a Fibonacci heap H is given by the pointer min[H],
- So the actual time for finding the minimum node is O(1). Because the potential of H does not change, and the amortized cost of this operation is equal to its O(1) actual cost.

# Fibonacci Heap (Operations)

## Uniting two Fibonacci heaps (Union)

This procedure unites Fibonacci heaps H1 and H2, destroying H1 and H2 in the process. It simply concatenates the root lists of H1 and H2 and then determines the new minimum node.

**Basic Idea:**

Step 1: Concatenate two Fibonacci heaps.

Step 2: Root lists are circular, doubly linked lists.

## Fibonacci Heap (Operations)

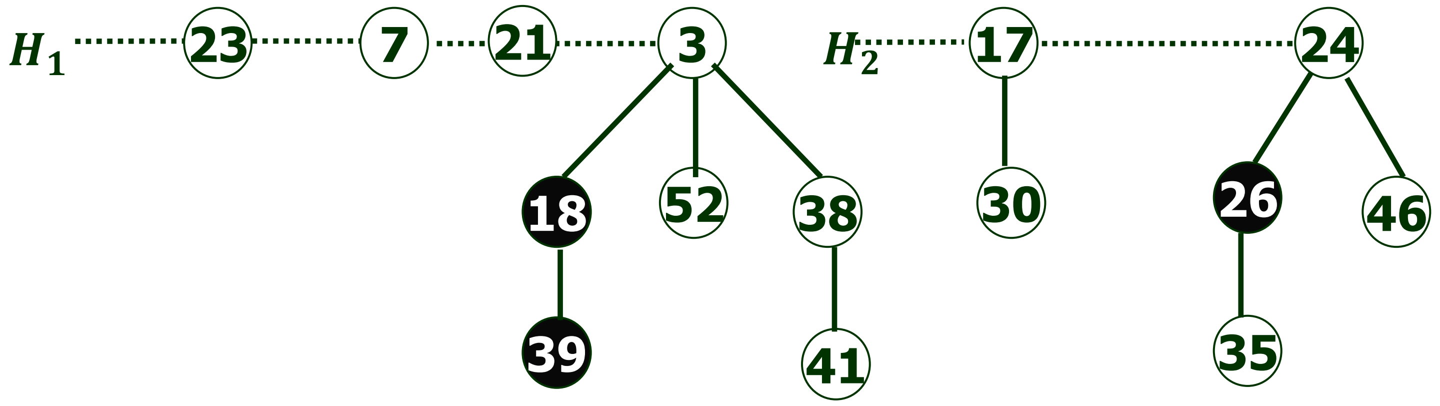
### Uniting two Fibonacci heaps

```
FIB-HEAP-UNION(H1, H2)
1   H ← MAKE-FIB-HEAP()
2   min[H] ← min[H1]
3   Concatenate the root list of H2 with the root list of H
4   if (min[H1] = NIL) or (min[H2] ≠ NIL) and min[H2] < min[H1])
5       then min[H] ← min[H2]
6   n[H] ← n[H1] + n[H2]
7   free the objects H1 and H2
8   return H
```

# Fibonacci Heap (Operations)

## Uniting two Fibonacci heaps

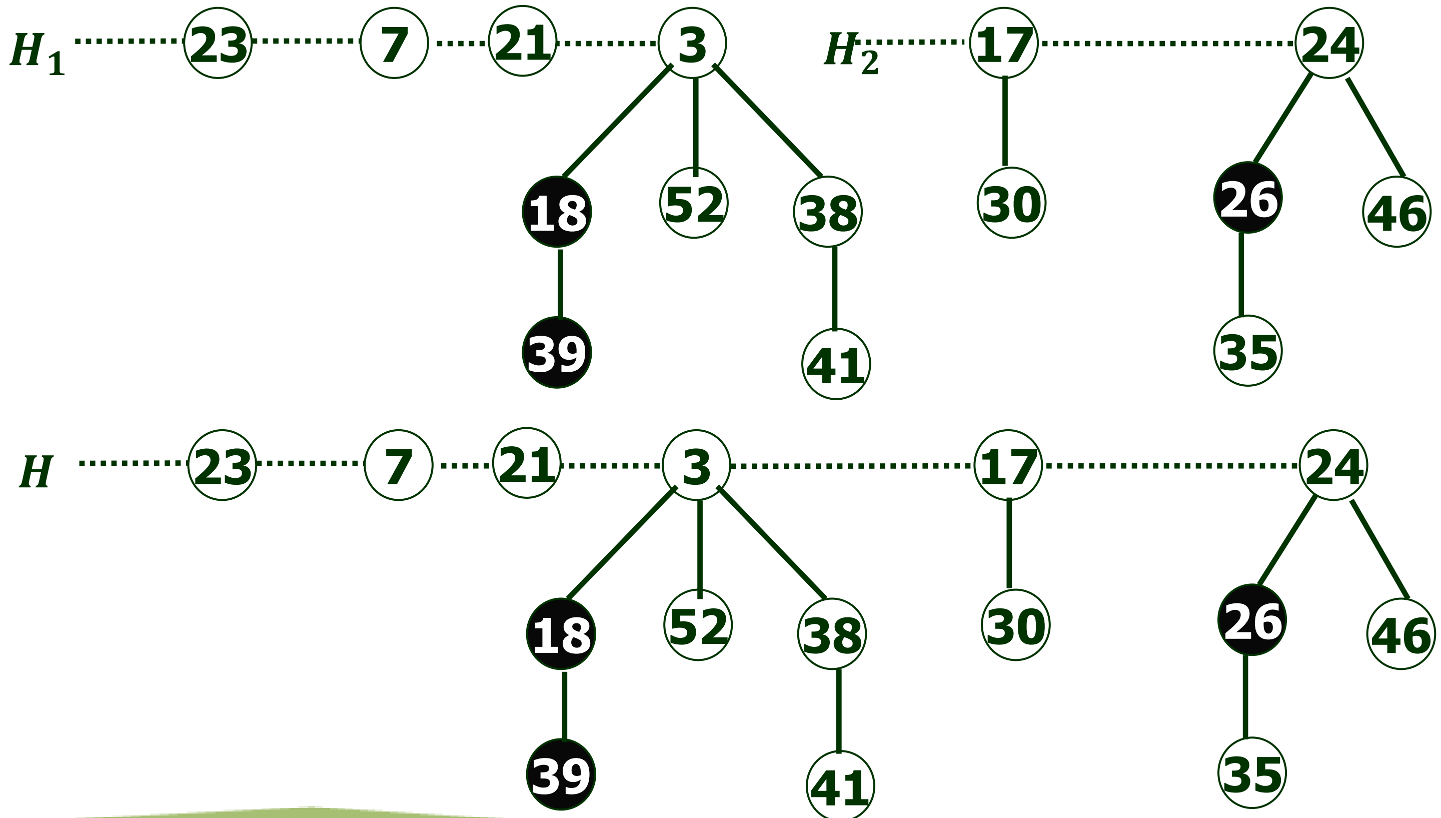
Example : Apply FIB-HEAP-UNION($H_1$, $H_2$) for uniting the following two Fibonacci Heaps



# Fibonacci Heap (Operations)

## Uniting two Fibonacci heaps

Example : Apply FIB-HEAP-UNION(H1, H2) for uniting the following two Fibonacci Heaps



# Fibonacci Heap (Operations)

## Uniting two Fibonacci heaps (Analysis)

```
FIB-HEAP-UNION(H1, H2)
1   H ← MAKE-FIB-HEAP()
2   min[H] ← min[H1]
3   Concatenate the root list of H2 with the root list of H
4   if (min[H1] = NIL) or (min[H2] ≠ NIL and min[H2] < min[H1])
5       then min[H] ← min[H2]
6   n[H] ← n[H1] + n[H2]
7   free the objects H1 and H2
8   return H
```

Lines 1-3 concatenate the root lists of H1 and H2 into a new root list H.

Lines 2, 4, and 5 set the minimum node of H ,

and line 6 sets n[H] to the total number of nodes.

The Fibonacci heap objects H1 and H2 are freed in line 7, and line 8 returns the resulting Fibonacci heap H.

# Fibonacci Heap (Operations)

## Uniting two Fibonacci heaps (Analysis)

As in the FIB-HEAP-INSERT procedure, no consolidation of trees occurs. The change in potential is

$= \Phi(H) - (\Phi(H1) + \Phi(H2))$

$= (t(H) + 2m(H)) - ((t(H1) + 2\,m(H1)) + (t(H2) + 2\,m(H2)))$

$= 0,$

because $t(H) = t(H1) + t(H2)$ and $m(H) = m(H1) + m(H2)$.

The amortized cost of FIB-HEAPUNION is therefore equal to its $O(1)$ actual cost.

## Fibonacci Heap (Operations)

### Extracting the minimum node

The process of extracting the minimum node is the most complicated of the operations presented in this section.

**Basic Idea:**

Step 1: Delete min and concatenate its children into root list.

Step 2: Consolidate trees so that no two roots have same degree.

## Fibonacci Heap (Operations)

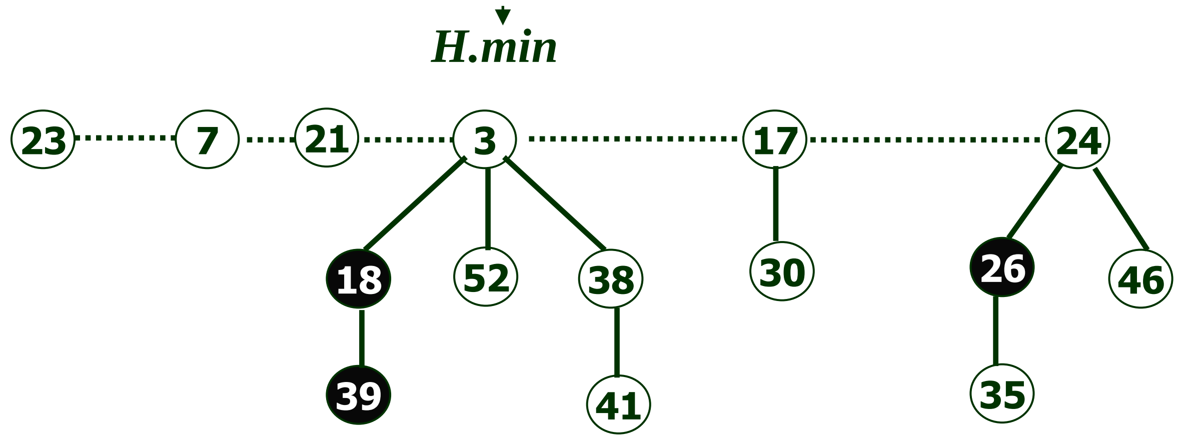
### Extracting the minimum node

```
FIB-HEAP-EXTRACT-MIN(H)
 1  z ← min[H]
 2  if z ≠ NIL
 3        then for each child x of z
 4             do add x to the root list of H
 5             p[x] ← NIL
 6             remove z from the root list of H
 7        if z = right[z]
 8             then min[H] ← NIL
 9        else min[H] ← right[z]
10             CONSOLIDATE(H) // merge the tree with same degree
11  n[H] ← n[H] - 1
12  return z.
```

# Fibonacci Heap (Operations)

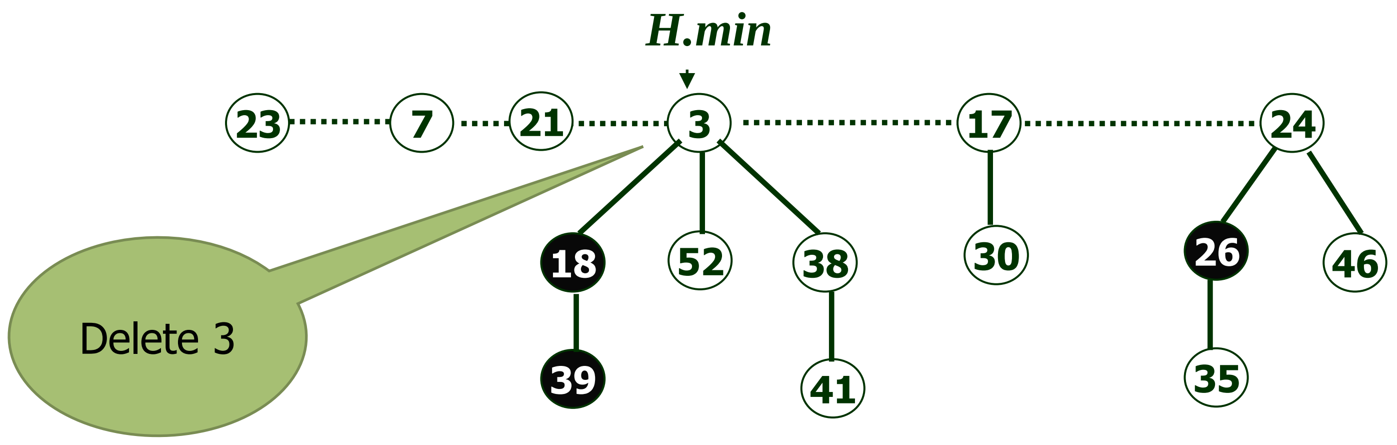
## Extracting the minimum node

Example: Extract the minimum element from the following Fibonacci heap.



