Algorithms II, CS 502

Fibonacci Heaps

Ugur Dogrusoz

Computer Eng Dept, Bilkent Univ

Amortized analysis

- Average cost of an operation is small when averaged over a sequence of operations even though a single operation might be expensive
- Methods
  - Aggregate
  - Accounting (associated with each object)
  - Potential (associated with whole data structure)
- Example: ArrayList in Java

Potential method

- Represents prepaid work as potential energy or just potential that can be released to pay for the future operations

$$C_i : \text{actual cost of } i^{\text{th}} \text{ operation}$$

$$D_i : \text{data structure after } i^{\text{th}} \text{ operation}$$

$$\phi(D_i) : \text{potential associated with } D_i$$

$$\hat{C}_i : \text{amortized cost of } i^{\text{th}} \text{ operation w.r.t. } \phi$$

Potential method

$$\hat{C}_i = C_i + \phi(D_i) - \phi(D_{i-1})$$

$$\sum_{i=1}^{n} \hat{C}_i = \sum_{i=1}^{n} [C_i + \phi(D_i) - \phi(D_{i-1})] =$$

$$\sum_{i=1}^{n} C_i + \phi(D_n) - \phi(D_0)$$

- If we ensure that $\phi(D_i) \geq \phi(D_0), 0 \leq i \leq n$ then total amortized cost is an upper bound on actual cost

Fibonacci heaps

- Mergeable heaps support:
  - MAKE-HEAP(): create and return a new empty heap
  - INSERT(H,x): insert element x into heap H
  - MINIMUM (H): return a pointer to element with minimum key in H
  - EXTRACT-MIN(H): delete and return a pointer to element with minimum key in H
  - UNION(H_1,H_2): create and return a new heap containing all elements of H_1 and H_2

Fibonacci heaps

- Additionally Fibanocci heaps support:
  - DECREASE-KEY(H,x,k): assign key k (no greater than current key value) to element x in H
  - DELETE(H,x): deletes element x from heap H

Fibonacci heaps

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	$\Theta(\lg n)$	$O(\lg n)$

Fibonacci heaps

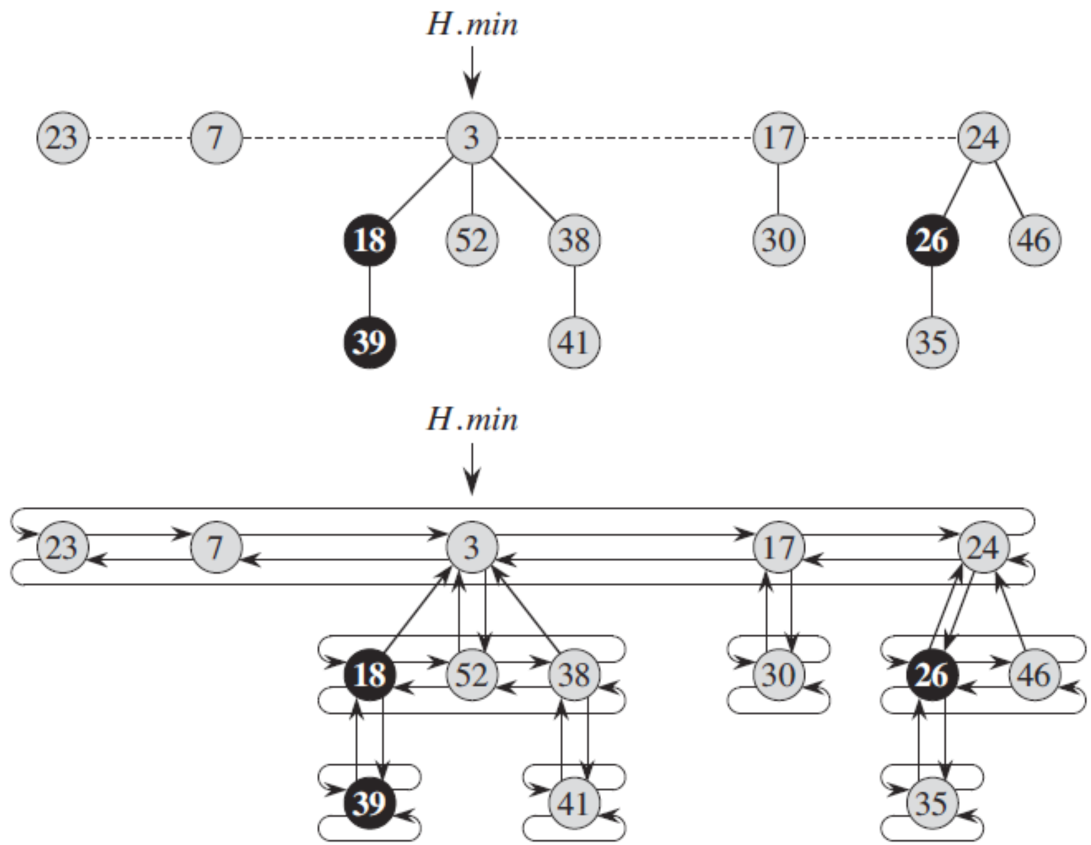
- Each node **x** of Fibonacci heap **H** contains
  - **x.key**: its key
  - **x.p**: its parent
  - **x.child**: any one of its children (forms a circular list)
  - **x.degree**: number of children
  - **x.mark**: whether **x** has lost a child since the last time **x** was made the child of another node
  - **y.left/right**: each child maintains left and right siblings (forms a circular list)

