Fibonacci Heaps

Fibonacci Heaps

- Binomial heaps support the mergeable heap operations (INSERT, MINIMUM, EXTRACT_MIN, UNION plus, DECREASE_KEY and DELETE) in $O(\lg n)$ worst-case time.
- Fibonacci heaps support the mergeable heap operations that do not involve deleting an element in $O(1)$ amortized time.

Fibonacci Heaps

- Fibonacci heaps are especially desirable when the number of EXTRACT-MIN and DELETE operations is small relative to the number of other operations.
- Fibonacci heaps are loosely based on binomial heaps.
- A collection of trees if neither DECREASE-KEY nor DELETE is ever invoked.
- Each tree is like a binomial tree.

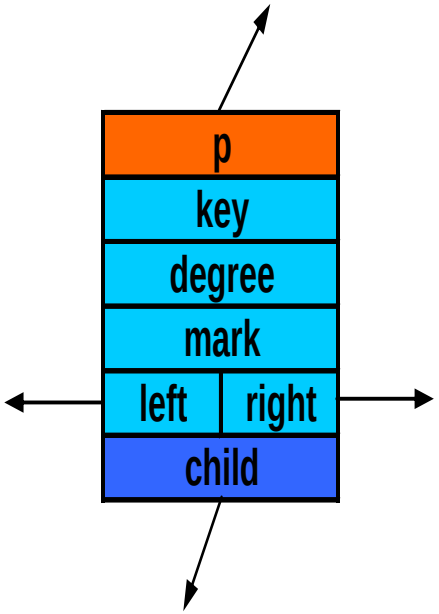
Fibonacci Heaps

- Fibonacci heaps differ from binomial-heaps, however, in that they have more more relaxed structure allowing for improved asymptotic time bounds work that maintains the structure is delayed until it is convenient to perform.
- Like a binomial heap, a fibonacci heap is a collection of heap-ordered trees however, trees are not constrained to be binomial trees.
- Trees within fibonacci heaps are rooted but unordered.

Structure of Fibonacci Heaps

Each node *x* contains:

- A pointer *p[x]* to its parent
- A pointer *child[x]* to one of its children
  - The children of *x* are linked together in a circular, doubly-linked list which is called the child-list.



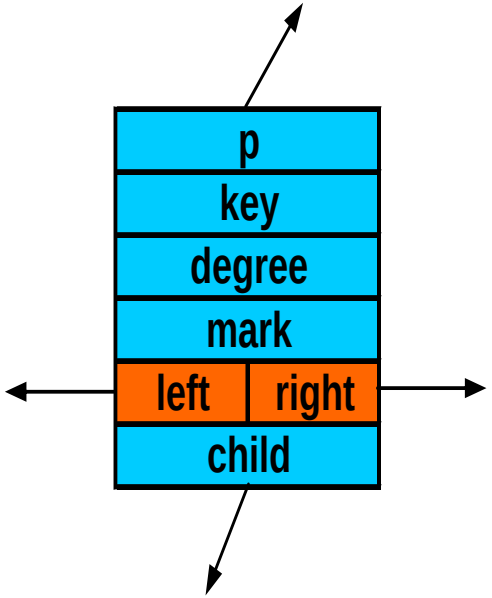
Structure of Fibonacci Heaps

- Each child *y* in a child list has pointers *left[y]* & *right[y]* that point to *y*'s left & right siblings respectively.
- If *y* is an only child, then

$$left[y] = right[y] = y.$$

Structure of Fibonacci Heaps

The roots of all trees are also linked together using their *left & right* pointers into a circular, doubly-linked list which is called the root list.



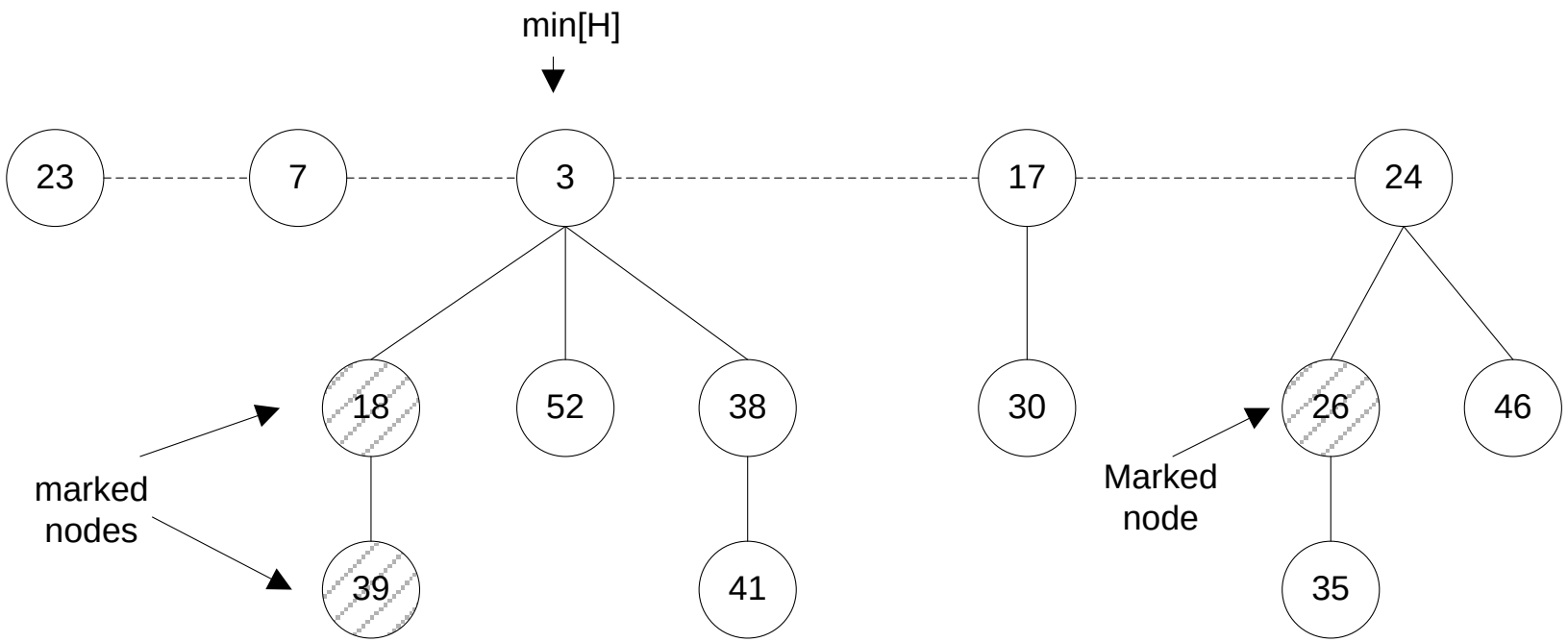
Structure of Fibonacci Heaps

- Circular, doubly-linked lists have two advantages for use in fib-heaps:
  - we can remove a node in *O(1)* time
  - given two such lists, we can concatenate them in *O(1)* time.

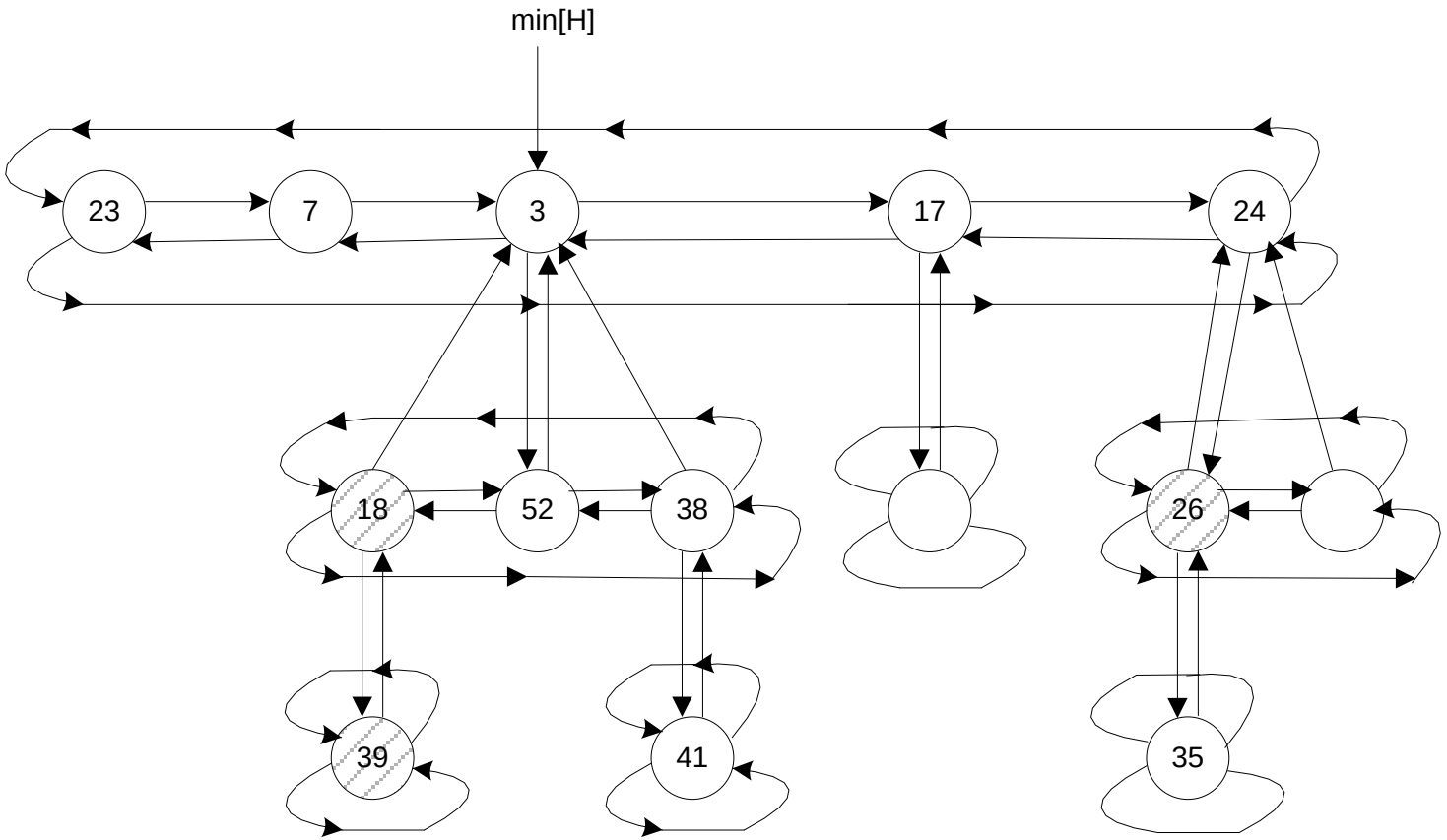
Structure of Fibonacci Heaps

- Two other fields in each node *x*
  - *degreee[x]:* the number of children in the child list of *x*
  - *mark[x]:* a boolean-valued field
    - indicates whether node *x* has lost a child since the last time *x* was made the child of another one
    - newly created nodes are unmarked
    - A node *x* becomes unmarked whenever it is made the child of another node