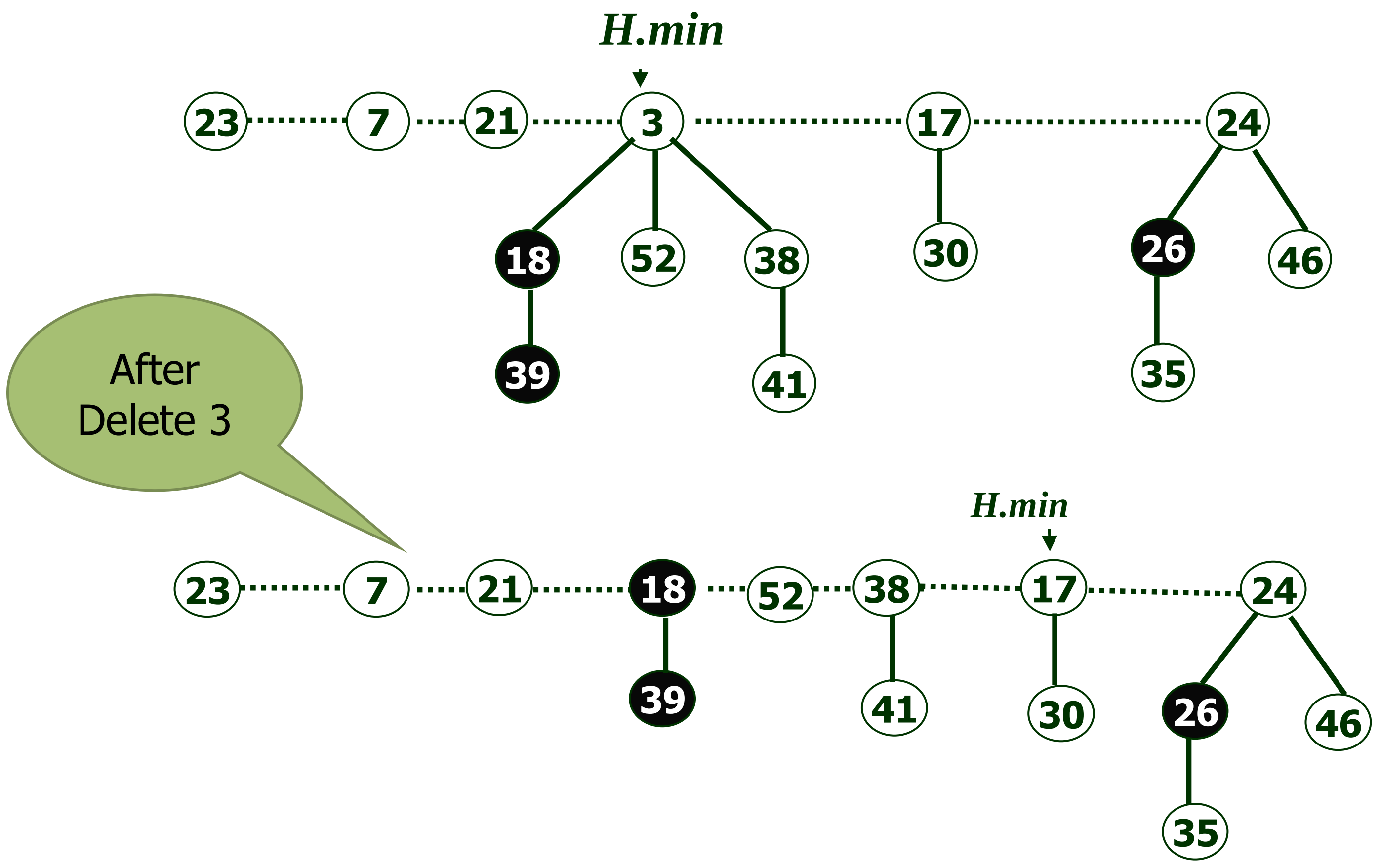
# Fibonacci Heap (Operations)

## Extracting the minimum node



# Fibonacci Heap (Operations)

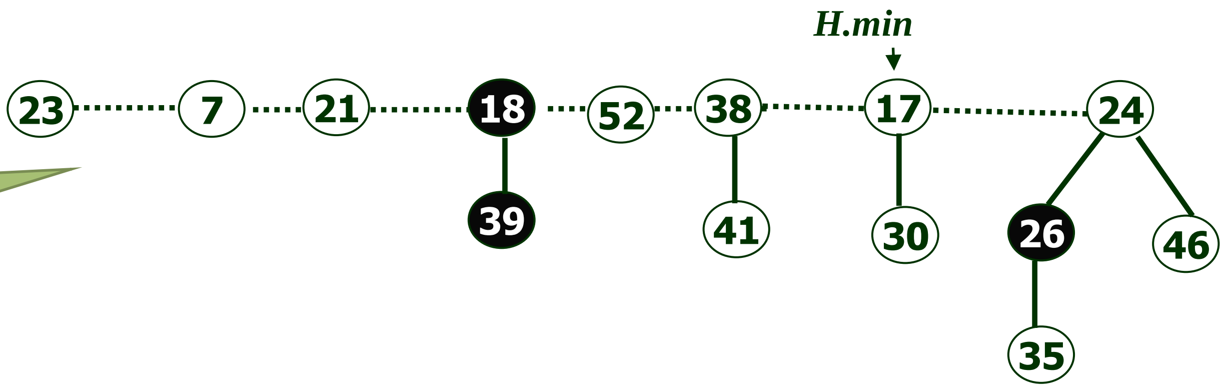
## Extracting the minimum node



# Fibonacci Heap (Operations)

## Extracting the minimum node

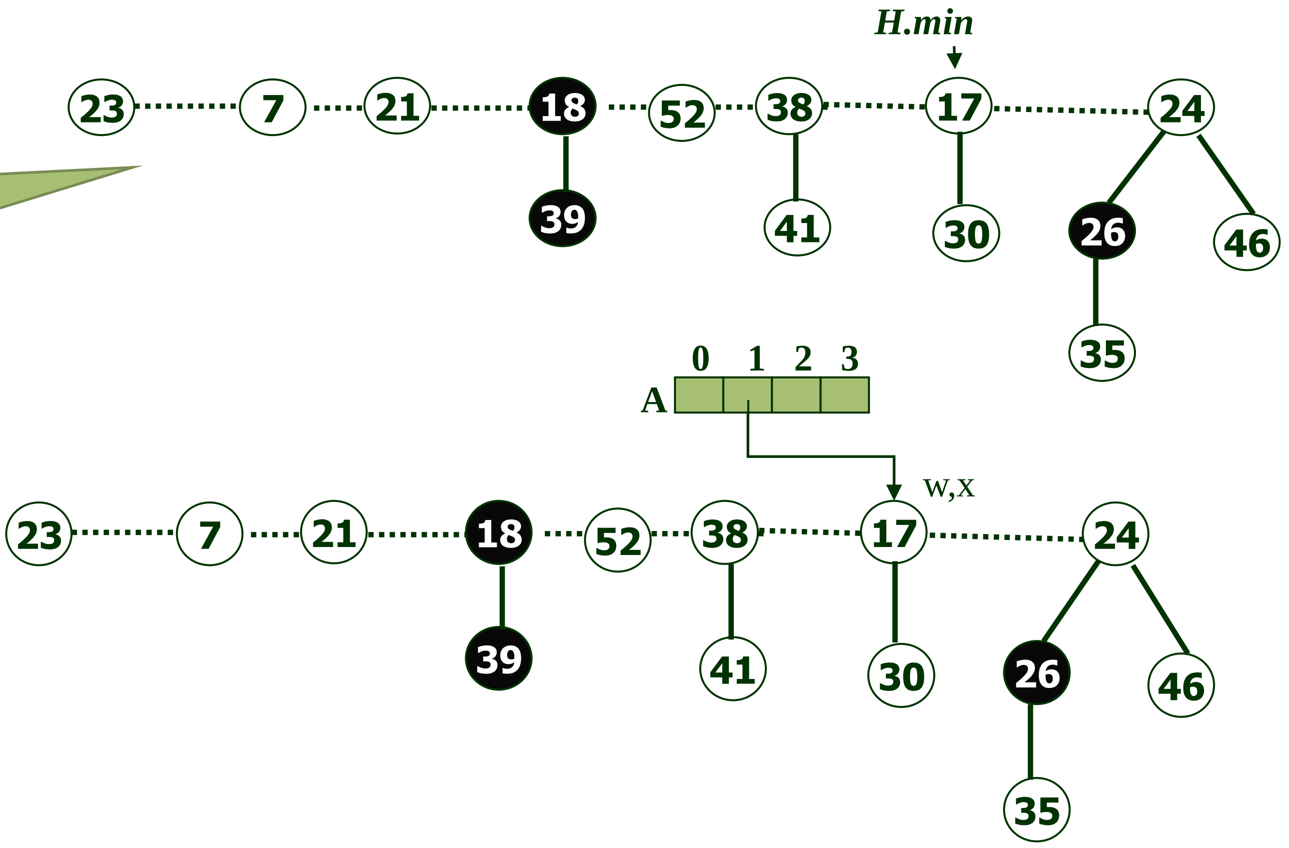
Apply Consolidate()



# Fibonacci Heap (Operations)

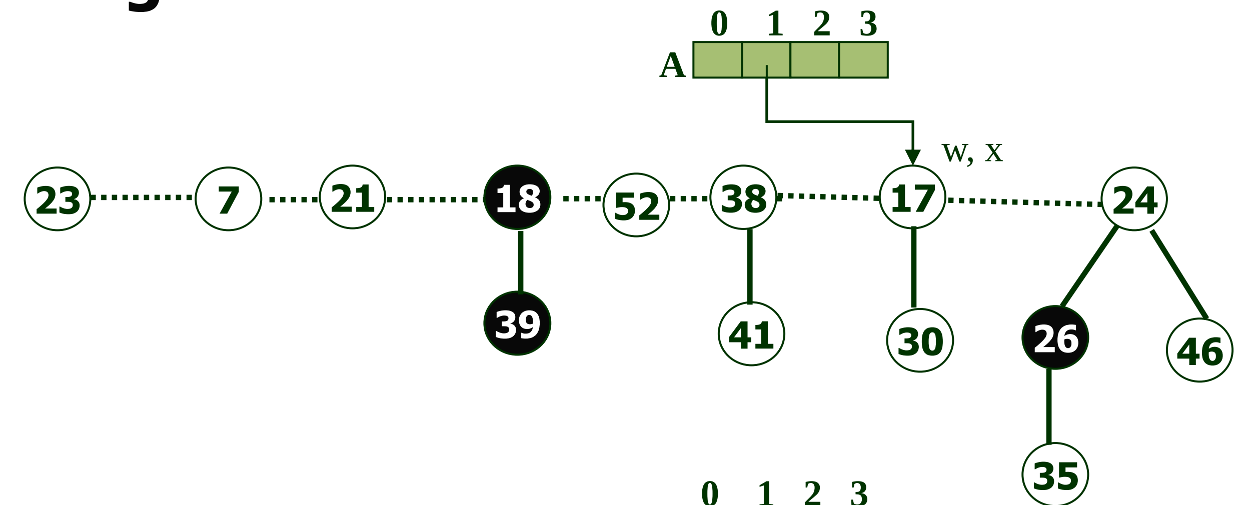
## Extracting the minimum node

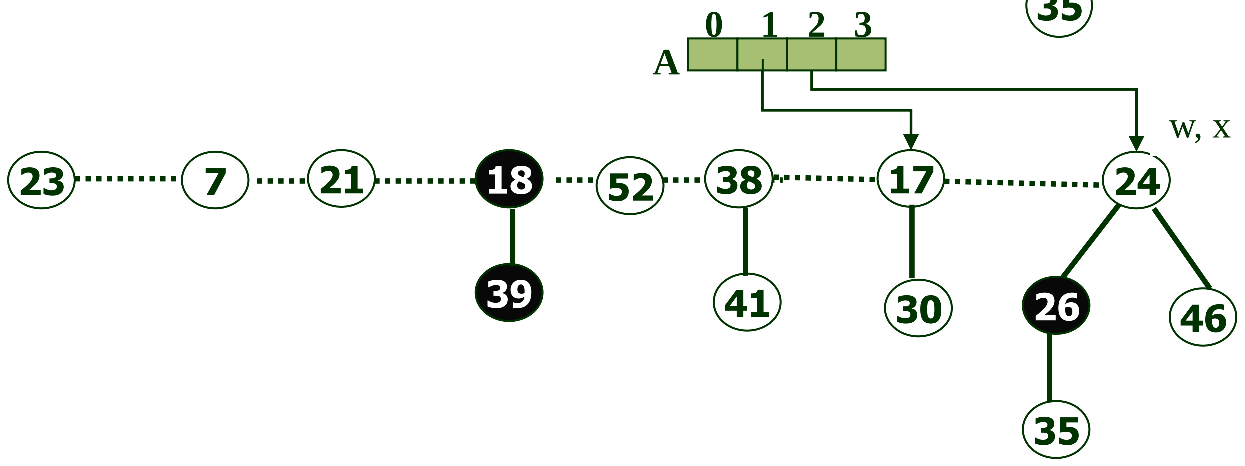
Apply Consolidate()