Fibonacci heaps

- For Fibonacci heap **`H`**
  - **`H.min`**: the root of a tree containing the minimum key
  - **`H.n`**: number of nodes in **`H`**

Fibonacci heaps

- a collection of rooted trees that are min-heap ordered



Potential function

- For Fibonacci heap **H**
  - $t(H)$: number of trees in the root list of **H**
  - $m(H)$: number of marked nodes in **H**
  - $\Phi(H)=t(H)+2m(H)$: potential of **H**
  - $\Phi(H_0)\leq\Phi(H_i)$ holds

Maximum degree

- Amortized analysis assumes we know
  - an upper bound on the maximum degree $D(n)$ of any node in a Fibonacci heap with n nodes
  - When only mergeable heap operations are supported $D(n) \le \lg n$
  - Need to show $D(n) = O(\lg n)$ with DECREASE-KEY and DELETE as well

Mergeable heap operations

- Delay work as long as possible
  - Do not consolidate trees in root list on operations like INSERT or UNION
  - Delay until the next EXTRACT-MIN when we really need to find a new minimum
  - After consolidation, each node in the root list has a degree that is unique, ensuring a root list of size at most **D(n)+1=O(lg n)**
- Creating a new empty Fibonacci heap is straightforward in O(1) time, returning **H** with
  - **H.n=0** and **H.min=NIL**