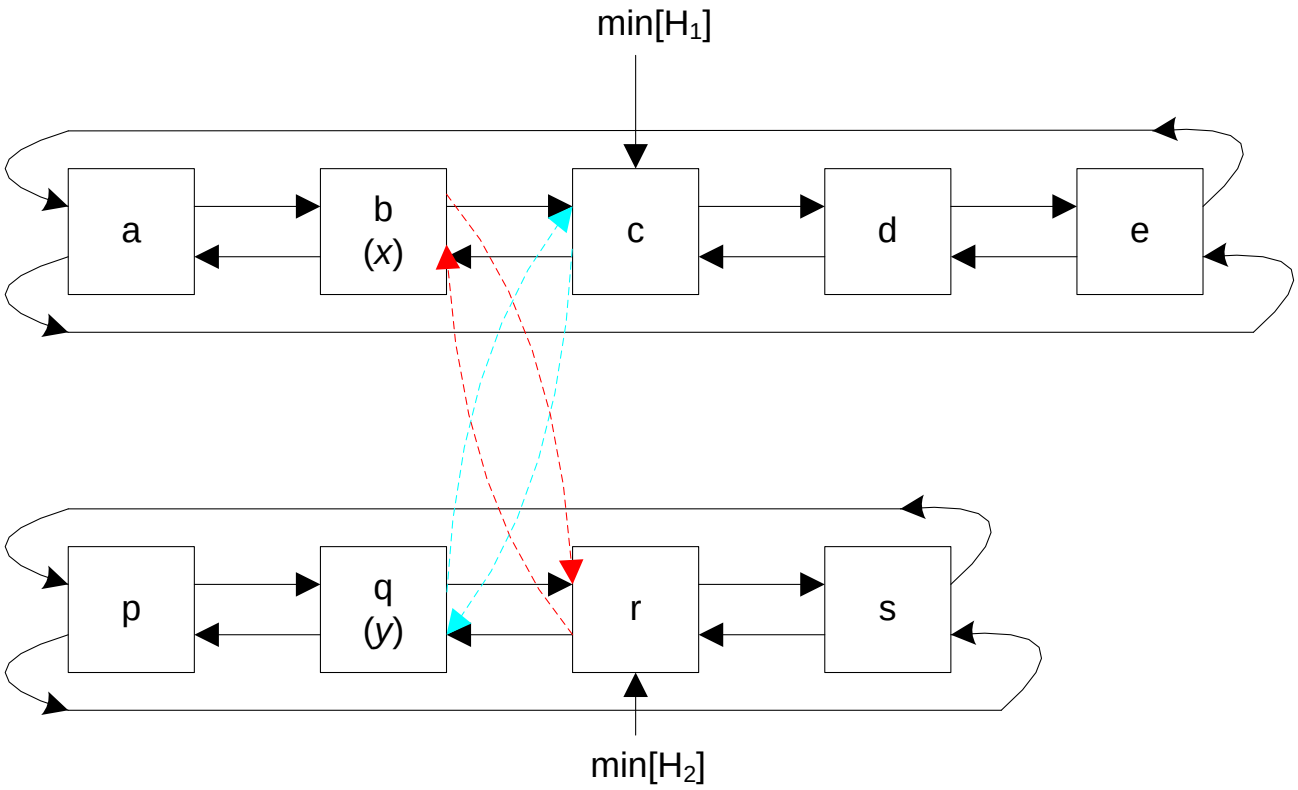
Structure of Fibonacci Heaps



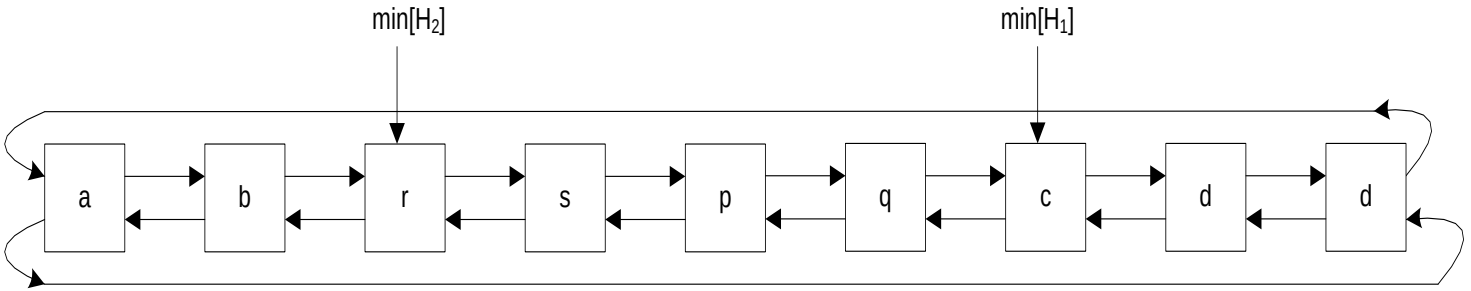
Structure of Fibonacci Heaps



Concetenation of Two Circular, Doubly – Linked Lists



Concetenation of Two Circular, Doubly – Linked Lists



Concetenation of Two Circular, Doubly – Linked Lists

CONCATENATE (H_1, H_2)

$x \leftarrow$ left[min[H_1]]

$y \leftarrow$ left[min[H_2]]

right[x] $\leftarrow$ min[H_2]

left[min[H_2]] $\leftarrow x$

right[y] $\leftarrow$ min[H_1]

left[min[H_1]] $\leftarrow y$

end

Running time is O(*1*)

Potential Function

- A given fibonacci heap H
  - $t(H)$: the number of trees in root list of H
  - $m(H)$: the number of marked nodes in H
- The potential of fibonacci heap H is:

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

- A fibonacci heap application begins with an empty heap:

the initial potential = 0

- The potential is non-negative at all subsequent times.

Maximum Degree

- We will assume that there is aknown upper bound $D(n)$ on the maximum degree of any node in an n node heap
- If only mergeable-heap operations are supported

$$D(n) = \lfloor \lg n \rfloor$$

- If decrease key & delete operations are supported

  $D(n) = O(\lg n)$

Mergeable Heap Operations

MAKE-HEAP, INSERT, MINIMUM, EXTRACT-MIN, UNION

If only these operations are to be supported, each fibonacci-heap is a collection of unordered binomial trees.

Mergeable Heap Operations

- An unordered binomial tree U_k
  - is like a binomial tree
  - defined recursively:
    - U_0 consists of a single node
    - U_k consists of two U_{k-1}'s for which the root of one is made into any child of the root of the other

Mergeable Heap Operations

- Lemma which gives properties of binomial trees holds for unordered binomial trees as well but with the following variation on property 4
- Property 4': For the unordered binomial tree U_k:
  - The root has degree k > the degree of any other node
  - The children of the root are the roots of subtrees

    $U_0, U_1, \ldots\ldots\ldots, U_{k-1}$ in some order

Mergeable Heap Operations

- The key idea in the mergeable heap operations on fibonacci heaps is to delay work as long as possible.
- Performance trade-off among implementations of the various operations:
  - If the number of trees is small we can quickly determine the new min node during EXTRACT-MIN
  - However we pay a price for ensuring that the number of trees is small

Mergeable Heap Operations

- However we pay a price for ensuring that the number of trees is small
- However it can take up to $\Omega(\lg n)$ time
  - to insert a node into a binomial heap
  - or to unite two binomial heaps
- We do not consolidate trees in a fibonacci heap when we insert a new node or unite two heaps
- We delay the consolidation for the EXTRACT-MIN operation when we really need to find the new minimum node.

Mergeable Heap Operations

Creating a new fibonacci heap:

MAKE-FIB-HEAP procedure

- allocates and returns the fibonacci heap object *H*
- Where *n[H]* = 0 and min[H] = NIL
- There are no trees in the heap

because *t(H)* = 0 and *m(H)* = 0 => *Φ(H)* = 0

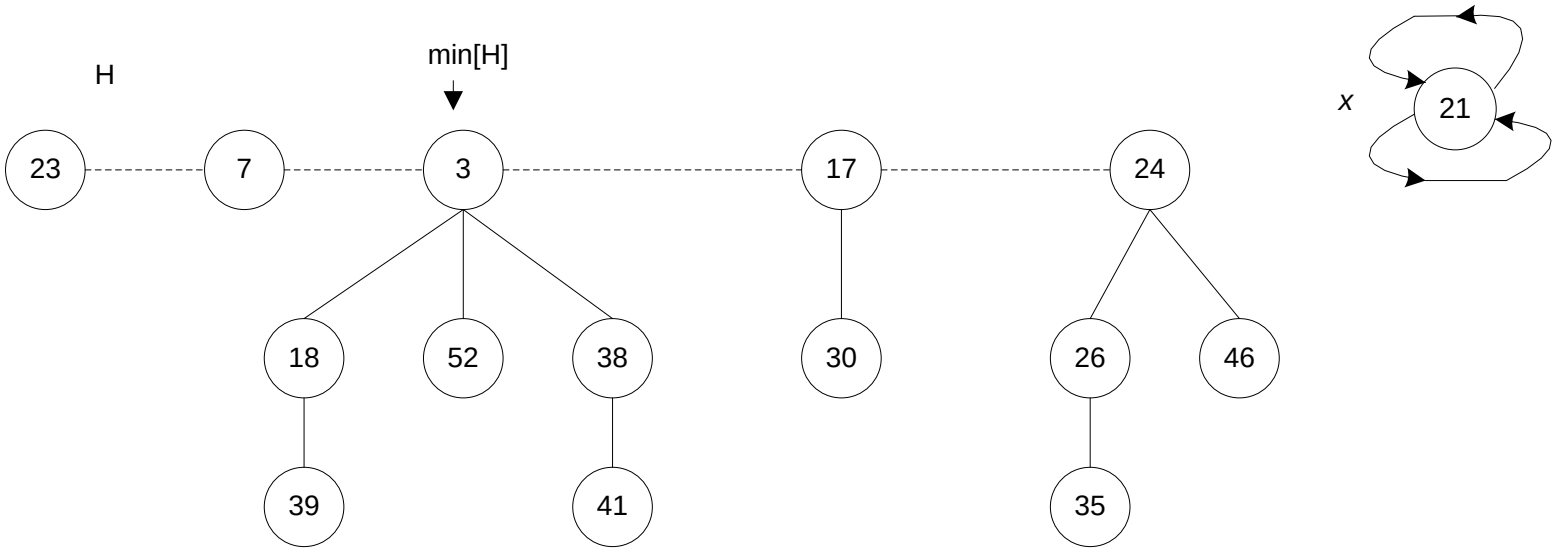
the amortized cost = *O(1)* = the actual cost