# Fibonacci Heap (Operations)

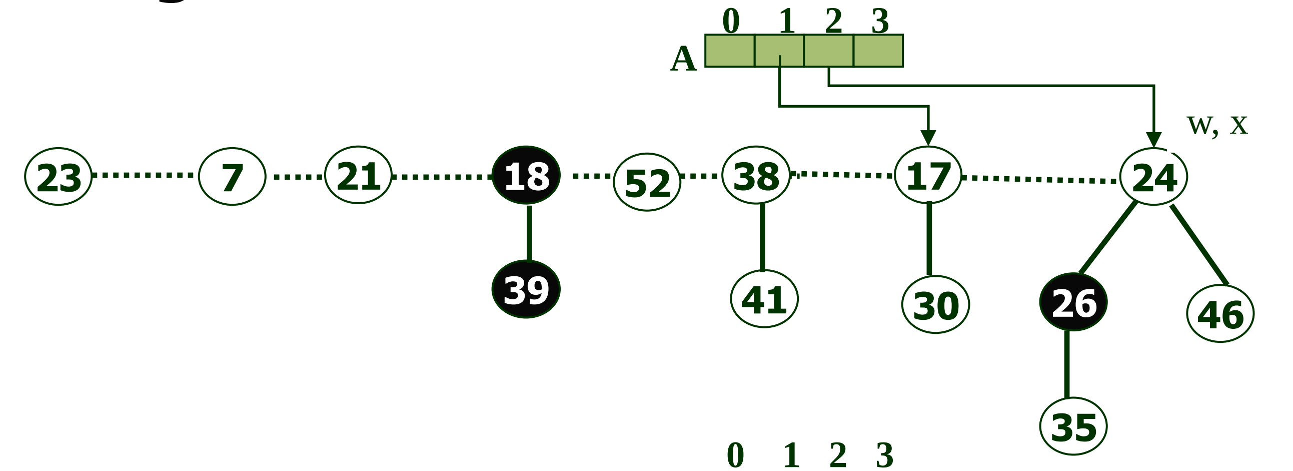
## Extracting the minimum node

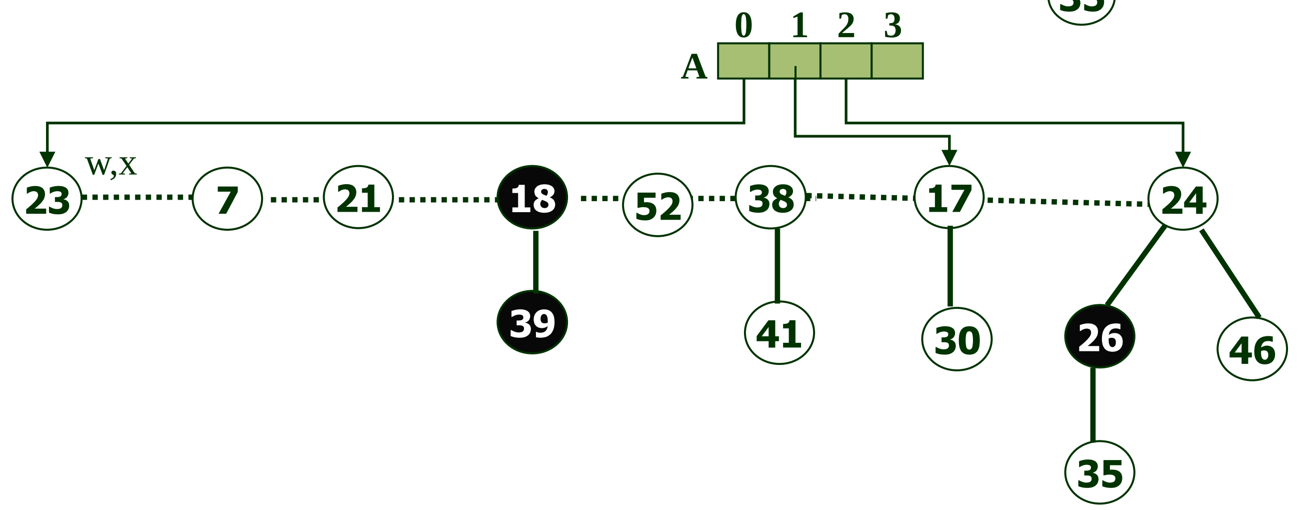




# Fibonacci Heap (Operations)

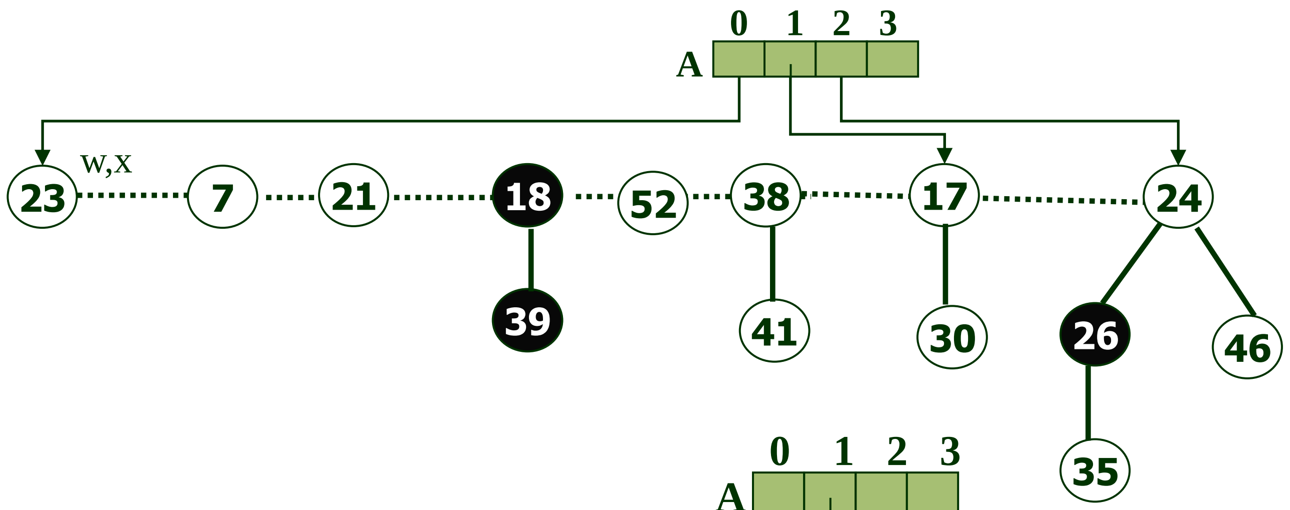
## Extracting the minimum node

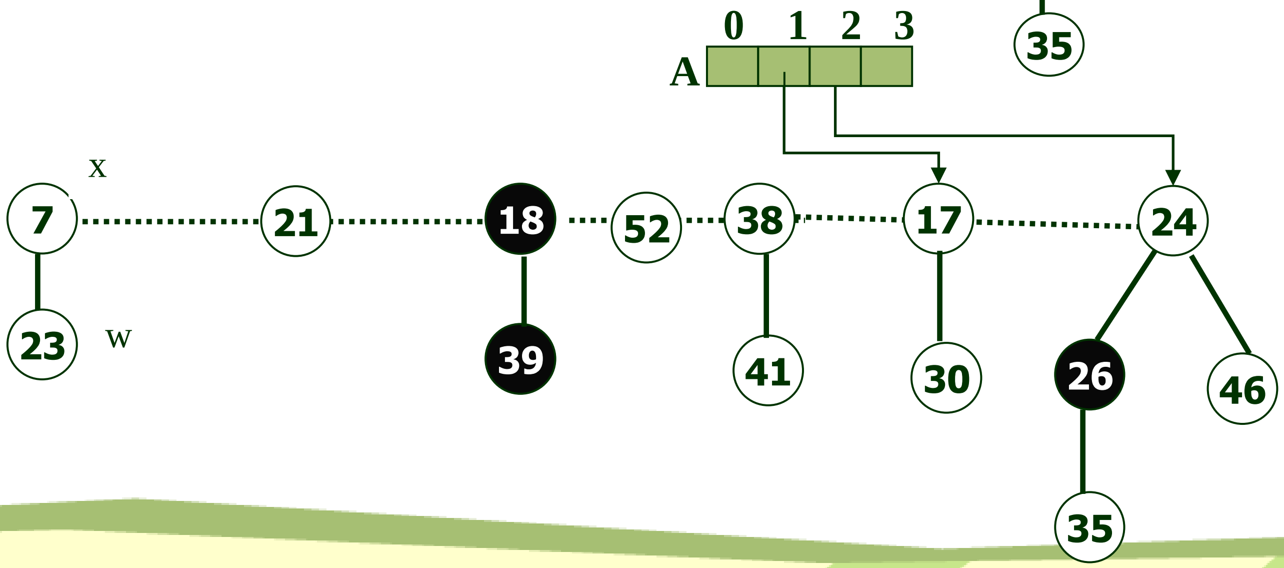