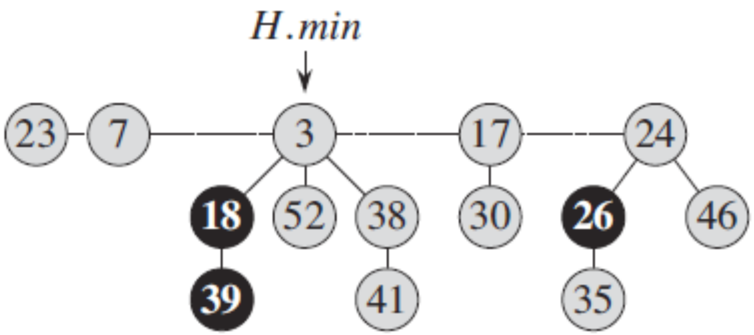
Inserting a node

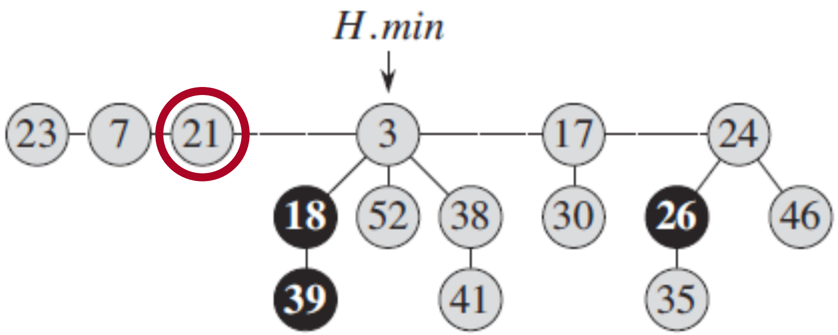
- The node with new key becomes its own min-heap-ordered tree, and added to root list

```
FIB-HEAP-INSERT(H, x)
x.degree = 0
x.p = NIL
x.child = NIL
x.mark = FALSE
if H.min == NIL
    create a root list for H containing just x
    H.min = x
else insert x into H's root list
    if x.key < H.min.key
        H.min = x
H.n = H.n + 1
```

Inserting a node

- The node with new key becomes its own min-heap-ordered tree, and added to root list





Inserting a node

- For a heap **H** before insertion and heap **H′** after insertion, we have
  - $t(H') = t(H) + 1$
  - $m(H') = m(H)$
  - Increase in potential
    - $[t(H)+1+2m(H)] - [t(H)+2m(H)] = 1$
  - Since actual cost is O(1), the amortized cost is
    - O(1)+1=O(1)

Finding minimum node

- Trivial by following **`H.min`** in O(1) time
- Potential of **`H`** does not change, thus amortized cost is equal to its O(1) actual cost

Uniting two Fibonacci heaps

- Simply concatenates root lists of given heaps and determines the new minimum in O(1) time

```
FIB-HEAP-UNION(H1, H2)
H = MAKE-FIB-HEAP()
H.min = H1.min
concatenate the root list of H2 with the root list of H
if (H1.min == NIL) or (H2.min ≠ NIL and H2.min.key < H1.min.key)
    H.min = H2.min
H.n = H1.n + H2.n
return H
```

Uniting two Fibonacci heaps

- No change in potential since

$$\phi(H)-[\phi(H_1)+\phi(H_2)]$$
$$=(t(H)+2m(H))-[(t(H_1)+2m(H_1))+(t(H_2)+2m(H_2))]$$
$$=0$$
$$\text{since } t(H)=t(H_1)+t(H_2) \text{ and } m(H)=m(H_1)+m(H_2)$$

- Amortized cost then is equal to its actual O(1) cost

Extracting the minimum

- Most complicated operation so far with delayed consolidation of trees in the root taking place
- Works as follows:
  - Make a root out of each of the minimum node's children
  - Remove the minimum node from root list
  - Consolidate the root list by linking roots of equal degree until at most one root remains of each degree