Mergeable Heap Operations

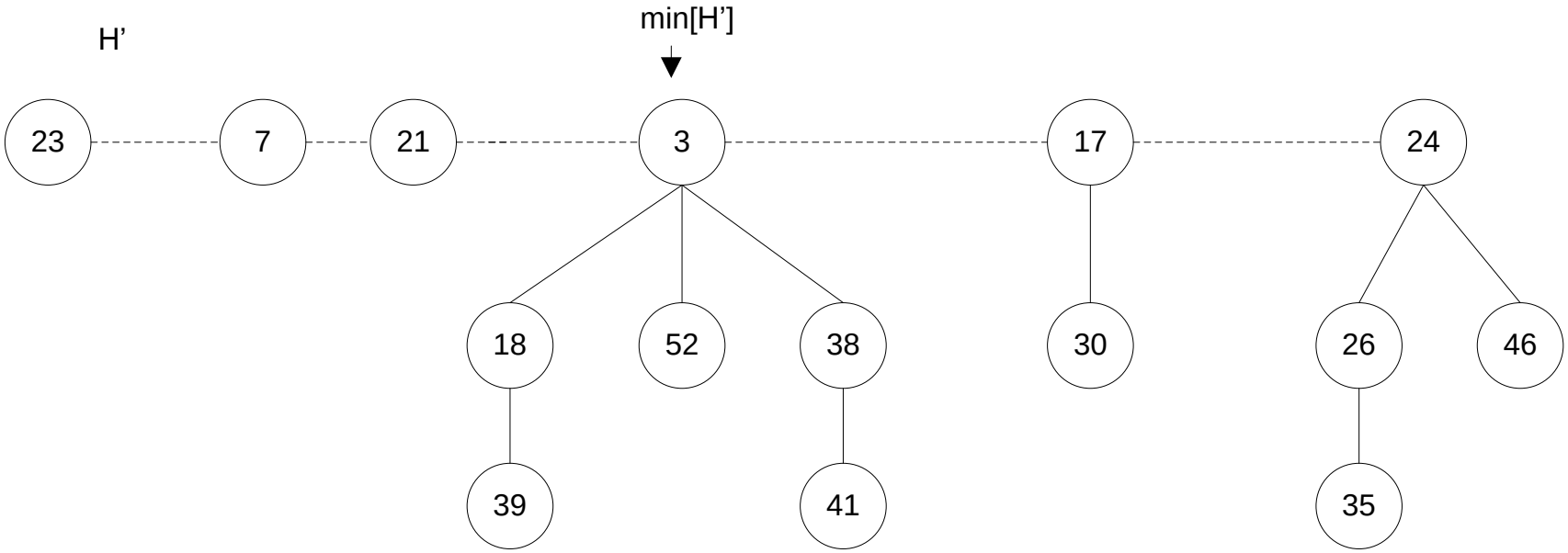
Inserting a node

```
FIB-HEAP-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 key[x] < key[min[H]] then
            min[H] ← x
    endif
    n[H] ← n[H] + 1
end
```

Mergeable Heap Operations



Mergeable Heap Operations



Mergeable Heap Operations

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

- Increase in potential:

$$\Phi(H') - \Phi(H) = [t(H) + 1 + 2m(H)] - [t(H) + 2m(H)]$$

$$= 1$$

The actual cost = $O(1)$

The amortized cost = $O(1) + 1 = O(1)$

Mergeable Heap Operations

Finding the minimum node:

Given by pointer min[*H*]

actual cost = *O(1)*

amortized cost = actual cost = *O(1)*

since the potential of *H* does not change

Uniting Two Fibonacci Heaps

FIB-HEAP-UNION(H_1, H_2)
    H = MAKE-FIB-HEAP()
    if key[min[H_1]] ≤ key[min[H_2]] then
        min[H] ← min[H_1]
    else
        min[H] ← min[H_2]
    endif
    concatenate the root lists of H_1 and H_2
    n[H] ← n[H_1] + n[H_2]
    Free the objects H_1 and H_2
    return H
end

Uniting Two Fibonacci Heaps

- No consolidation of trees
- Actual cost = $O(1)$
- Change in potential

$$\Phi(H) - (\Phi(H_1) + \Phi(H_2)) =$$
$$= (t(H) + 2m(H)) - ((t(H_1) + 2m(H_1)) + (t(H_2) + 2m(H_2)))$$
$$= 0 \text{ since } t(H) = t(H_1) + t(H_2)$$
$$m(H) = m(H_1) + m(H_2)$$

Therefore amortized cost = actual cost = $O(1)$

Extracting the Minimum Node

The most complicated operation the delayed work of consolidating the trees in the root list occurs

```
FIB-HEAP-EXTRACT-MIN(H)
    z = min[H]
    for each child x of z
            add x to the root list of H
            p[x] ← NIL
    endfor
    remove z from the root list of H
    min[H] ← right[z]
    CONSOLIDATE(H)
end
```

Extracting the Minimum Node

- Repeatedly execute the following steps until every root in the root list has a distinct degree value

(1) Find two roots x and y in the root list with the same degree

where key[x] ≤ key[y]

(2) Link y to x : Remove y from the root list and make y a child of x

This operation is performed by procedure FIB-HEAP-LINK

Procedure CONSOLIDATE uses an auxiliary pointer array A[0.....*D(n)*]

A[i] = y : y is currently a root with degree[y] = i

Extracting the Minimum Node

```
CONSOLIDATE(H)
    for i← 0 to D(n) do
            A[i] ← N IL
    endfor
    for each node w in the root list of H do
            x ← w
            d ← degree[x]
            while A[d] ≠ NIL do
                        y ← A[d]
                        if key[x] > key[y] then
                                    exchange x ↔ y
                        endif
                        FIB-HEAP-LINK(H,y,x)
                        A[d] ← NIL
                        d ← d + 1
            endwhile
            A[d] ← x
    endfor
```

```
min[H] ← +∞
for i ← 0 to D(n[H]) do
if A[i] ≠ NIL then
            add A[i] to the root list of H
            if key[A[i]] < key[min[H]] then
                        min[H] ← A[i]
            endif
            endif
endfor
end
```

Extracting the Minimum Node

FIB-HEAP-LINK(H,*y*,*x*)

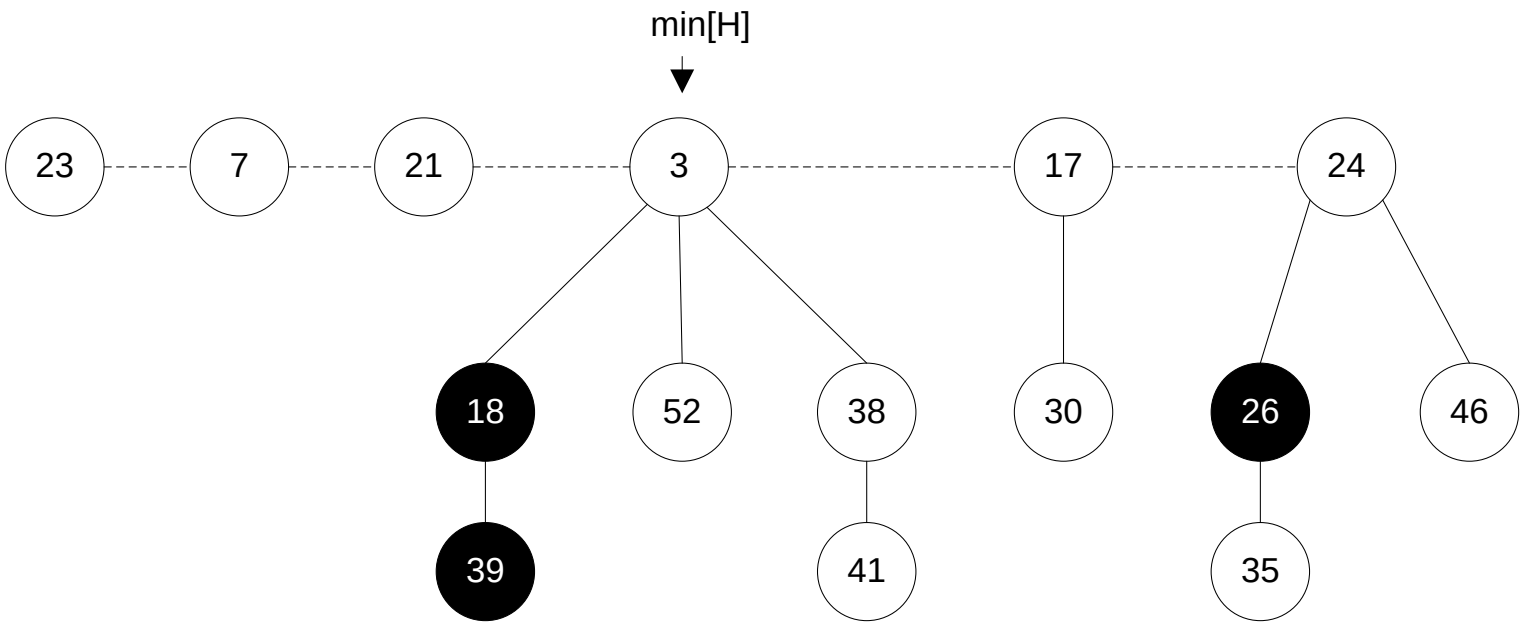
remove *y* from the root list of H

make *y* a child of *x*, incrementing degree[*x*]