# Fibonacci Heap (Operations)

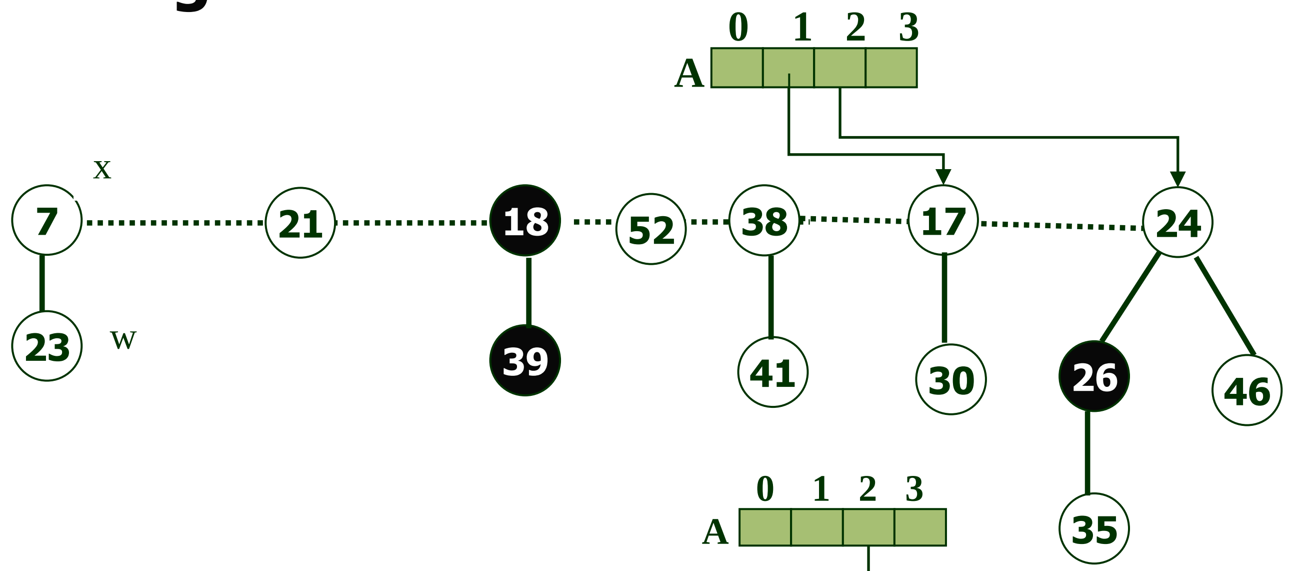
## Extracting the minimum node

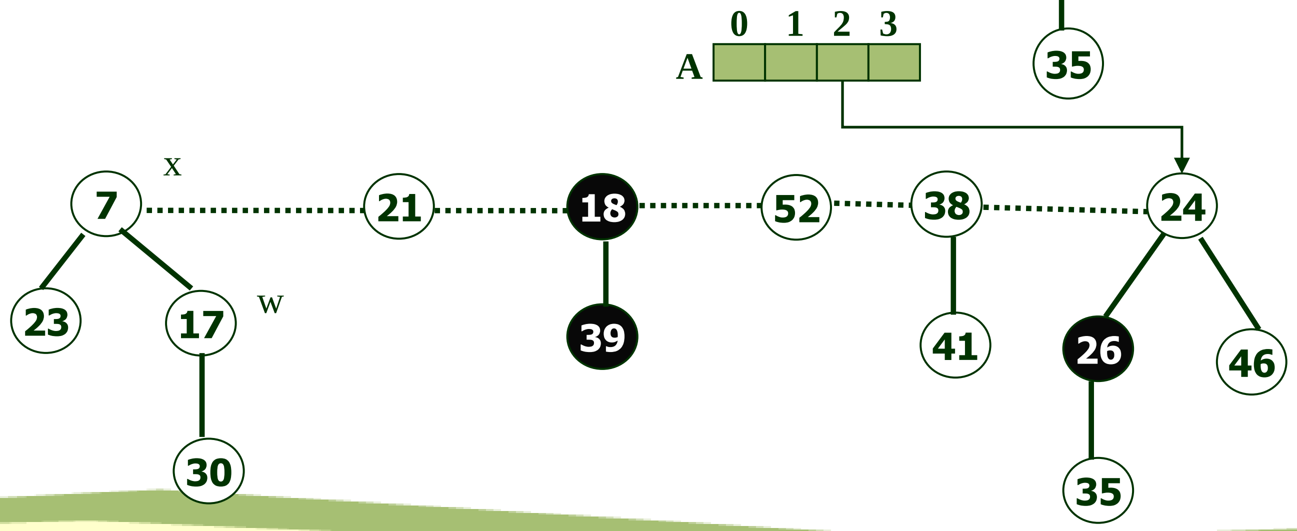




# Fibonacci Heap (Operations)

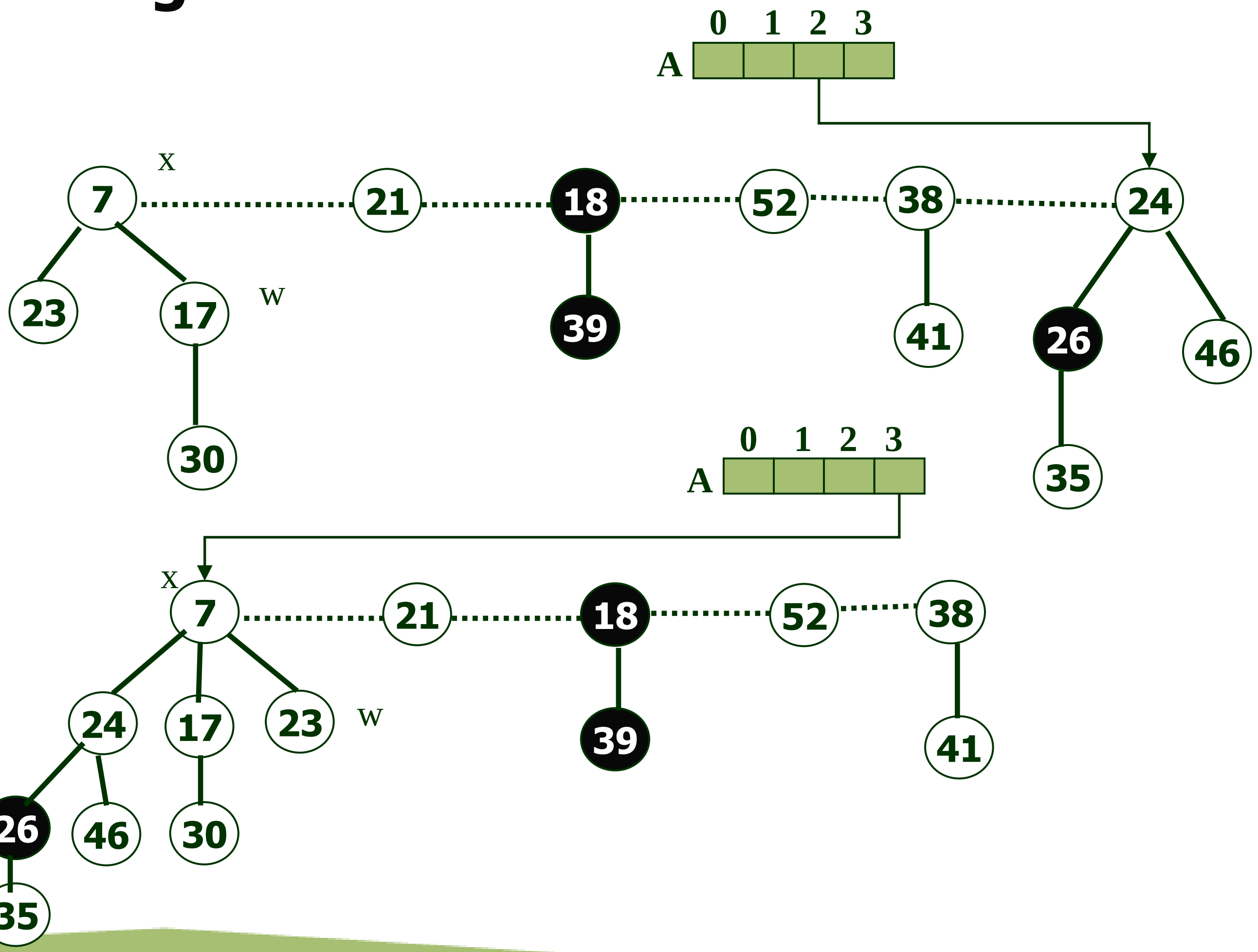
## Extracting the minimum node





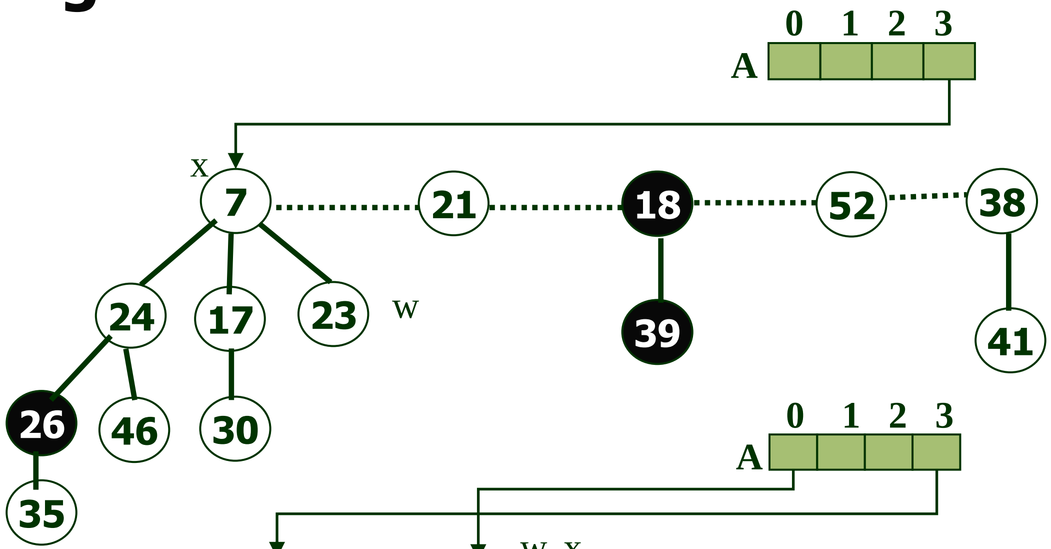
# Fibonacci Heap (Operations)