Extracting the minimum

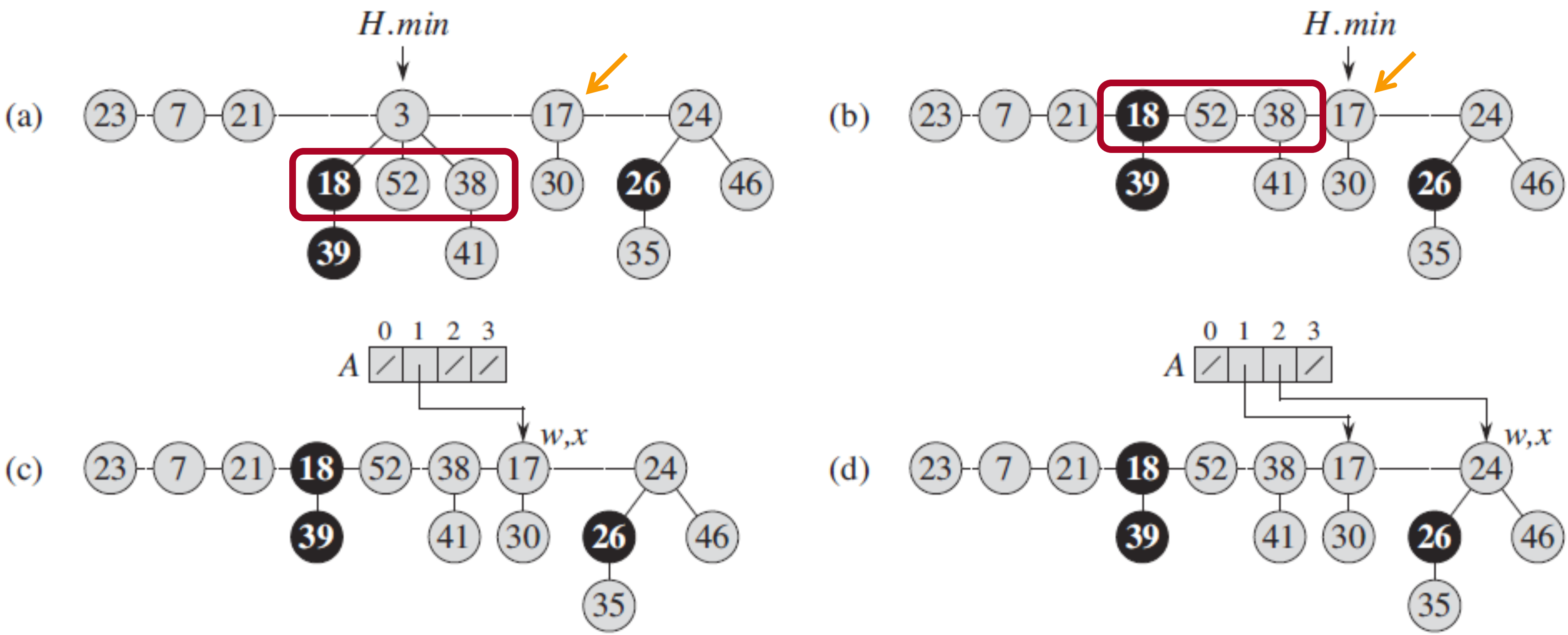
```
FIB-HEAP-EXTRACT-MIN(H)
z = H.min
if z ≠ NIL
    for each child x of z
        add x to the root list of H
        x.p = NIL
    remove z from the root list of H
    if z == z.right
        H.min = NIL
    else H.min = z.right
        CONSOLIDATE(H)
    H.n = H.n − 1
return z
```