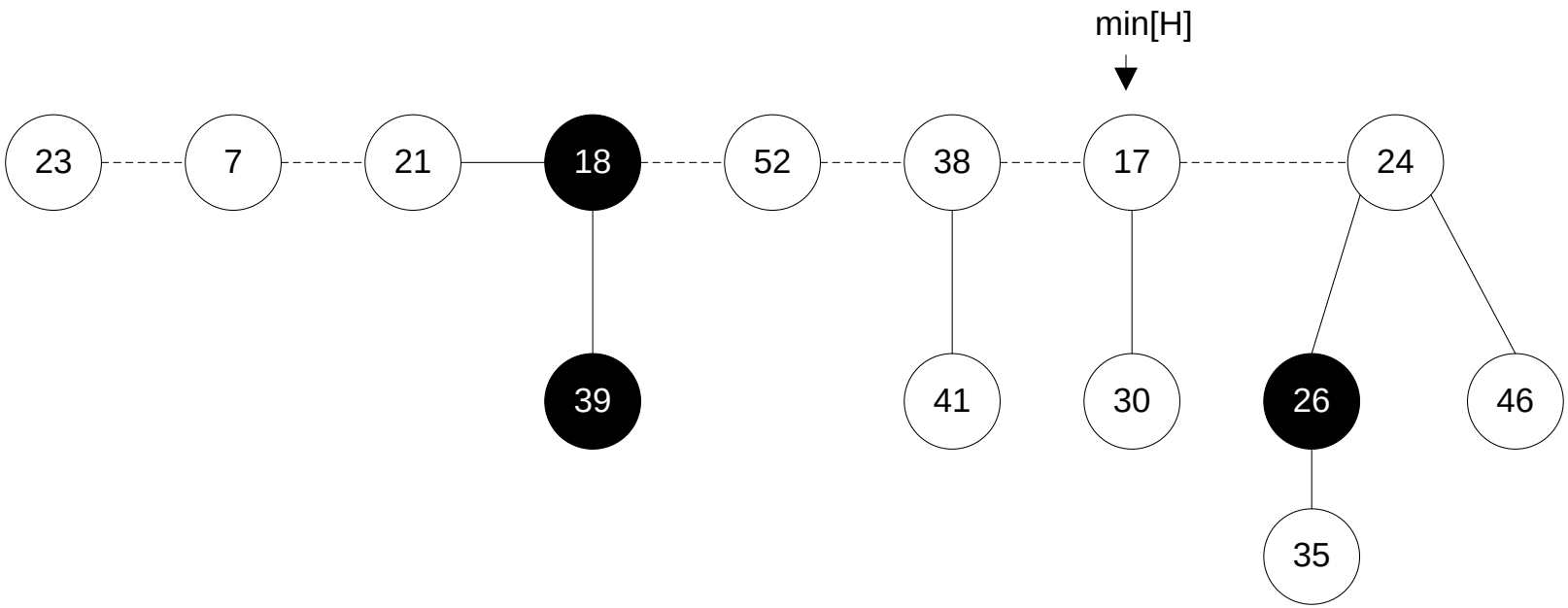
mark[*y*] ← FALSE

end

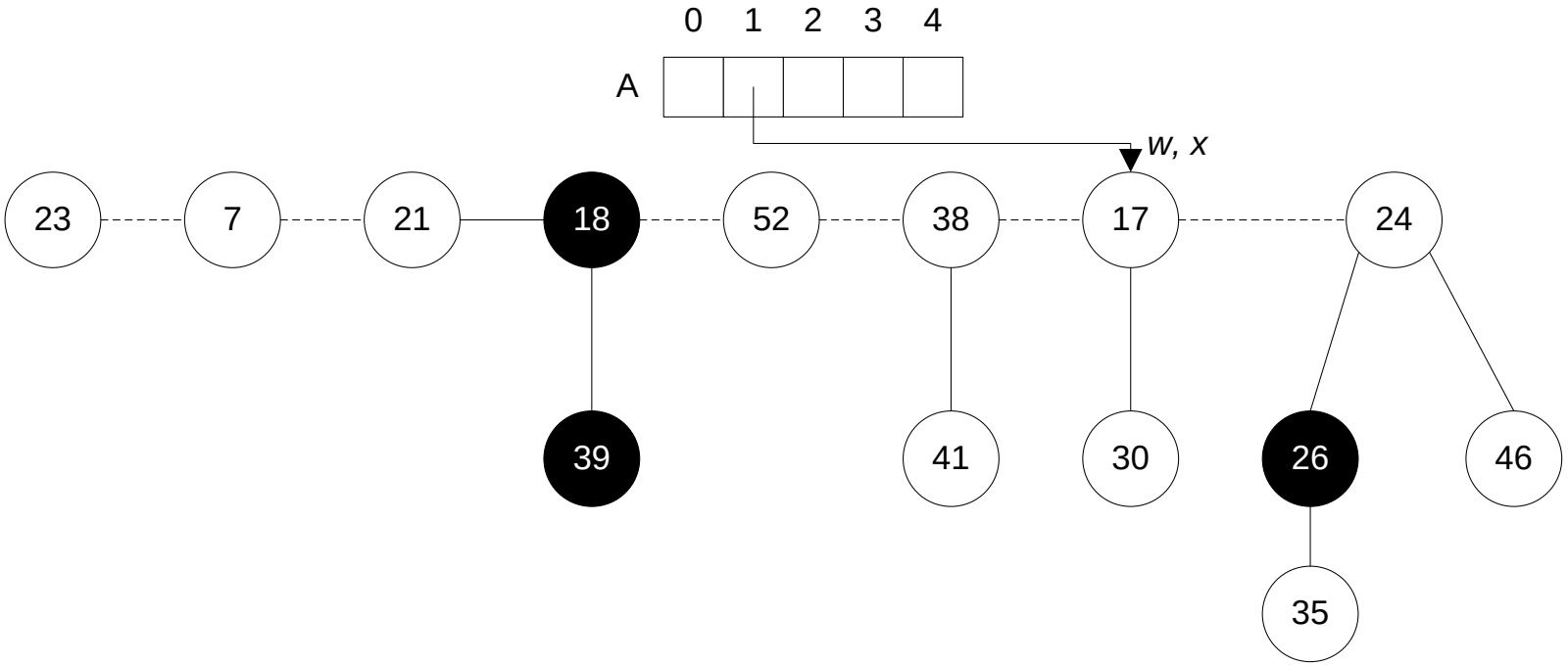
Extracting the Minimum Node



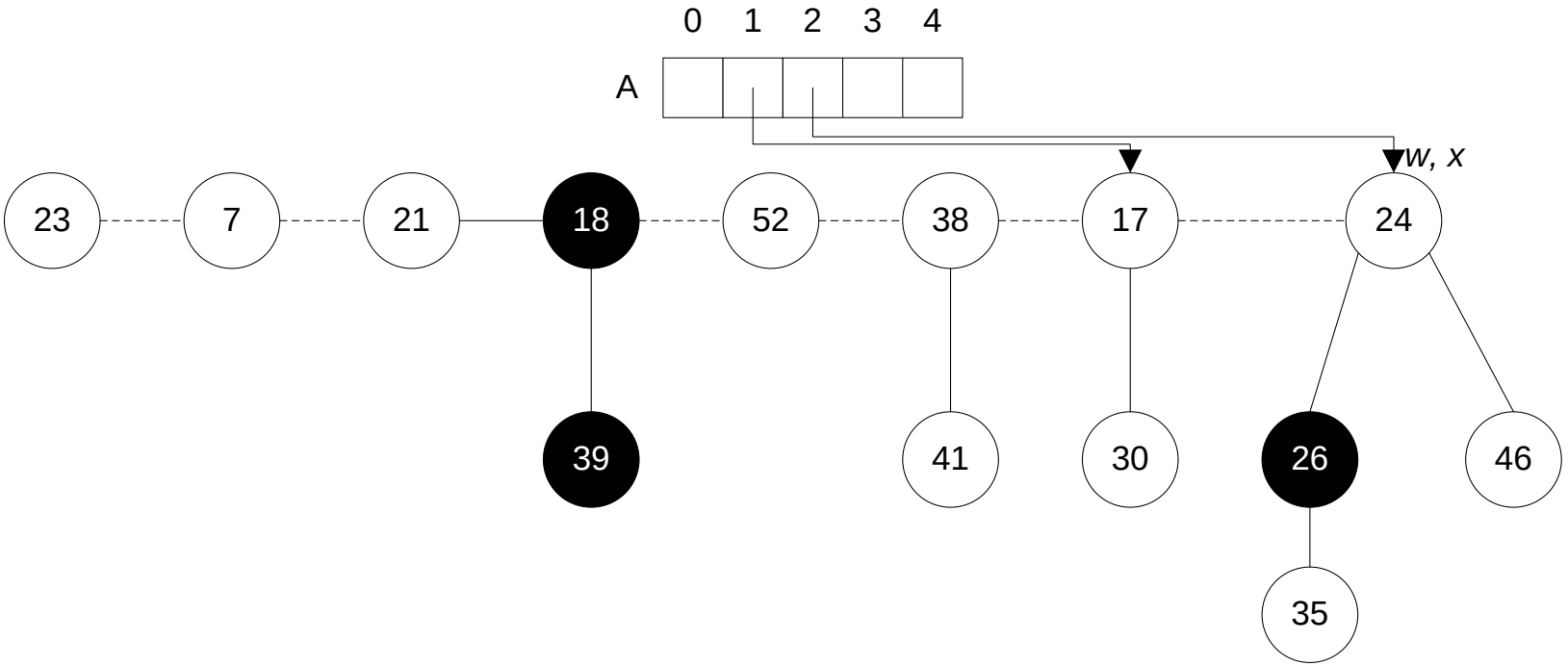
Extracting the Minimum Node



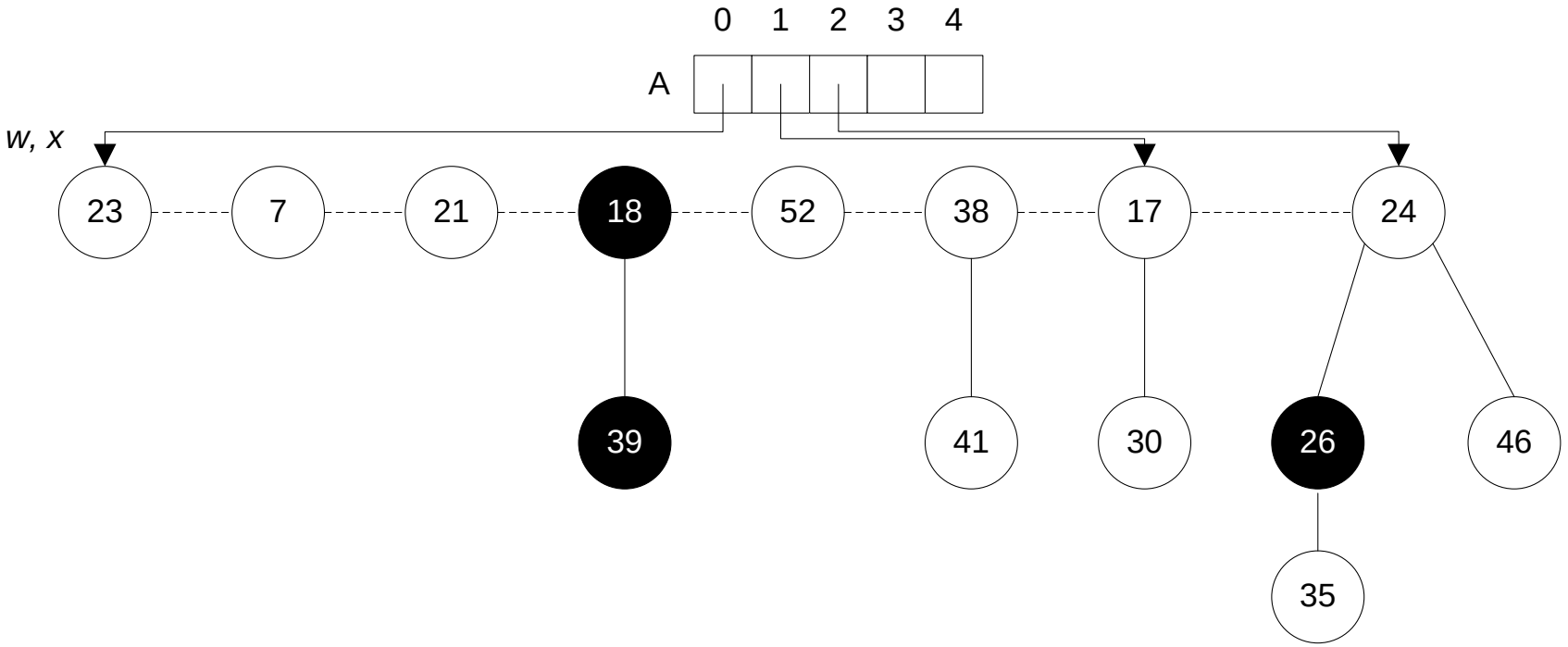
Extracting the Minimum Node



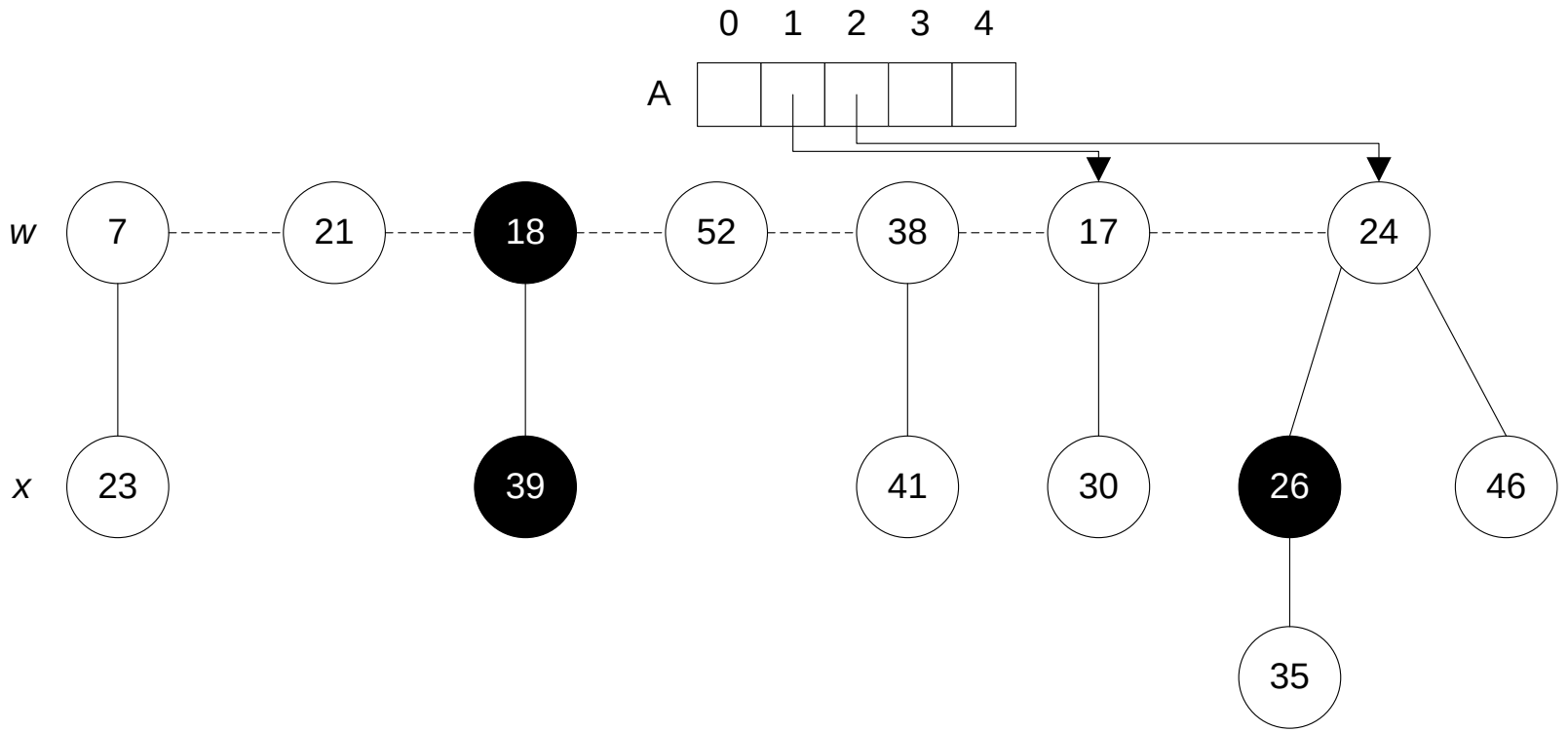