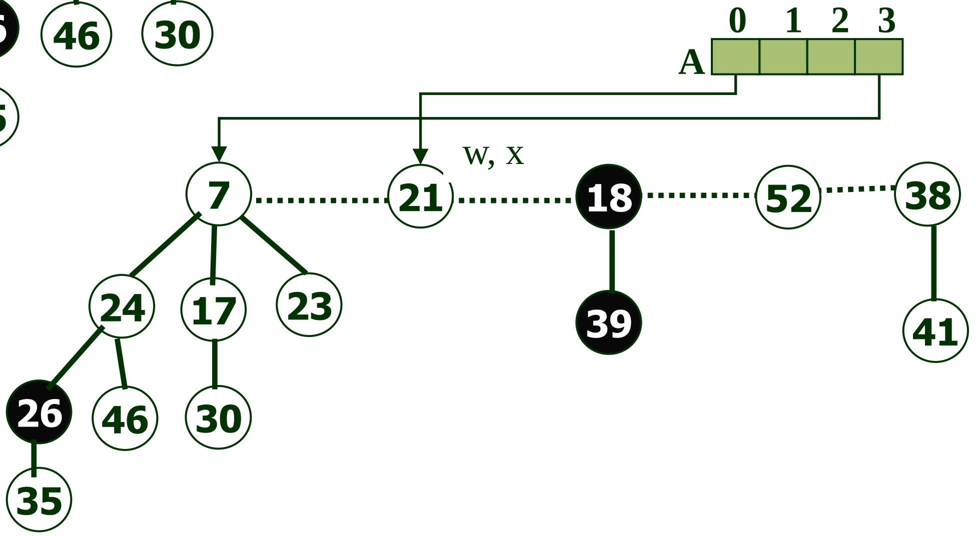
## Extracting the minimum node



# Fibonacci Heap (Operations)

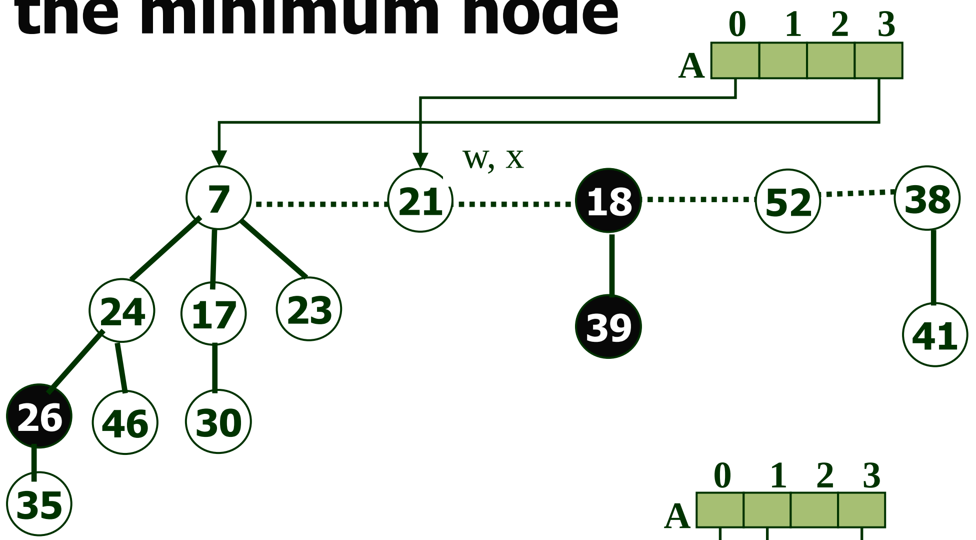
## Extracting the minimum node

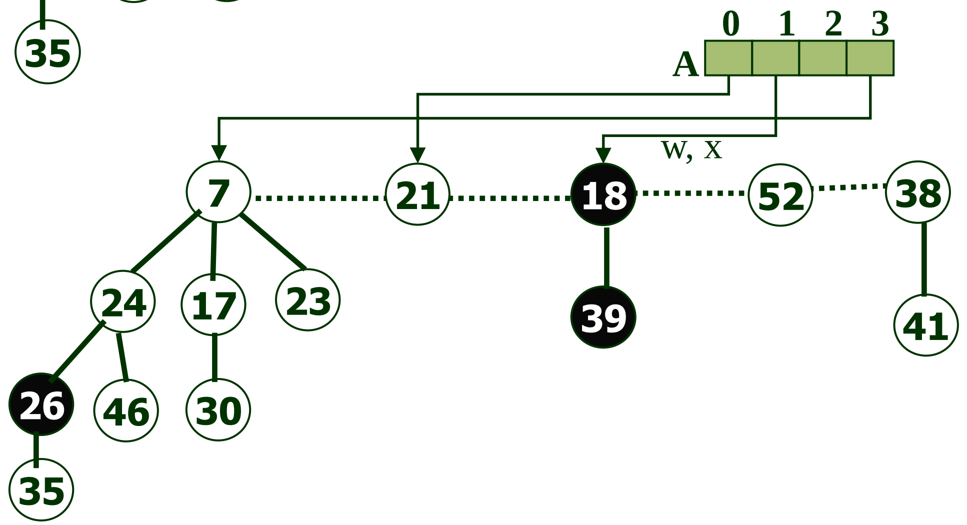




# Fibonacci Heap (Operations)

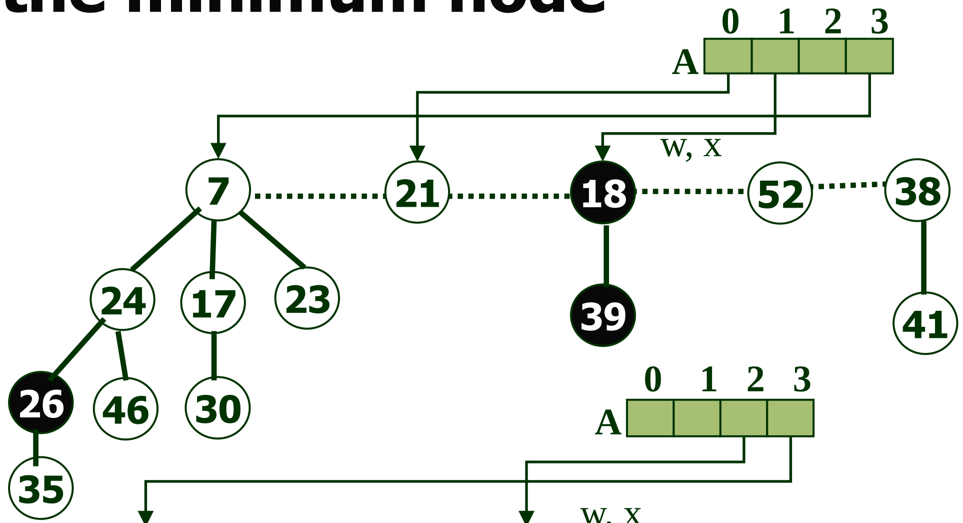
## Extracting the minimum node

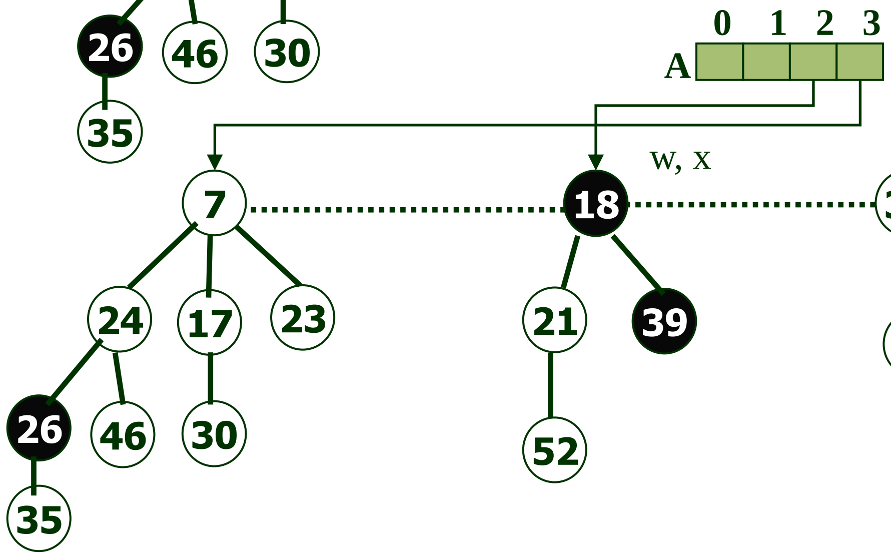




# Fibonacci Heap (Operations)

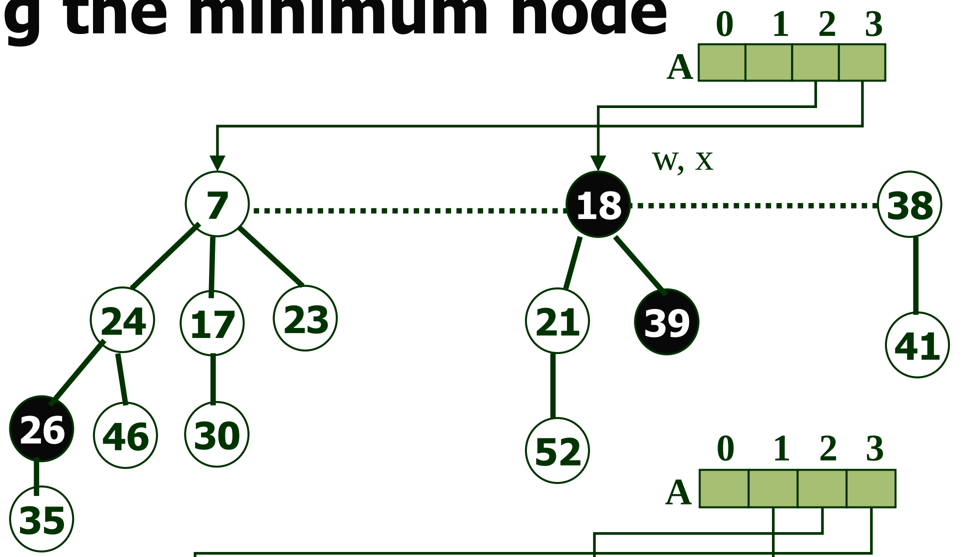
## Extracting the minimum node

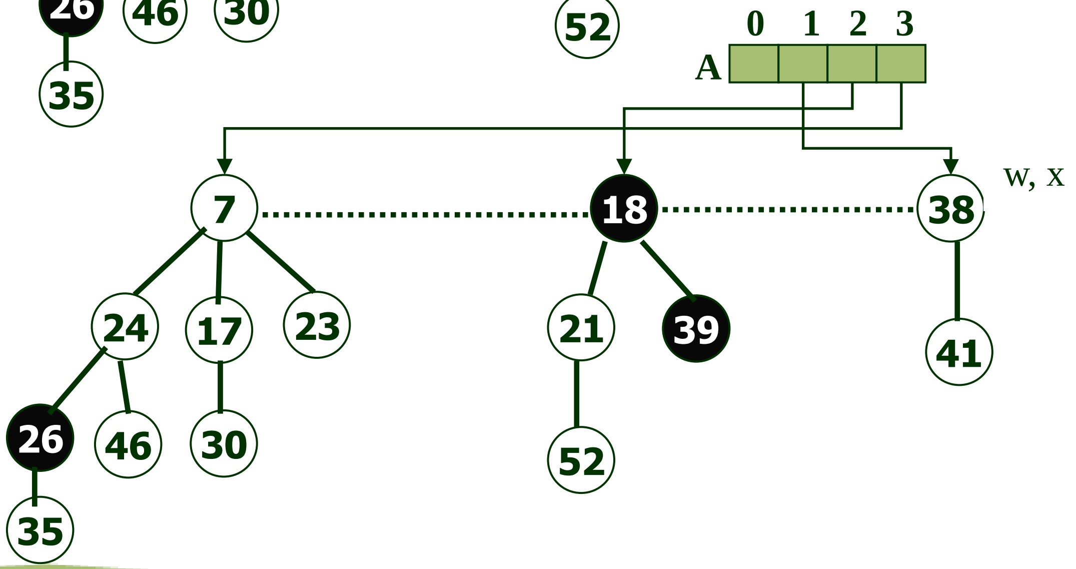




# Fibonacci Heap (Operations)

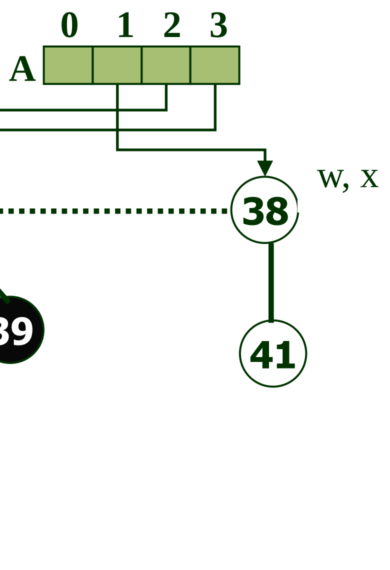
## Extracting the minimum node

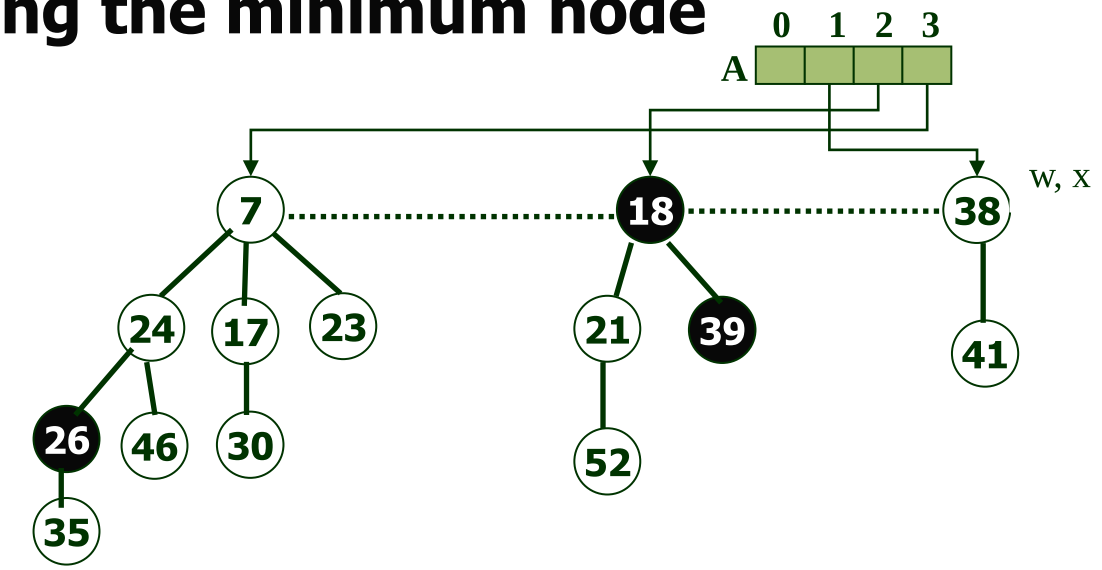


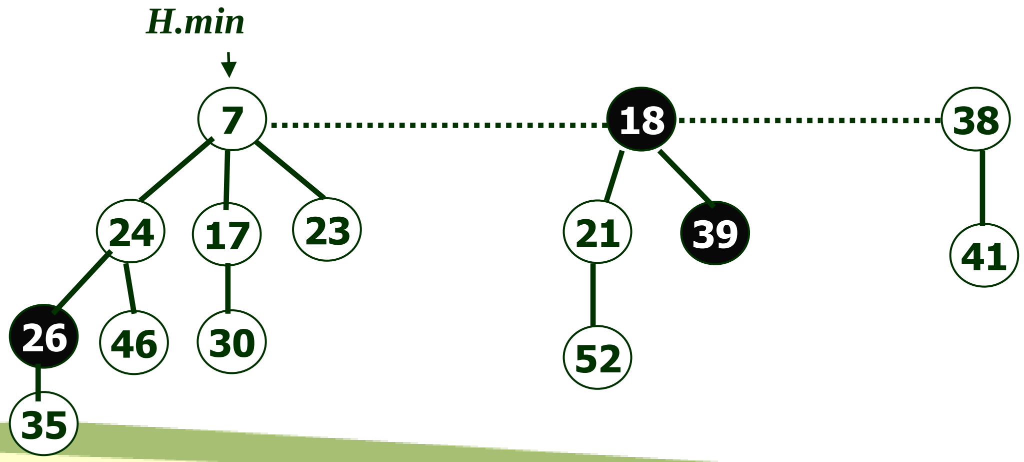


# Fibonacci Heap (Operations)

## Extracting the minimum node







# Fibonacci Heap (Operations)

## Extracting the minimum node

```
CONSOLIDATE(H)
1  for i ← 0 to D(n[H])
2      do A[i] ← NIL
3  for each node w in the root list of H
4      do x ← w
5          d ← degree[x]
6          while A[d] ≠ NIL
7              do y ← A[d]    ▹ Another node with the same degree as x.
8                  if key[x] > key[y]
9                      then exchange x ↔ y
10                 FIB-HEAP-LINK(H, y, x)
11                 A[d] ← NIL
12                 d ← d + 1
13         A[d] ← x
```

## Fibonacci Heap (Operations)

### Extracting the minimum node

```
14 min[H] ← NIL
15 for i ← 0 to D(n[H])
16    do if A[i] ≠ NIL
17          then add A[i] to the root list of H
18                if min[H] = NIL or key[A[i]] < key[min[H]]
19                   then min[H] ← A[i]
```

```
FIB-HEAP-LINK(H, y, x)
1 remove y from the root list of H
2 make y a child of x, incrementing degree[x]
3 mark[y] ← FALSE
```

# Fibonacci Heap (Operations)

## Extracting the minimum node(Analysis)

**Notation:**

$D(n) = \max degree\ of\ any\ node\ in\ Fibonacci\ heap\ with\ n\ nodes.$
$t(H) = Number\ of\ trees\ in\ heap\ H.$
$m(H) = the\ number\ of\ marked\ nodes\ in\ H.$

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

Actual cost :

$O\big(D(n)\big)$: for loop in Fib-Heap-Extract-Min

After extracting a minimum node from the Fibonacci heap, the heap contain $t(H) - 1$ trees.