Consolidation during extracting

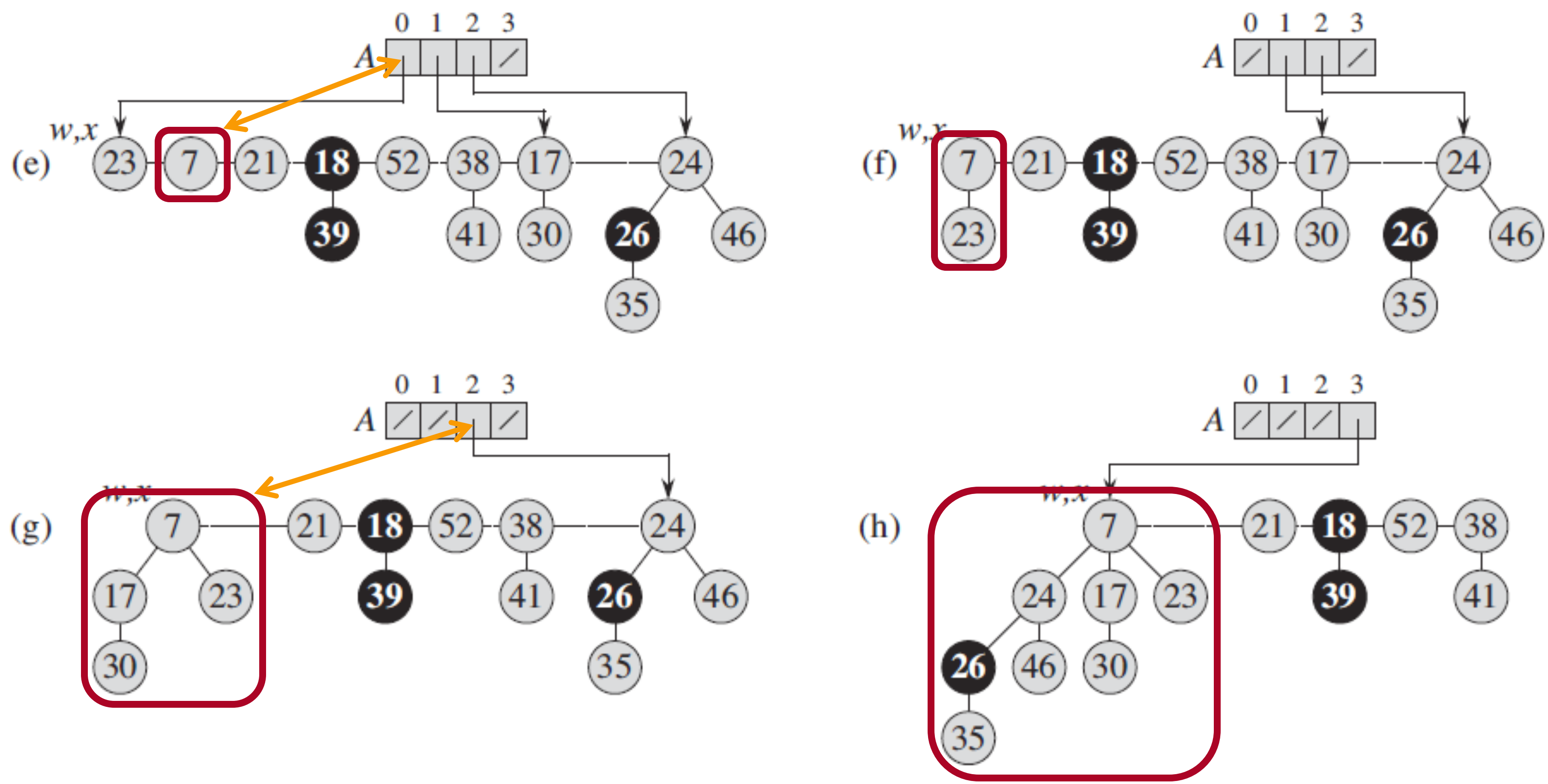
- Repeatedly execute until every root has a distinct degree:
  - Find two roots x and y with same degree **`x.key ≤ y.key`**
  - Link **`y`** to **`x`** by removing **`y`** from root list and making **`y`** a child of **`x`** with FIB-HEAP-LINK
    - Increments **`x.degree`** and clears **`y.mark`**