Extracting the Minimum Node



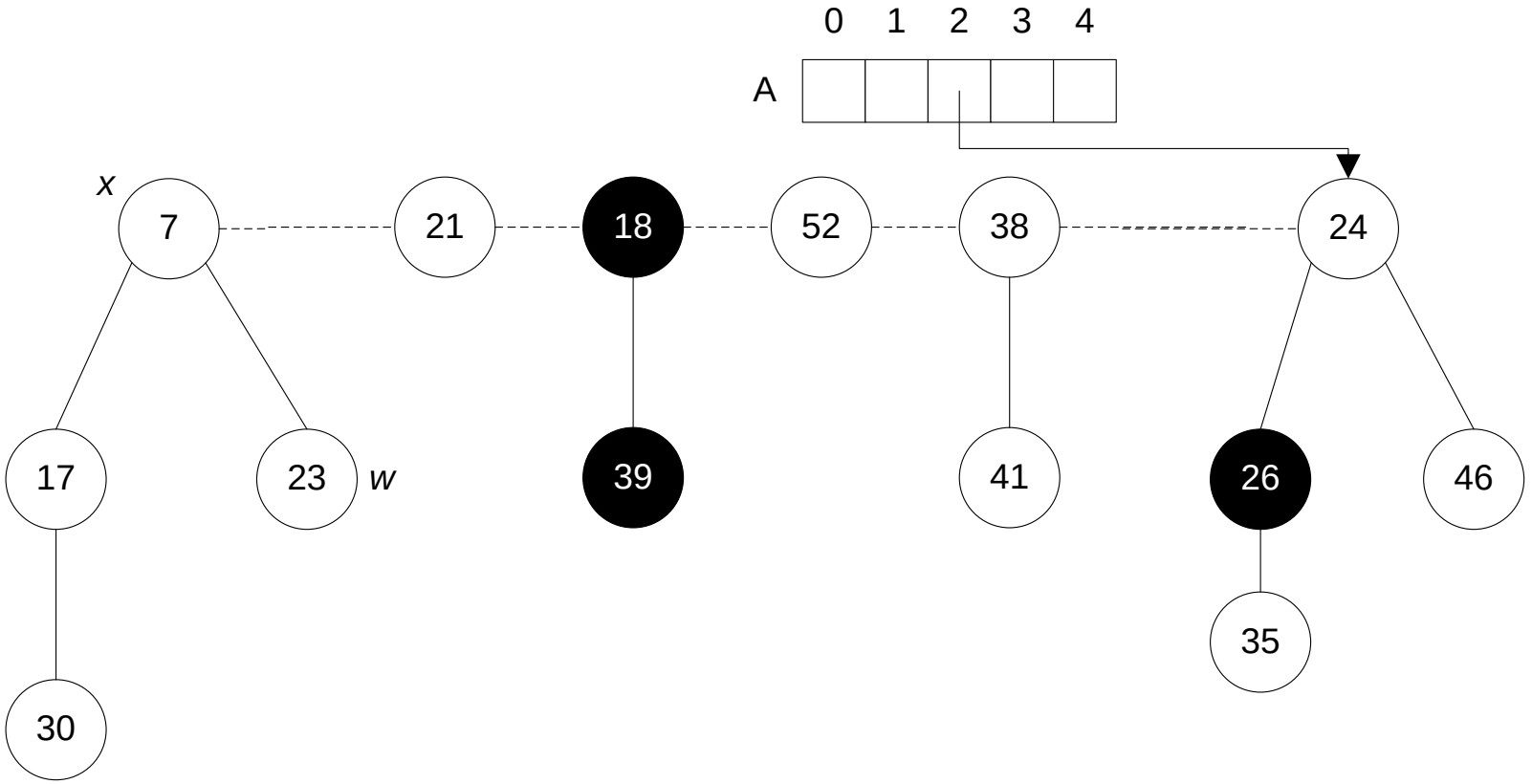
Extracting the Minimum Node



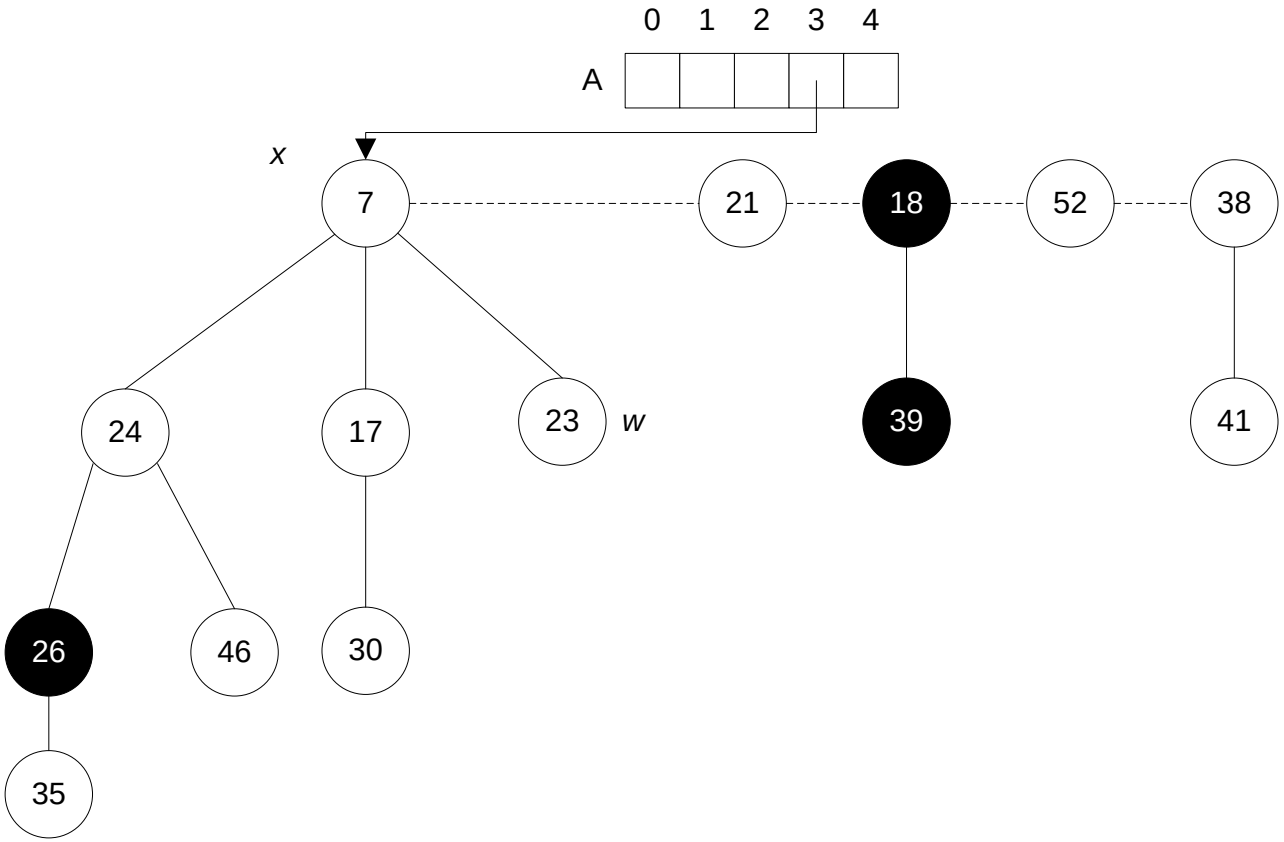
Extracting the Minimum Node



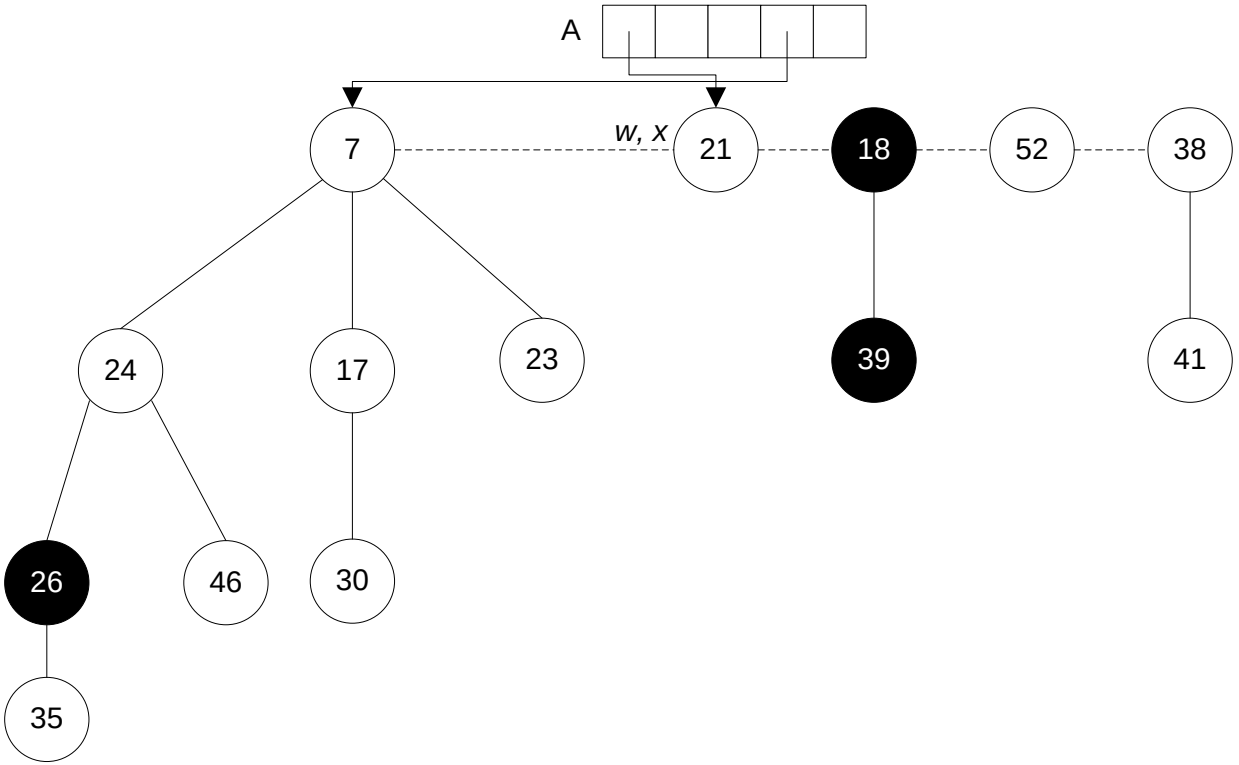
Extracting the Minimum Node



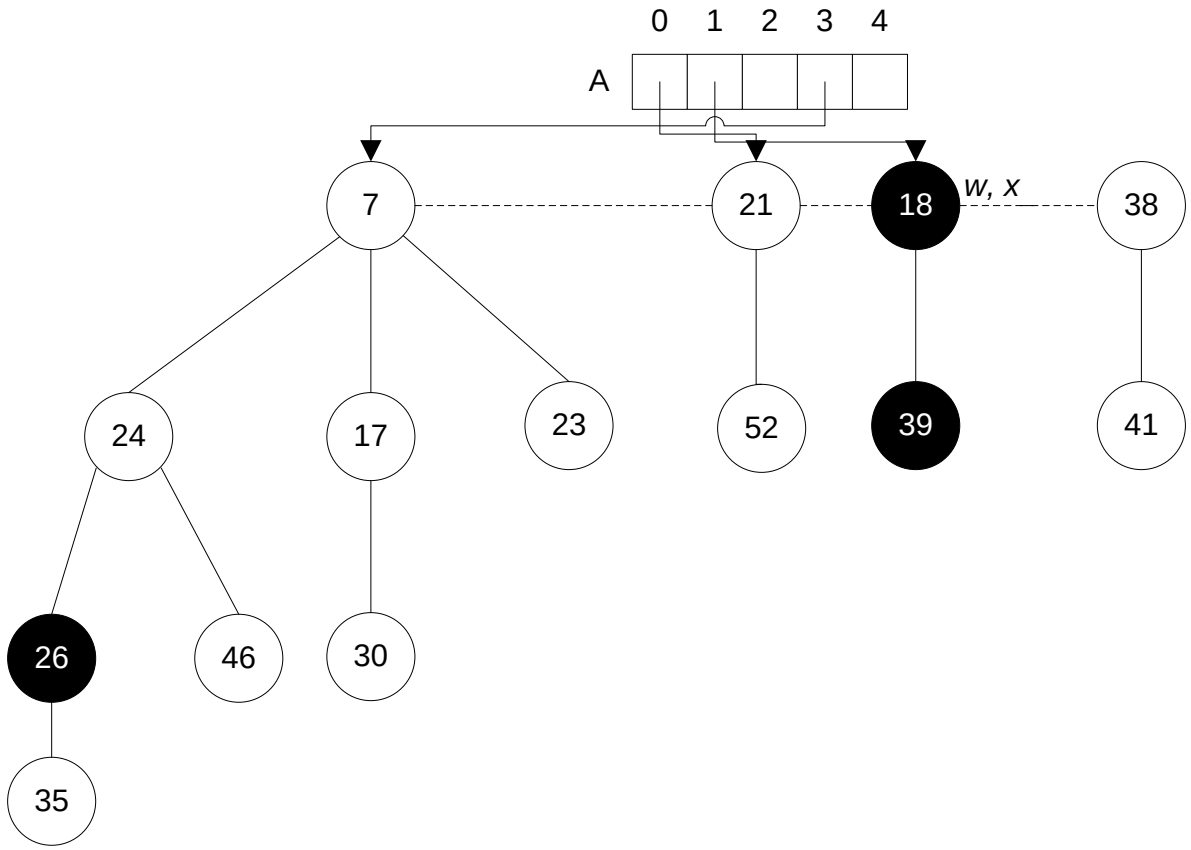
Extracting the Minimum Node