Hence, the current size of root list is : $D(n) + t(H) - 1$

Total actual cost: $O(D(n)) + t(H)$

# Fibonacci Heap (Operations)

## Extracting the minimum node(Analysis)

Potential before extracting : $t(H) + 2m(H)$

Potential after extracting : $\leq\ D(n) + 1 + 2m(H)$

$\therefore$ At most D(n)+1 nodes remain on the list and no nodes become marked.

Thus the amortized cost is at most:

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

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

$$= O(D(n))$$

$$= O(\log n)$$

*[Note: An n node Binomial heap H consists of at most* $\lfloor log\ n \rfloor + 1$ *binomial Tree]*

## Fibonacci Heap (Operations)

### Decreasing a key

This pseudocode for the operation FIB-HEAP-DECREASE-KEY, work with an assumption that removing a node from a linked list does not change any of the structural fields in the removed node.

# Fibonacci Heap (Operations)

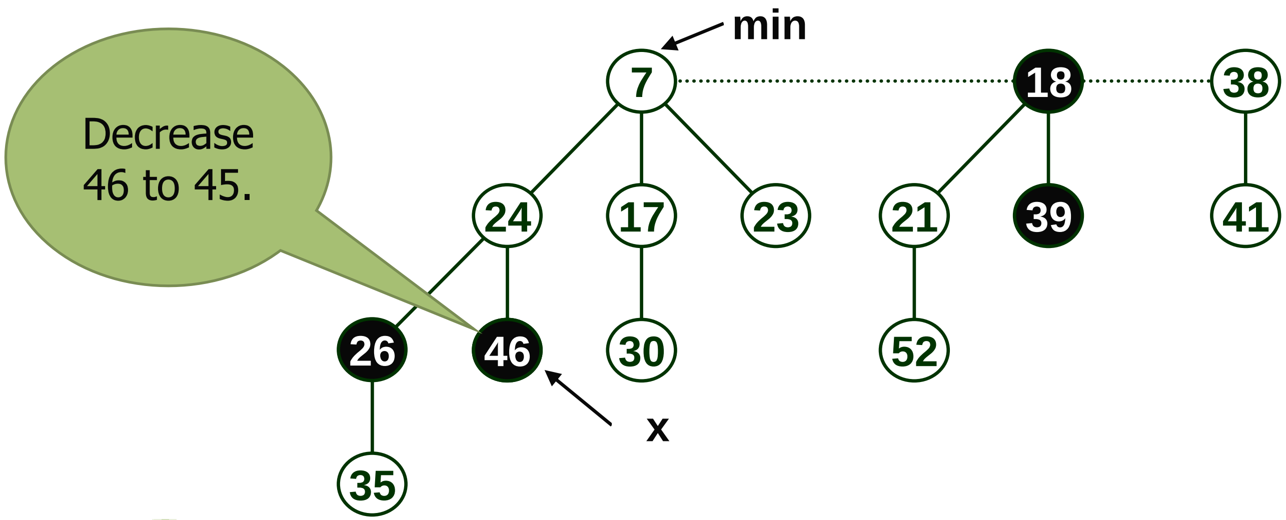
**Decreasing a key**

**Basic Idea:**

Case 0: min-heap property not violated.

- decrease key of x to k
- change heap min pointer if necessary

Example:



# Fibonacci Heap (Operations)

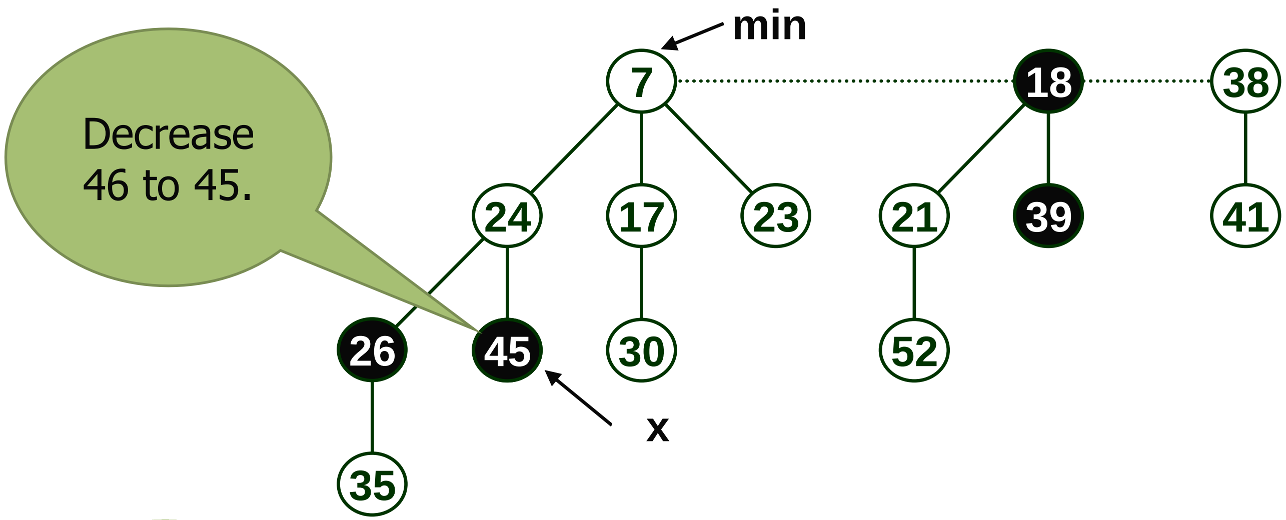
## Decreasing a key

## Basic Idea:

Case 0: min-heap property not violated.

- decrease key of x to k
- change heap min pointer if necessary

Example:



# Fibonacci Heap (Operations)

## Decreasing a key

## Basic Idea:

Case 1: parent of x is unmarked.

- decrease key of x to k
- cut off link between x and its parent
- mark parent
- add tree rooted at x to root list, updating heap min pointer

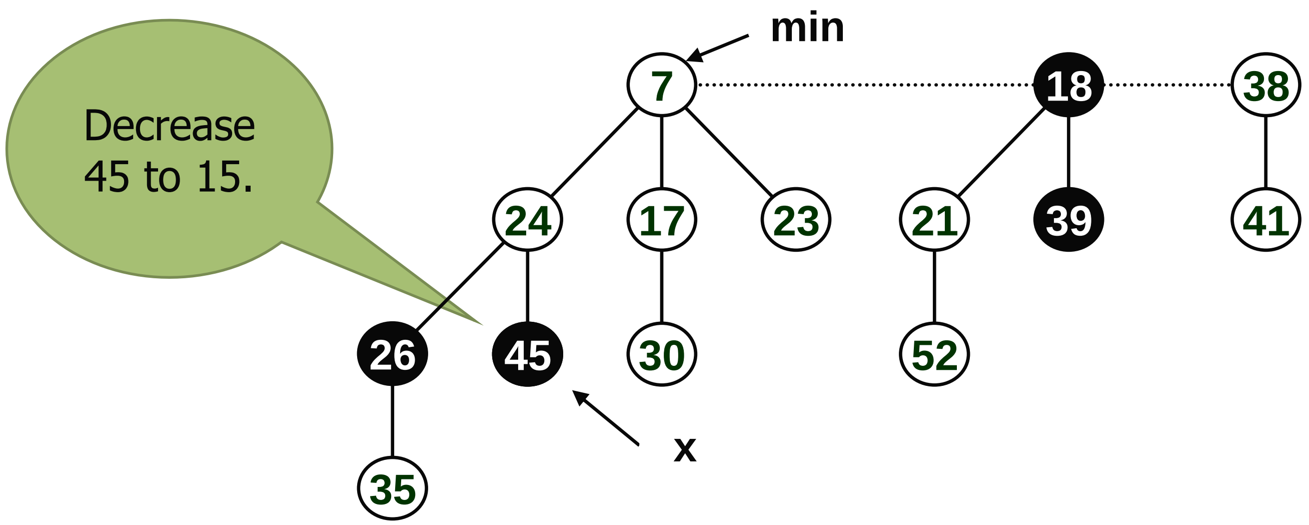
# Fibonacci Heap (Operations)

## Decreasing a key and deleting node

## Basic Idea:

Case 1: parent of x is unmarked.

- decrease key of x to k
- cut off link between x and its parent
- mark parent
- add tree rooted at x to root list, updating heap min pointer



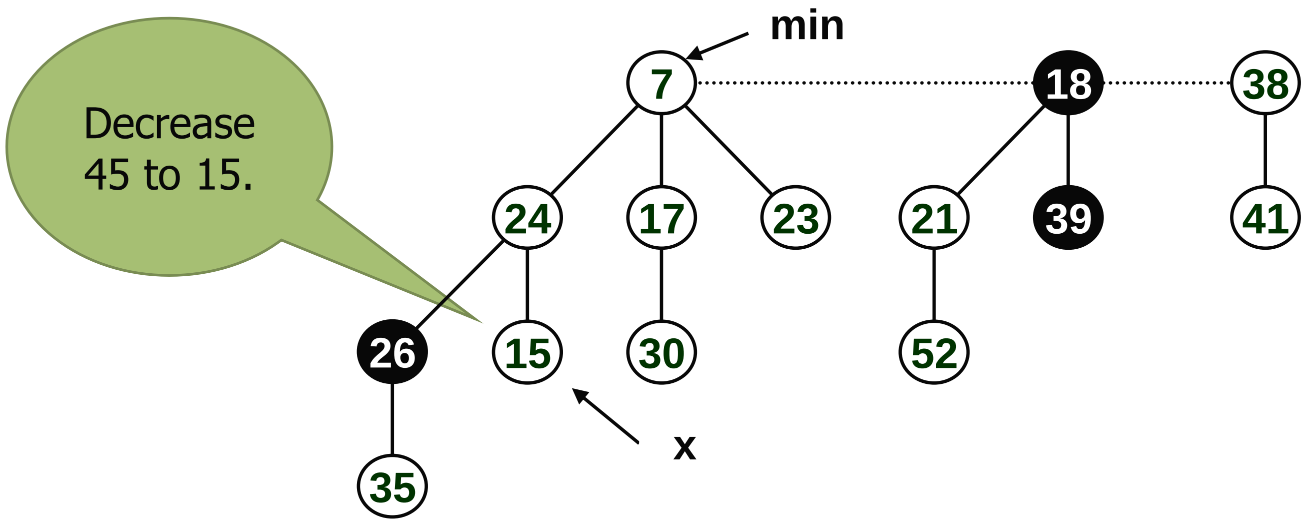
# Fibonacci Heap (Operations)

## Decreasing a key

## Basic Idea:

Case 1: parent of x is unmarked.