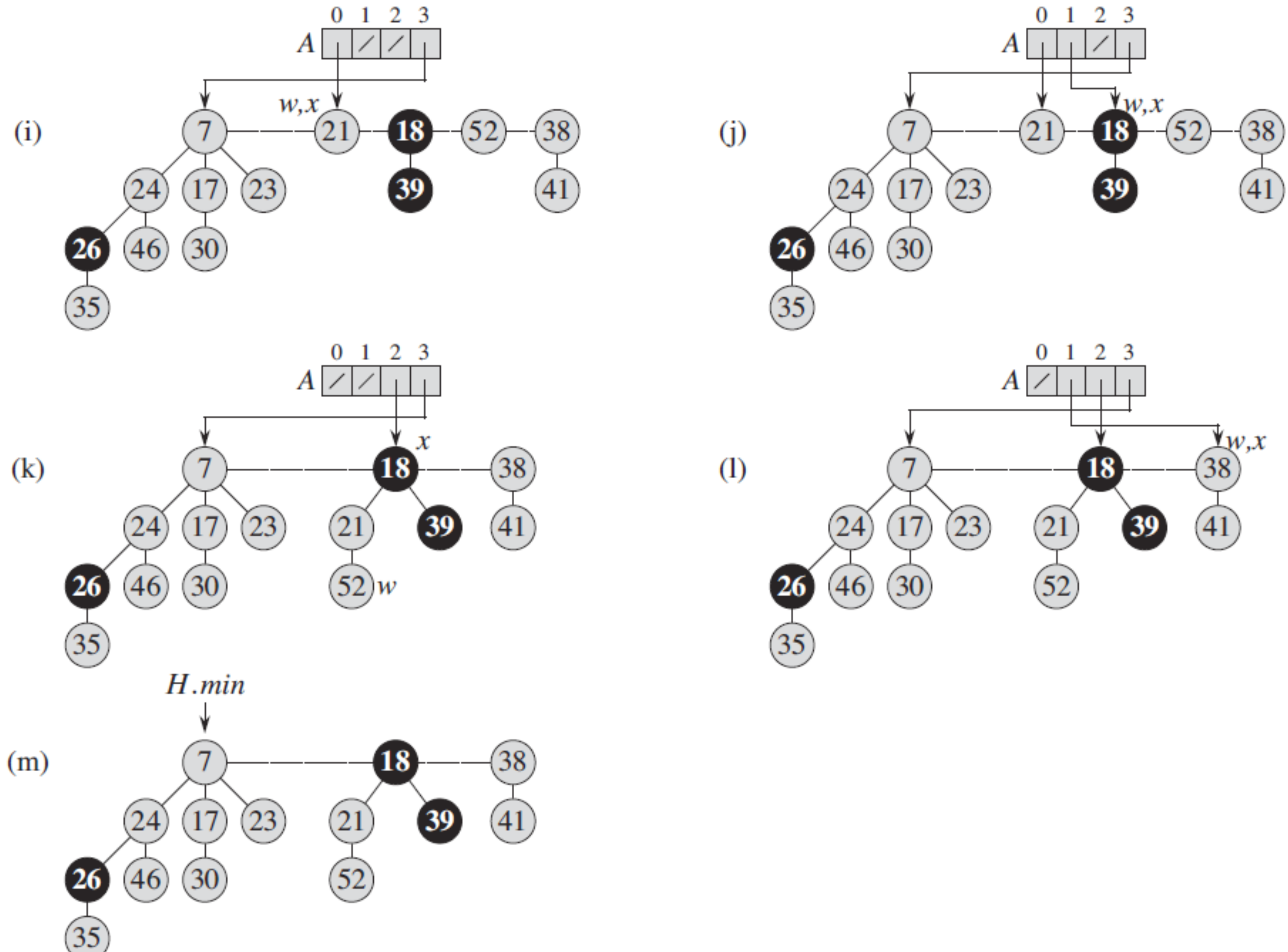
```
FIB-HEAP-LINK(H, y, x)
remove y from the root list of H
make y a child of x, incrementing x.degree
y.mark = FALSE
```

Consolidation during extracting



Consolidation during extracting




(i)
A
0 1 2 3
w,x
7 21 18 52 38
24 17 23 39 41
26 46 30
35
(j)
A
0 1 2 3
w,x
7 21 18 52 38
24 17 23 39 41
26 46 30
35
(k)
A
0 1 2 3
x
7 18 38
24 17 23 21 39 41
26 46 30 52 w
35
(l)
A
0 1 2 3
w,x
7 18 38
24 17 23 21 39 41
26 46 30 52
35
(m)
H.min
7 18 38
24 17 23 21 39 41
26 46 30 52
35

Consolidation during extracting

```
CONSOLIDATE(H)
let A[0 . . D(H.n)] be a new array
for i = 0 to D(H.n)
    A[i] = NIL
for each node w in the root list of H
    x = w
    d = x.degree
    while A[d] ≠ NIL
        y = A[d]            // another node with the same degree as x
        if x.key > y.key
            exchange x with y
        FIB-HEAP-LINK(H, y, x)
        A[d] = NIL
        d = d + 1
    A[d] = x
H.min = NIL
for i = 0 to D(H.n)
    if A[i] ≠ NIL
        if H.min == NIL
            create a root list for H containing just A[i]
            H.min = A[i]
        else insert A[i] into H's root list
            if A[i].key < H.min.key
                H.min = A[i]
```