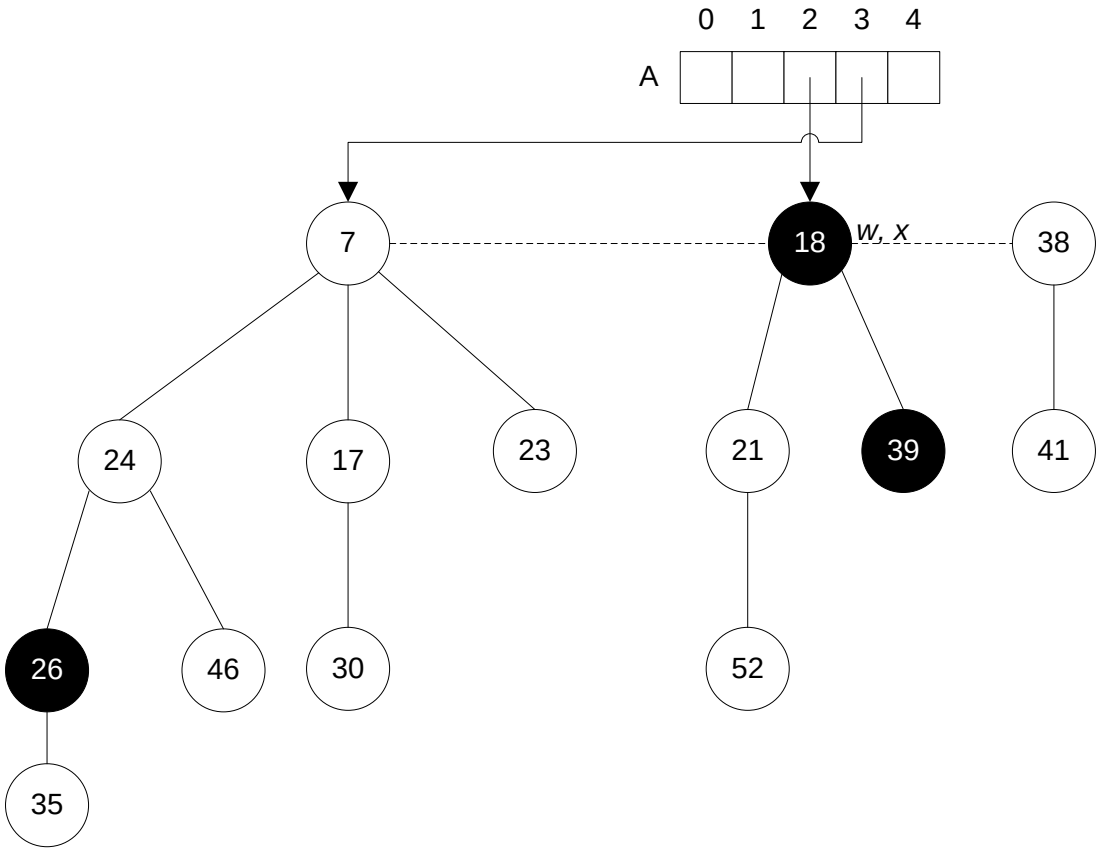
Extracting the Minimum Node



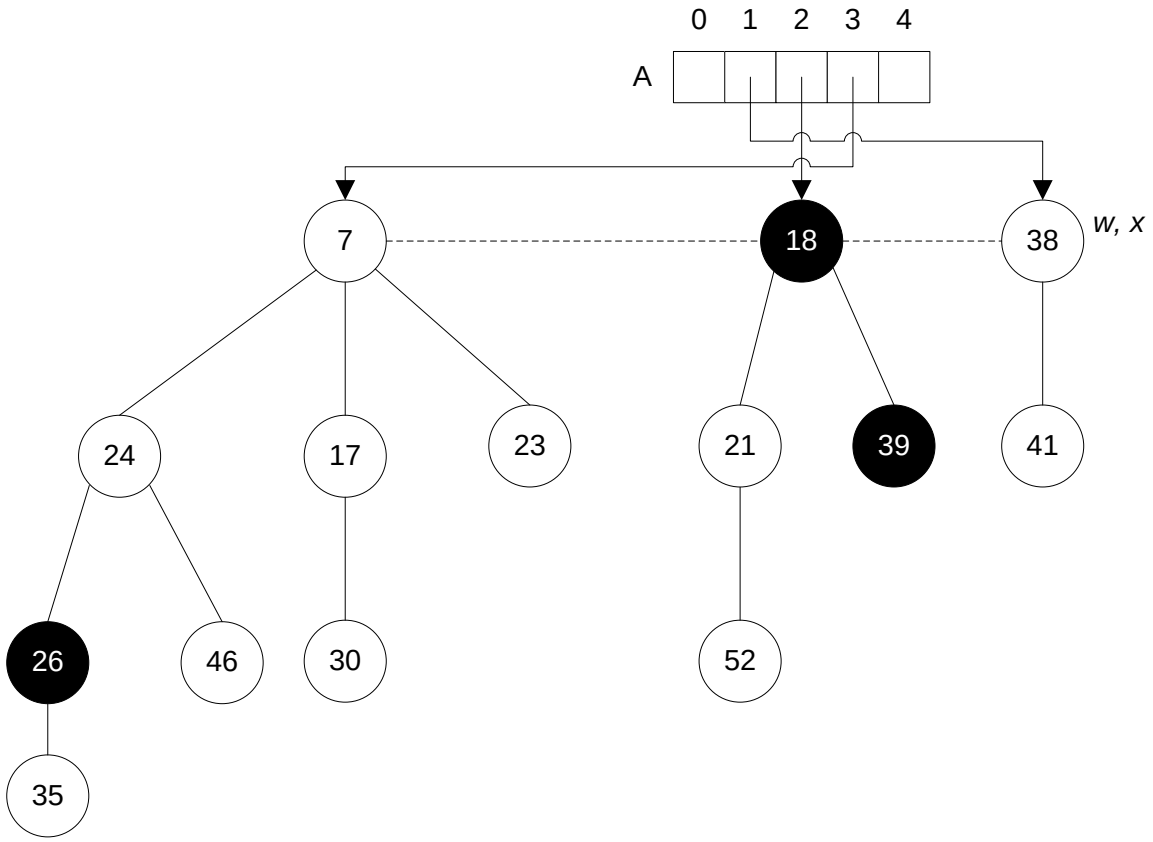
Extracting the Minimum Node



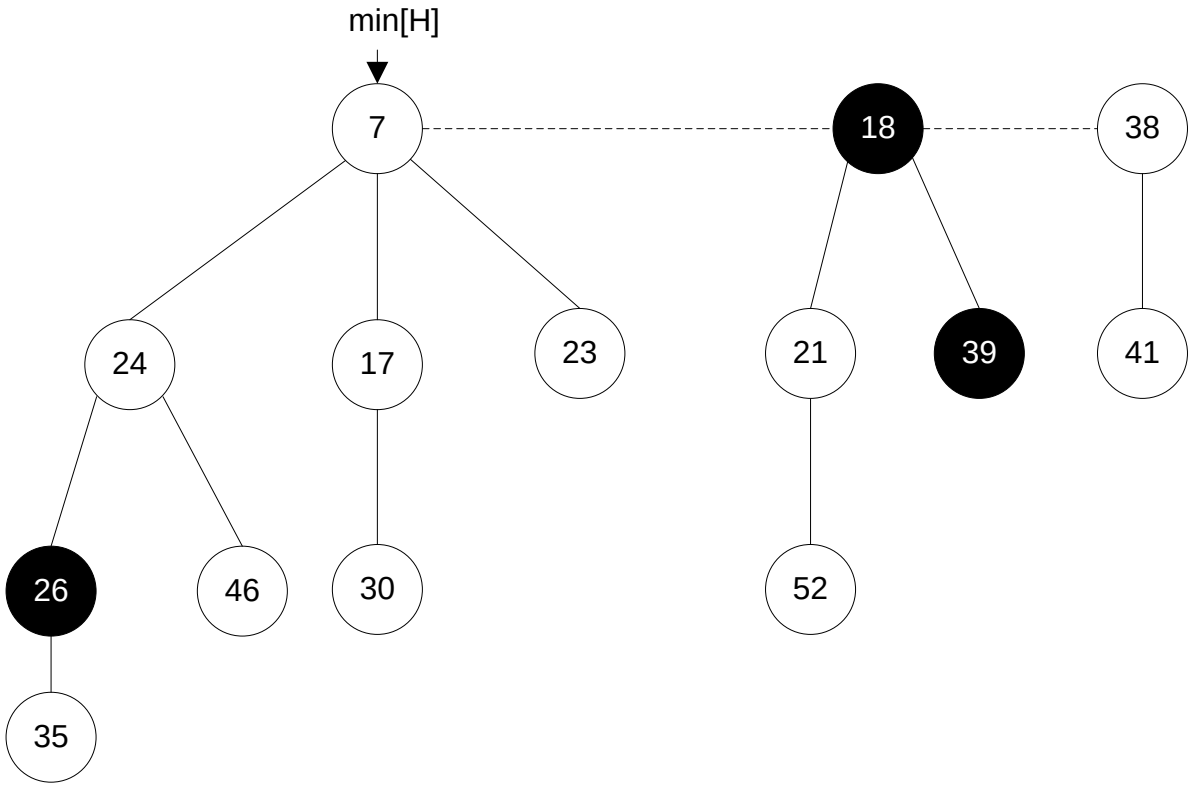
Extracting the Minimum Node



Extracting the Minimum Node



Extracting the Minimum Node



Analysis of the FIB-HEAP-EXTRACT-MIN Procedure

- If all trees in the fib-heap are unordered binomial trees before the execution of the EXTRACT-MIN operation then they are all unordered binomial trees afterward.
- There are two ways in which trees are changed:

  (1) each child of the extracted root node becomes a child, each new tree is itself an unordered binomial tree.

