Lakehead UNIVERSITY

COMP 4433: Algorithm Design and Analysis

Dr. Y. Gu

March 20, 2023 (Lecture 16)

Amortized Analysis (Dynamic Table) (Chapter 17)

Summary of Amort. Analysis

- The Aggregate analysis determines the upper bound $T(n)$ on the total cost of a sequence of n operations, then calculates the amortized cost to be $T(n) / n$.
- The accounting method is a form of aggregate analysis which assigns to each operation an *amortized cost* which may differ from its actual cost. Early operations have an amortized cost higher than their actual cost, which accumulates a saved "credit" that pays for later operations having an amortized cost lower than their actual cost.
- The potential method is a form of the accounting method where the saved credit is computed as a function (the "potential") of the state of the data structure. The amortized cost is the immediate cost plus the change in potential.

Topic IV: Dynamic Table

In some software environments, we shall assume that our software environment provides a memory-management system that can allocate and free blocks of storage on request. Thus, upon inserting an item into a full table, we can expand the table by allocating a new table with more slots than the old table had. Because we always need the table to reside in contiguous memory, we must allocate a new array for the larger table and then copy items from the old table into the new table. A common heuristic allocates a new table with twice as many slots as the old one. If the only table operations are insertions, then the load factor of the table is always at least 1/2, and thus the amount of wasted space never exceeds half the total space in the table.

Topic IV: Dynamic Table

To implement a Table-Delete operation, when the number of items in the table drops too low, we allocate a new, smaller table and then copy the items from the old table into the new one. We can then free the storage for the old table by returning it to the memory-management system.

We may halve the size when a deleting an item would cause the table to become less than half full. This strategy would guarantee that the load factor of the table never drops below ½; however, it can cause the amortized cost of an operation to be quite large.

Topic IV: Dynamic Table

Consider the following scenario. We perform n operations on a table T , where n is an exact power of 2. The first n/2 operations are insertions, which by our previous analysis cost a total of $\Theta(n)$. At the end of this sequence of insertions, T.num = T.size = n/2. For the second n/2 operations, we perform the following sequence:
insert, delete, delete, insert, insert, delete, delete, insert, insert, . . .

The first insertion causes the table to expand to size n. The two following deletions cause the table to contract back to size n/2.

Topic IV: Dynamic Table

Two further insertions cause another expansion, and so forth. The cost of each expansion and contraction is Θ(n), and there are Θ(n) of them. Thus, the total cost of the n operations is Θ((n^2)), making the amortized cost of an operation Θ(n).

We can improve upon this strategy by allowing the load factor of the table to drop below 1/2. Specifically, we continue to double the table size upon inserting an item into a full table, but we halve the table size when deleting an item causes the table to become less than 1/4 full, rather than 1/2 full as before. The load factor of the table is therefore bounded below by the constant 1/4.

Topic IV: Dynamic Table

To use the potential method to analysis the cost of a sequence of n Table-Insert and Table-Delete operations, we need to define the load factor α(T) = T.num/T.size. For an empty table, T.num = T.size = 0, we define α(T) = 1. Then we always have T.num = α(T) · T.size no matter the table is empty or not. We shall use the potential function

$$\Phi(T) = \begin{cases} 2 \cdot T.num - T.size & \text{if } \alpha(T) \geq 1/2, \\ T.size/2 - T.num & \text{if } \alpha(T) < 1/2. \end{cases} \tag{2}$$

Topic IV: Dynamic Table

Note that the potential of an empty table is 0 and that the potential is never negative. Thus the total amortized cost of a sequence of operations with respect to Φ provides an upper bound on the actual cost of the sequence.

- ❑ When the load factor is 1/2, the potential is 0.
- ❑ When the load factor is 1, Φ(T) = T.num, and thus the potential can pay for an expansion if an item is inserted.
- ❑ When the load factor is 1/4, we have T.size = 4T.num, which implies Φ(T) = T.num, and thus the potential can pay for a contraction if an item is deleted.

Topic IV: Dynamic Table

To analyze a sequence of n insert and delete operations, we let c_i denote the actual cost of the ith operation, $\hat{c}_i$ denote its amortized cost with respect to Φ, num_i denote the number of items stored in the table after the i-th operation, $size_i$ denote the total size of the table after the ith operation, α_i denote the load factor of the table after the i-th operation, and Φ_i denote the potential after the i-th operation.