Consolidation during extracting

```
CONSOLIDATE(H)
let A[0 .. D(H.n)] be a new array
for i = 0 to D(H.n)
    A[i] = NIL
for each node w in the root list of H
    x = w
    d = x.degree
    while A[d] ≠ NIL
        y = A[d]          // another node with the same degree as x
        if x.key > y.key
            exchange x with y
        FIB-HEAP-LINK(H, y, x)
        A[d] = NIL
        d = d + 1
    A[d] = x
H.min = NIL
for i = 0 to D(H.n)
    if A[i] ≠ NIL
        if H.min == NIL
            create a root list for H containing just A[i]
            H.min = A[i]
        else insert A[i] into H's root list
            if A[i].key < H.min.key
                H.min = A[i]
```

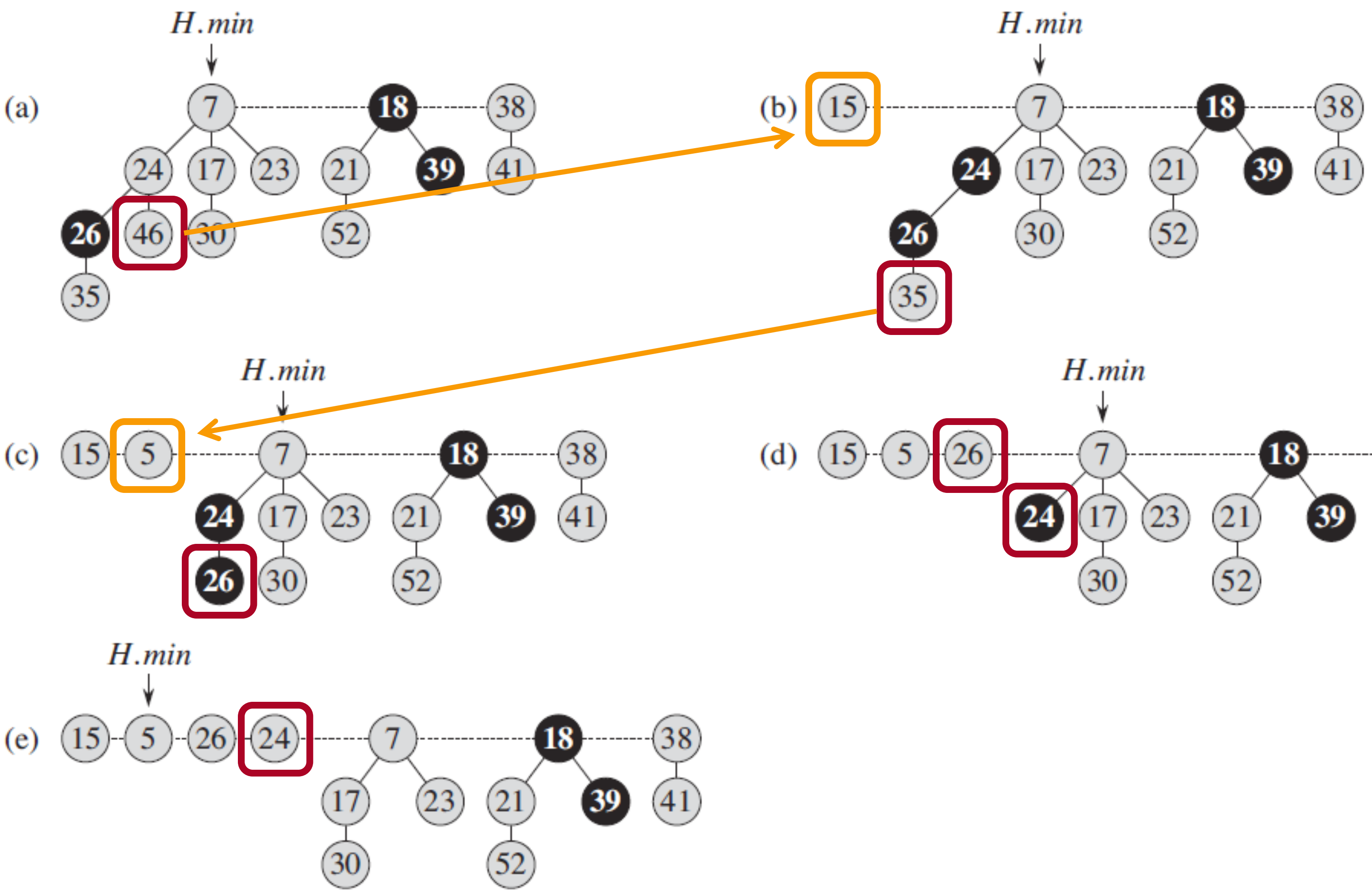
Consolidation during extracting

- Amortized cost is at most

$$
\begin{aligned}
\hat{C}_i &= C_i + \phi(D_i) - \phi(D_{i-1}) \\
&= O(D(n) + t(H)) + ((D(n)+1) + 2m(H)) - (t(H) + 2m(H)) \\
&= O(D(n)) + O(t(H)) - t(H) \\
&= O(D(n))
\end{aligned}
$$

- since at most **`D(n)+1`** roots remain and no nodes become marked during the operation
- Intuitively cost of performing each link is paid for by the reduction in potential due to the link's reducing the number of roots by one
- Since **`D(n)=O(lg n)`**, amortized cost is **`O(lg n)`**.

Decreasing a key



Decreasing a key

```
FIB-HEAP-DECREASE-KEY(H, x, k)
if k > x.key
    error "new key is greater than current key"
x.key = k
y = x.p
if y ≠ NIL and x.key < y.key
    CUT(H, x, y)
    CASCADING-CUT(H, y)
if x.key < H.min.key
    H.min = x
```

```
CUT(H, x, y)
remove x from the child list of y, decrementing y.degree
add x to the root list of H
x.p = NIL
x.mark = FALSE
```

```
CASCADING-CUT(H, y)
z = y.p
if z ≠ NIL
    if y.mark == FALSE
        y.mark = TRUE
    else CUT(H, y, z)
        CASCADING-CUT(H, z)
```

Decreasing a key