  (2) trees are linked by FIB-HEAP-LINK procedure only if they have the same degree hence U_k is linked to U_k to form a U_{k+1}

Complexity Analysis of the FIB-HEAP-EXTRACT-MIN Procedure

Actual Cost

1-st for loop: Contributes O(D(n))

3-rd for loop: Contributes O(D(n))

2-nd for loop:

Size of the root-list upon calling CONSOLIDATE is at most:

$$D(n) + t(H) - 1$$

D(n): upper bound on the number of children of the extracted node

t(H) – 1: original *t(H)* root list nodes – the extracted node

Complexity Analysis of the FIB-HEAP-EXTRACT-MIN Procedure

Each iteration of the inner while-loop links one root to another thus reducing the size of the root list by 1

Therefore the total amount work performed in the 2-nd for loop is at most proportional to $D(n) + t(H)$

Thus, the total actual cost is $O(D(n) + t(H))$

Complexity Analysis of the FIB-HEAP-EXTRACT-MIN Procedure

Amortized Cost

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

Potential after: at most $(D(n) + 1)+2m(H)$ since at most $D(n)+1$ roots remain & no nodes marked

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

$[(D(n) + 1) + 2m(H)] –$

$[t(H) + 2m(H)]$

$= O(D(n)) + O(t(H)) – D(n) – t(H)$

$= \boldsymbol{O(D(n))}$

Complexity Analysis of the FIB-HEAP-EXTRACT-MIN Procedure

The cost of performing each link is paid for by the reduction in potential due to the link reducing the number of roots by one

EXTRACT-MIN Procedure for Fibonacci Heaps

```
FIB-HEAP-EXTRACT-MIN (H)
z ← min[ H ]
if  z ≠ NIL then
        for each child x of z do
                add x to the root list of H
                p [ x ] ← NIL
        endfor
        remove z from the root list of H
        if right [ z ] = z then
                min [ H ] ← NIL
        else
                min [ H ] ← right [ z ]
                CONSOLIDATE (H)
        endif
        n [ H ] ← n [ H ] – 1
endif
return z
end
```

EXTRACT-MIN Procedure for Fibonacci Heaps

FIB-HEAP-LINK (H, *y*, *x*)

remove *y* from the root list of H

make *y* a child of *x*, incrementing degree [x]

mark [*y*] ← FALSE

end

EXTRACT-MIN Procedure for Fibonacci Heaps

```
CONSOLIDATE ( H )
    for i ← 0 to D ( n ( H ) )
            A[ i ] ← NIL
    endfor
    for each node w in the root list of H do
            x ← w
            d ← degree [ x ]
            while A [ d ] ≠ NIL do
                    y ← A [ d ]
                    if key [ x ] > key [ y ] then
                                exchange x ↔ y
                    endif
                    FIB-HEAP-LINK ( H , y, x )
                    A [ d ] ← NIL
                    d ← d + 1
            endwhile
            A [ d ] ← x
    endfor
```

```
min [ H ] ← NIL
   for i ← 0 to D ( n [ H ] ) do
      if A [ i ] ≠ NIL then
         Add A [ i ] to the root list of H
         if min [ H ] = NIL or key [ A [ i ] ] <
                              key [ min [ H ] ] then
            min [ H ] ← A [ i ]
         endif
      endif
   endfor
end
```

Bounding the Maximum Degree

For each node x within a fibonacci heap, define

size(x): the number of nodes, including itself, in the subtree rooted at x

NOTE: x need not to be in the root list, it can be any node at all.

We shall show that size(x) is exponential in degree[x]

Bounding the Maximum Degree

Lemma 1: Let x be a node with degree[x]=k

Let $y_1, y_2, ..., y_k$ denote the children of x in the order in which they are linked to x, from earliest to the latest, then

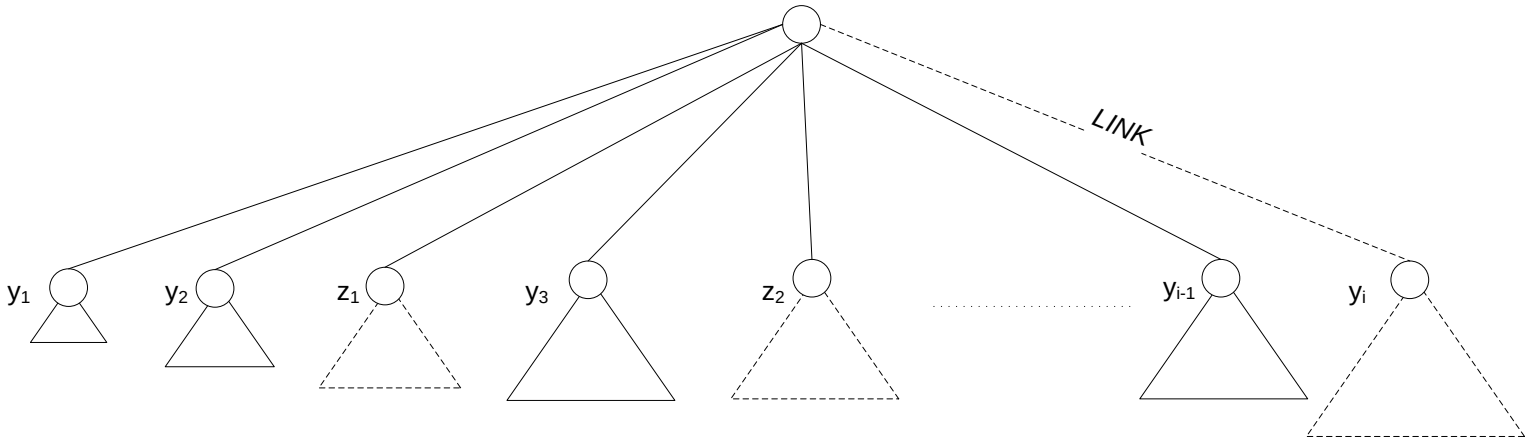
$$\text{degree}[y_1] \geq 0 \text{ and } \text{degree}[y_i] \geq i-2$$

$$\text{for } i = 2,3,...,k$$

Bounding the Maximum Degree

Proof: degree[y_1] ≥ 0 =>
obvious

For $i \geq 2$:



Bounding the Maximum Degree

- When y_i is linked to x: at least $y_1, y_2, \ldots, y_{i-1}$ were all children of x so we must have had degree[x] $\geq i - 1$
- NOTE: z node(s) denotes the node(s)

  that were children of x just before the link of y_i that are lost after the link of y_i
- When y_i is linked to x:

  $$\text{degree}[y_i] = \text{degree}[x] \geq i - 1$$

  since then, node y_i has lost at most one child, we conclude that degree[y_i] $\geq$ i-2

Bounding the Maximum Degree

As in Lemma-1, let $y_1, y_2, ..., y_k$ denote the children of node *x* in the order in which they were linked to *x*

$$size(x) \geq S_k \geq 1 + 1 + \sum_{i=2}^{k} S_{i-2} = 2 + \sum_{i=2}^{k} S_{i-2}$$

for x itself

S_1 for y_1

for $y_2, ..., y_k$ due to Lemma-1

Bounding the Maximum Degree

Corollary: Max. degree *D(n)* in an n node fib-heap is *O(lg n)*

Proof: Let *x* be any node with degree[*x*] = k in an n-node fib-heap by Lemma-3 we have

$$n \geq \text{size}(x) \geq \Phi^k$$

taking base-Φ log => $k \leq \log_\Phi n$

therefore *D(n) = O(lg n)*

Decreasing a Key

```
FIB-HEAP-DECREASE-KEY(H, x, k)
    key[x] ← k
    y ← p[x]
    if y ≠ NIL and key[x] < key[y] then
            CUT(H, x, y)
            CASCADING-CUT(H,y)
    endif   /* else no structural change */
    if key[x] < key[min[H]] then
            min[H] ← x
    endif
end
```

Decreasing a Key

CUT(H, x, y)

remove x from the child list of y, decrementing degree y

add x to the root list of H

p[x] ← NIL

mark[x] ← FALSE

end

Decreasing a Key

```
CASCADING-CUT(H, y)
        z ← p[y]
        if z ≠ NIL then
                if mark[y] = FALSE then
                        mark[y] = TRUE
                else
                        CUT(H, y, z)
                        CASCADING-CUT(H, z)
                endif
        endif
end
```

Amortized Cost of FIB-HEAP-DECREASE-KEY Procedure

Actual Cost

O(1) time + the time required to perform the cascading cuts, suppose that CASCADING-CUT is recursively called *c* times each call takes *O(1)* time exclusive of recursive calls therefore,

the actual cost = $O(1) + O(c) = \boldsymbol{O(c)}$

Amortized Cost of FIB-HEAP-DECREASE-KEY Procedure

Amortized Cost

Let H denote the fib-heap prior to the DECREASE-KEY operation.

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

Amortized Cost of FIB-HEAP-DECREASE-KEY Procedure

Hence, after the DECREASE-KEY operation

$$number\,of\,trees = t(H) + 1 + (c-1) = t(H) + C$$

tree rooted at x

trees produced by cascading cuts

$$number\,of\,marked\,nodes = m(H) - (c-1) + 1 = m(H) - c + 2$$

unmarked during the first c-1 CASCADING-CUTS

marked during the last CASCADING-CUT

Amortized Cost of FIB-HEAP-DECREASE-KEY Procedure

Potential Difference

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

$= 4 - c$

Amortized Cost $= O(c) + 4 - c = \mathbf{O(1)}$

Deleting a Node

FIB-HEAP-DELETE(H, x)

FIB-HEAP-DECREASE-KEY(H, x, -∞) => *O(1)*

FIB-HEAP-EXTRACT-MIN(H) => *O(D(n))*

end

Amortized Cost = *O(1) + O(D(n)) = O(D(n))*

Bounding the Maximum Degree

- Why do we apply CASCADING-CUT during DECREASE-KEY operation?
- To maintain the size of any tree/subtree exponential in the degree of its root node.

  e.g.: to prevent cases where

  $$size[x] = degree[x] + 1$$