- decrease key of x to k
- cut off link between x and its parent
- mark parent
- add tree rooted at x to root list, updating heap min pointer



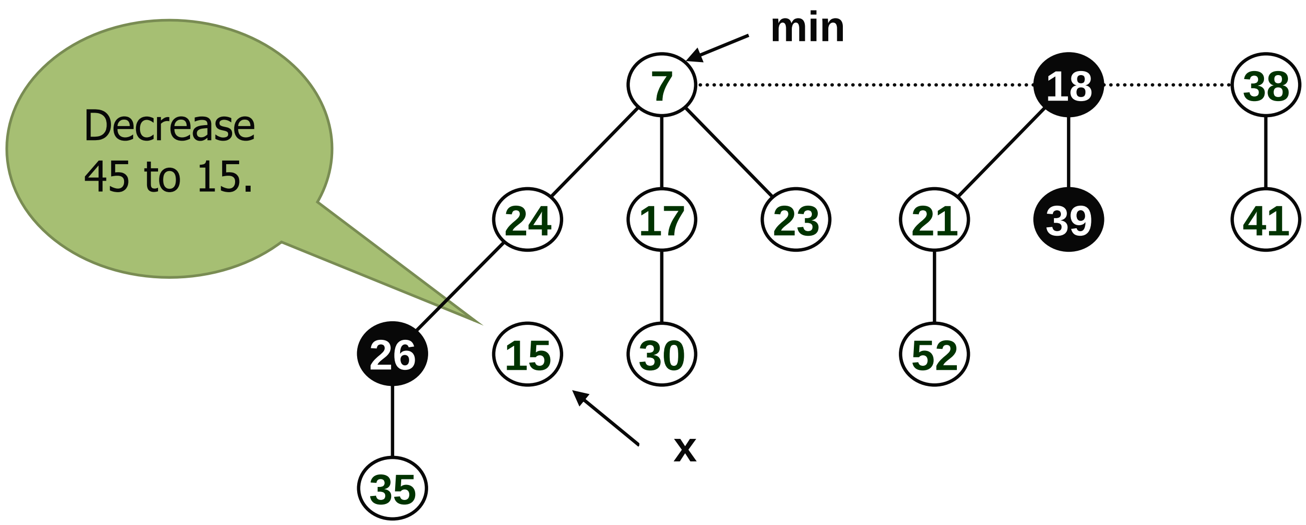
# Fibonacci Heap (Operations)

## Decreasing a key

## Basic Idea:

Case 1: parent of x is unmarked.

- decrease key of x to k
- cut off link between x and its parent
- mark parent
- add tree rooted at x to root list, updating heap min pointer



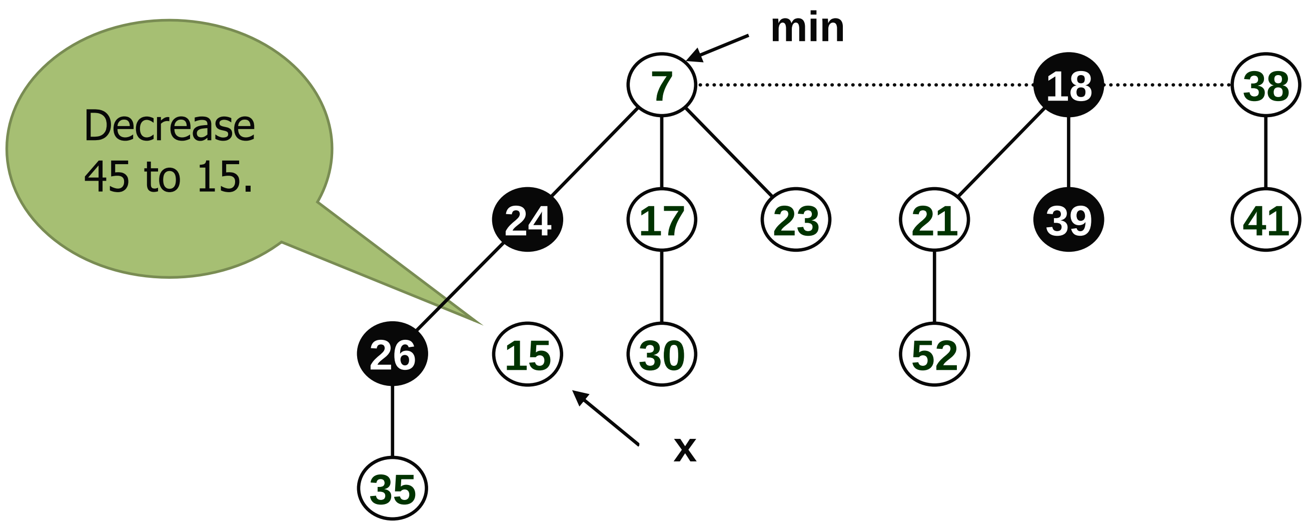
## Fibonacci Heap (Operations)

### Decreasing a key

### Basic Idea:

Case 1: parent of x is unmarked.

- decrease key of x to k
- cut off link between x and its parent
- mark parent
- add tree rooted at x to root list, updating heap min pointer



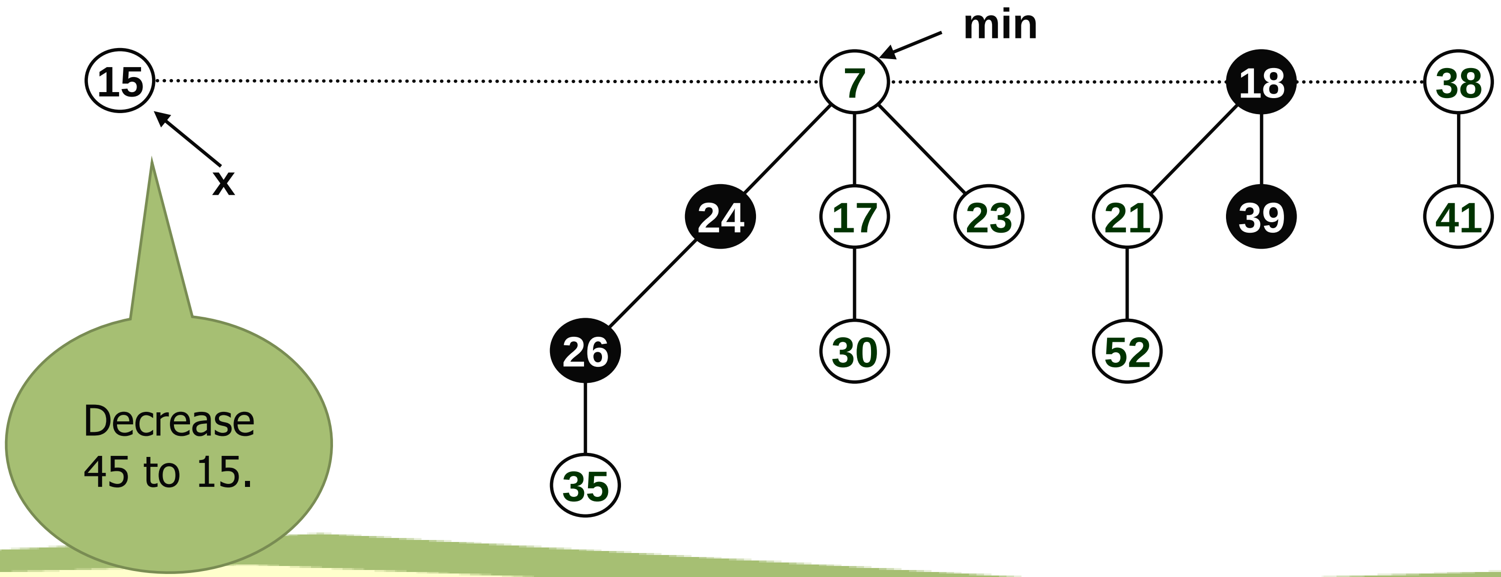
# Fibonacci Heap (Operations)

## Decreasing a key

## Basic Idea:

Case 1: parent of x is unmarked.

- decrease key of x to k
- cut off link between x and its parent
- mark parent
- add tree rooted at x to root list, updating heap min pointer



## Fibonacci Heap (Operations)

**Decreasing a key**

**Basic Idea:**

Case 2: parent of x is marked.

- decrease key of x to k
- cut off link between x and its parent p[x], and add x to root list
- cut off link between p[x] and p[p[x]], add p[x] to root list
  - > If p[p[x]] unmarked, then mark it.
  - > If p[p[x]] marked, cut off p[p[x]], unmark, and repeat.

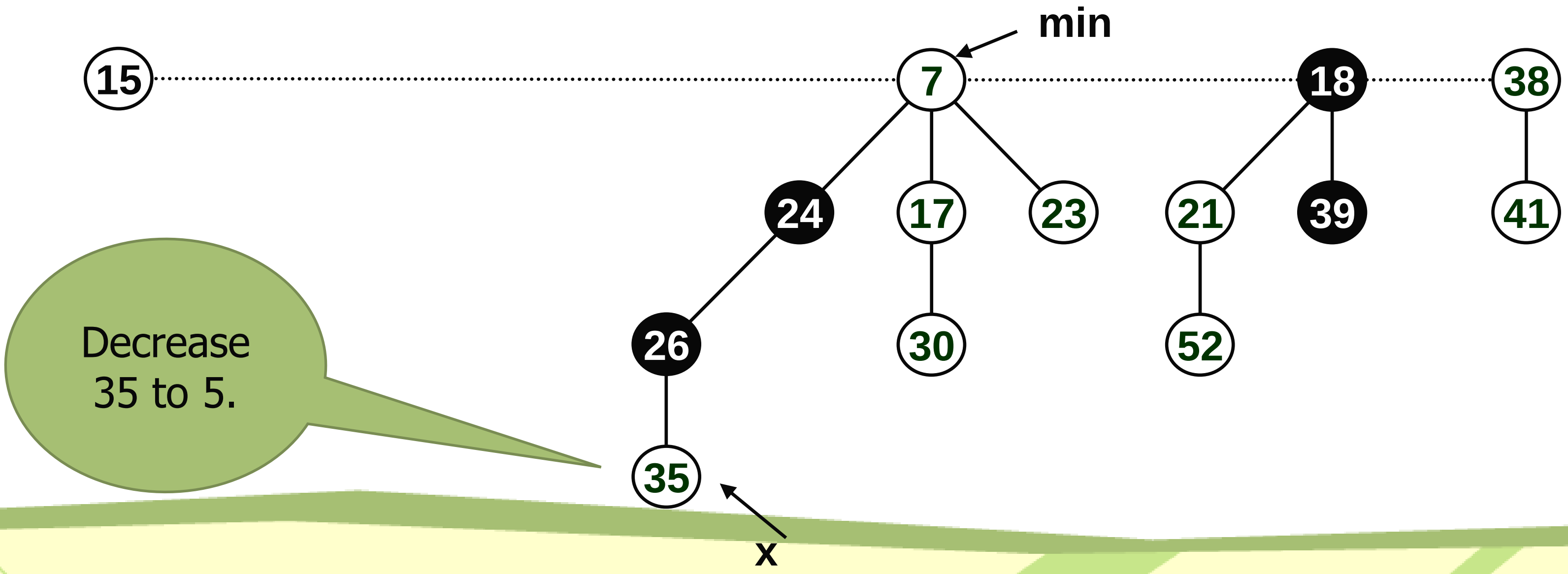
# Fibonacci Heap (Operations)

## Decreasing a key

## Basic Idea:

Case 2: parent of x is marked.

- decrease key of x to k
- cut off link between x and its parent p[x], and add x to root list
- cut off link between p[x] and p[p[x]], add p[x] to root list
  - > If p[p[x]] unmarked, then mark it.
  - > If p[p[x]] marked, cut off p[p[x]], unmark, and repeat.



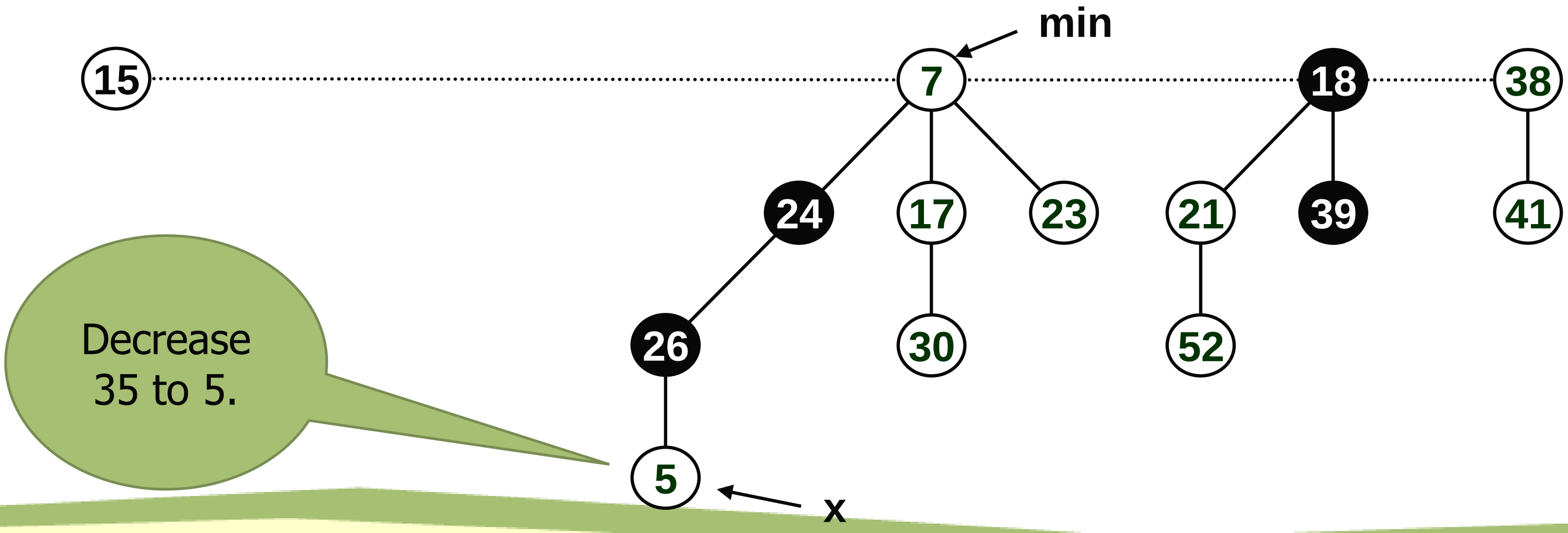
# Fibonacci Heap (Operations)

## Decreasing a key

## Basic Idea:

Case 2: parent of x is marked.