- Amortized cost is

$$\hat{C}_i = C_i + \phi(D_i) - \phi(D_{i-1})$$
$$= O(c) + (t(H) + c) + 2(m(H) - c + 2) - (t(H) + 2m(H))$$
$$= O(c) + 4 - c = O(1)$$

- where each FIB-HEAP-DECREASE-KEY results in **c** calls to CASCADING-CUT

Deleting a node

- First make **x** the minimum node by decreasing its key to -∞, and then extract it

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

- Same amortized cost **O(D(n))=O(lg n)** as extraction

Bounding maximum degree

- **Lemma** Let **`x`** be any node in a Fibonacci heap, and suppose that **`x.degree=k`**. *Let* $y_1, y_2, \ldots y_k$ denote the children of **`x`** in the order in which they were linked to **`x`**, from the earliest to the latest. Then, $y_1.degree \geq 0$ *and* $y_i.degree \geq i-2$ for $i=2,3,\ldots,k$.

Bounding maximum degree

- **Lemma** For all integers $\mathbf{k} \geq \mathbf{0}$,

$$F_{k+2} = 1 + \sum_{i=0}^{k} F_i$$

- **Proof** Use induction on $\mathbf{k}$

Bounding maximum degree

- **Lemma** For all integers **k≥0**,

$$F_{k+2} \geq \phi^k$$

- **Proof** Use induction on **k**

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

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

Bounding maximum degree

- **Lemma** Let **`x`** be any node in a Fibonacci heap, and let **`k=x.degree`**. Then,

$$size(x) \geq F_{k+2} \geq \phi^k \text{, where } \phi = (1+\sqrt{5})/2$$

- **Proof**

$$size(x) \geq s_k \geq 2 + \sum_{i=2}^{k} s_{y_i.\text{degree}} \geq 2 + \sum_{i=2}^{k} s_{i-2}$$

$$s_k \geq 2 + \sum_{i=2}^{k} s_{i-2} \geq 2 + \sum_{i=2}^{k} F_i = 1 + \sum_{i=0}^{k} F_i = F_{k+2} \geq \phi_k$$

Bounding maximum degree

- **Corollary** The maximum degree **`D(n)`** of any node in an **`n`**-node Fibonacci heap is **`O(lg n)`**.
- **Proof**

$$n \geq size(x) \geq \phi^k \Rightarrow \log_\phi n \geq k$$