- decrease key of x to k
- cut off link between x and its parent p[x], and add x to root list
- cut off link between p[x] and p[p[x]], add p[x] to root list
  - > If p[p[x]] unmarked, then mark it.
  - > If p[p[x]] marked, cut off p[p[x]], unmark, and repeat.



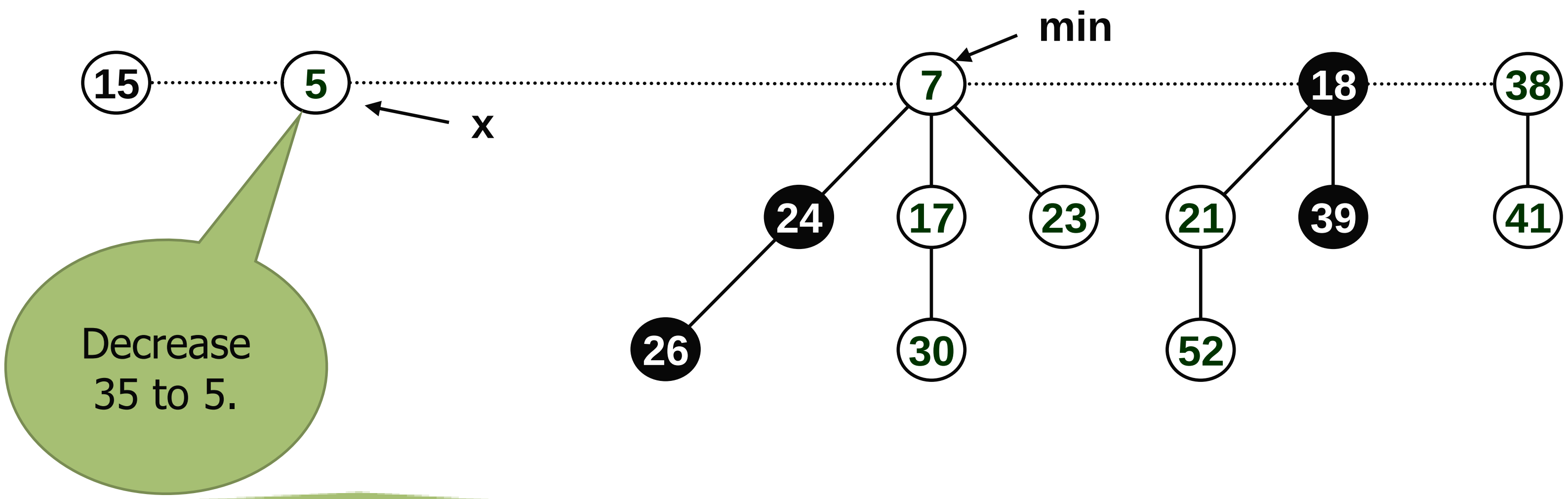
# Fibonacci Heap (Operations)

## Decreasing a key

## Basic Idea:

Case 2: parent of x is marked.

- decrease key of x to k
- cut off link between x and its parent p[x], and add x to root list
- cut off link between p[x] and p[p[x]], add p[x] to root list
  - > If p[p[x]] unmarked, then mark it.
  - > If p[p[x]] marked, cut off p[p[x]], unmark, and repeat.



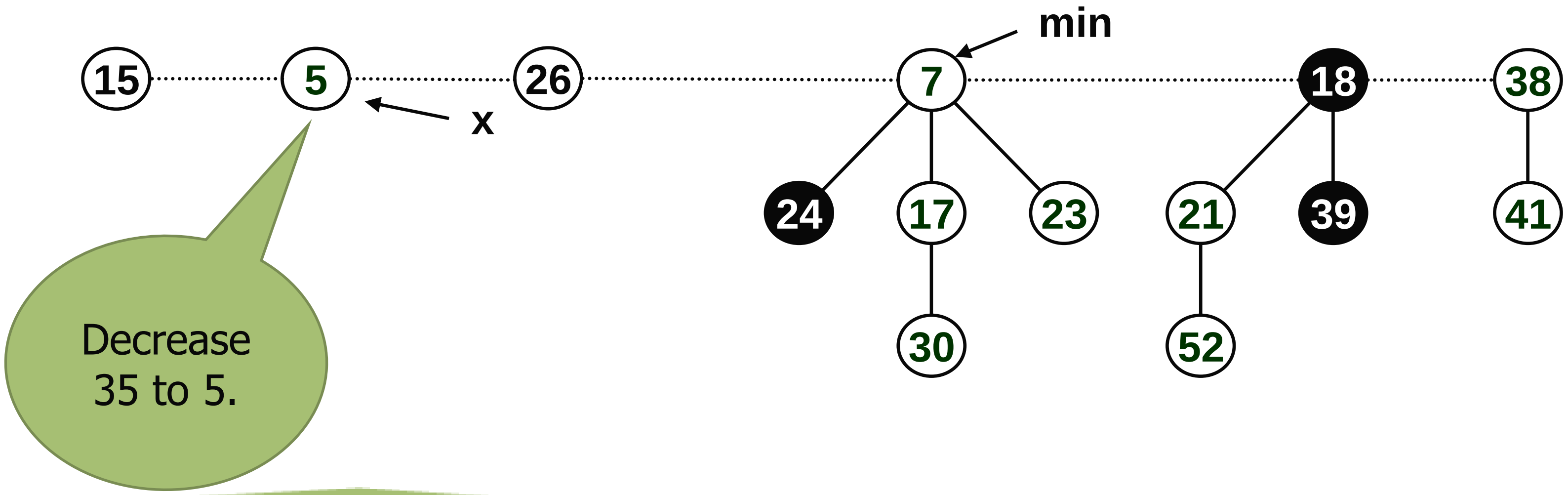
# Fibonacci Heap (Operations)

## Decreasing a key

## Basic Idea:

Case 2: parent of x is marked.

- decrease key of x to k
- cut off link between x and its parent p[x], and add x to root list
- cut off link between p[x] and p[p[x]], add p[x] to root list
  - > If p[p[x]] unmarked, then mark it.
  - > If p[p[x]] marked, cut off p[p[x]], unmark, and repeat.



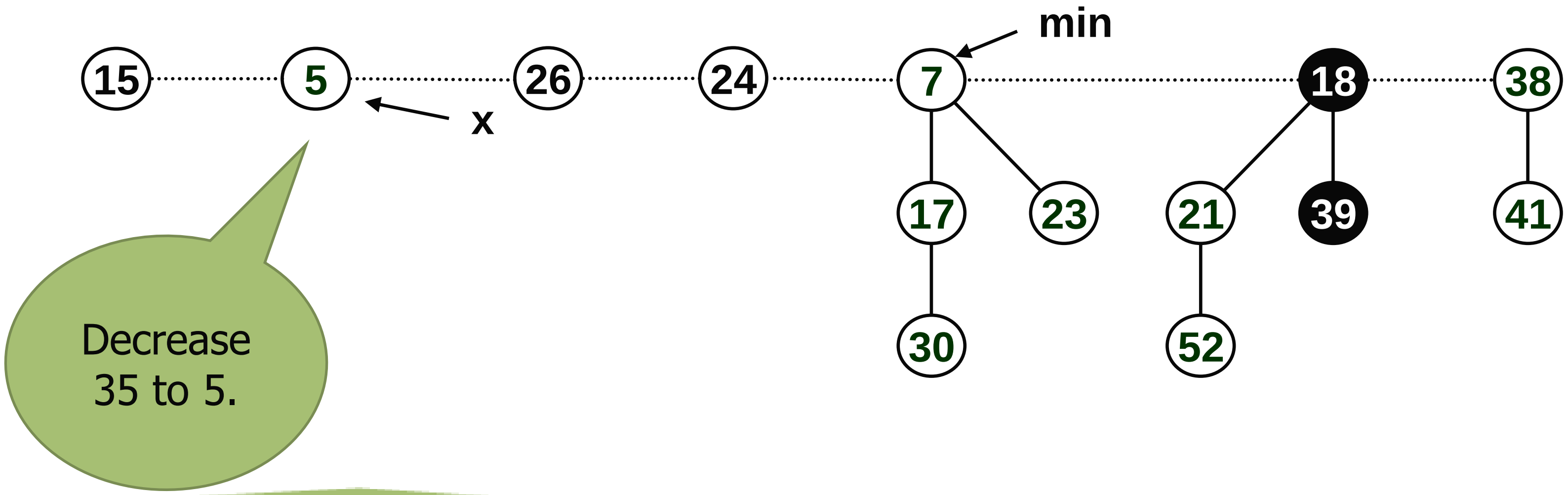
# Fibonacci Heap (Operations)

## Decreasing a key

## Basic Idea:

Case 2: parent of x is marked.

- decrease key of x to k
- cut off link between x and its parent p[x], and add x to root list
- cut off link between p[x] and p[p[x]], add p[x] to root list
  - > If p[p[x]] unmarked, then mark it.
  - > If p[p[x]] marked, cut off p[p[x]], unmark, and repeat.



## Fibonacci Heap (Operations)

### Decreasing a key

```
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
```

# Fibonacci Heap (Operations)

## Decreasing a key

```
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)
```

# Fibonacci Heap (Operations)

## Decreasing a key (Analysis)

- The FIB-HEAP-DECREASE-KEY procedure takes O(1) time, plus the time to perform the cascading cuts.
- Suppose that CASCADING-CUT is recursively called c times from a given invocation of FIB-HEAP-DECREASE-KEY.
- Each call of CASCADING-CUT takes O(1) time exclusive of recursive calls.
- Thus, the actual cost of FIB-HEAP-DECREASE-KEY, including all recursive calls, is O(c)

# Fibonacci Heap (Operations)

## Decreasing a key (Analysis)

- Each recursive call of CASCADING-CUT except for the last one, cuts a marked node and clears the mark bit.

  [Note: Last call of CASCADING-CUT may have marked a node]

- After Decrease-key, there are at most $t(H) + c$ trees, and at most $m(H) - c + 2$ marked nodes.

[Note: c-1 were unmarked by cascading cuts and the last call of CASCADING-CUT may have marked a node]

- Thus the difference in potential cost is:

$$= [t(H) + c + 2(m(H) - c + 2)] - [t(H) + 2m(H)]$$
$$= 4 - c$$

- The amortized cost of Fibonacci heap is:

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

## Fibonacci Heap (Operations)

### Deleting a node

- It is easy to delete a node from an n-node Fibonacci heap in O(D(n)) amortized time, as is
- done by the following pseudocode. We assume that there is no key value of -∞ currently in the Fibonacci heap.

```
FIB-HEAP-DELETE(H, x)
1    FIB-HEAP-DECREASE-KEY(H, x, -∞)
2    FIB-HEAP-EXTRACT-MIN(H)
```

# Fibonacci Heap (Operations)

## Deleting a node

- It is easy to delete a node from an n-node Fibonacci heap in O(D(n)) amortized time, as is
- done by the following pseudocode. We assume that there is no key value of -∞ currently in the Fibonacci heap.

```
FIB-HEAP-DELETE(H, x)
1   FIB-HEAP-DECREASE-KEY(H, x, -∞)
2   FIB-HEAP-EXTRACT-MIN(H)
```

## Deleting a node(Analysis)

The amortized time of FIB-HEAP-DELETE is $O(\lg n)$. [As $D(n) = \lg n$]

# Thank u