Initially, $num_0 = 0$, $size_0 = 0$, $\alpha_0 = 1$, and $\Phi_0 = 0$.

Topic IV: Dynamic Table

We start with the case in which the i-th operation is Table-Insert. If $\alpha_{i-1} < 1/2$, the table cannot expand as a result of the operation, since the table expands only when $\alpha_{i-1} = 1$. If $\alpha_i < 1/2$ as well, then the amortized cost of the ith operation is

$$\begin{aligned}
\hat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) \\
&= 1 + (size_i/2 - num_i) - (size_i/2 - num_i - 1)) \\
&= 0
\end{aligned}$$

Topic IV: Dynamic Table

If $\alpha_{i-1} < 1/2$ but $\alpha_i \geq 1/2$, then

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (2 \cdot num_i - size_i) - (size_{i-1}/2 - num_{i-1}) \\
&= 1 + (2(num_{i-1} + 1) - size_{i-1}) - (size_{i-1}/2 - num_{i-1}) \\
&= 3 \cdot num_{i-1} - \frac{3}{2} size_{i-1} + 3 \\
&= 3\alpha_{i-1} size_{i-1} - \frac{3}{2} size_{i-1} + 3 \\
&< \frac{3}{2} size_{i-1} - \frac{3}{2} size_{i-1} + 3 \\
&= 3
\end{aligned}$$

Thus the amortized cost of an insert operation is at most 3.

Topic IV: Dynamic Table

When the i-th operation is delete, then $num_i = num_{i-1} - 1$. If $\alpha_{i-1} < 1/2$, then we must consider whether the operation causes the table to contract. If it does not, then $size_i = size_{i-1}$ and the amortized cost of the operation is

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) \\
&= 1 + (size_i/2 - num_i) - (size_i/2 - (num_i + 1)) \\
&= 2
\end{aligned}$$

If $\alpha_{i-1} < 1/2$ and the i-th operation does trigger a contraction, then the actual cost of the operation is $c_i = num_i + 1$, since we delete on item and move num_i items.

Topic IV: Dynamic Table

We have $size_i/2 = size_{i-1}/4 = num_{i-1} = num_i + 1$, and the amortized cost of the operation is

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= (num_i + 1) + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) \\
&= (num_i + 1) + ((num_i + 1) - num_i) - ((2 \cdot num_i + 2) - (num_i + 1)) \\
&= 1\,.
\end{aligned}$$

When the i-th operation is Table-Delete and $\alpha_{i-1} \geq 1/2$, the amortized cost is also bounded above by a constant.

In summary, since the amortized cost of each operation is bounded above by a constant, the actual time for any sequence of n operations on a dynamic table is O(n).

After Class

After class reading: Part IV 17.4

Fibonacci Heaps (Chapter 19)

– Another example of Amort. Analysis

Introduction

Fibonacci heaps provide operations on “mergeable heap”.

A **mergeable heap** is any data structure that supports the following five operations, in which each element has a key:

- Make-Heap() creates and returns a new heap containing no elements.
- Insert(H, x) inserts element x, whose key has already been filled in, into heap H.
- Minimum(H) returns a pointer to the element in heap H whose key is minimum.
- Extract-Min(H) deletes the element from heap H whose key is minimum, returning a pointer to the element.
- Union(H_1, H_2) creates and returns a new heap that contains all the elements of heaps H_1 and H_2. Heaps H_1 and H_2 are “destroyed” by this operation.

Introduction

In addition to the mergeable-heap operations above, Fibonacci heaps also support the following two operations:

- Decrease-Key(H, x, k) assigns to element x within heap H the new key value k, which we assume to be no greater than its current key value.
- Delete(H, x) deletes element x from heap H.

Note that here we just consider mergeable min-heaps. The mergeable max-heaps can be defined similarly.

Several Fibonacci-heap operations run in constant amortized time, which makes this data structure well suited for applications that invoke these operations frequently.

Structure of Fibonacci Heaps

A Fibonacci heap is a collection of rooted trees that are min-heap (or max-heap) ordered. In a Fibonacci heap, each node x contains a pointer $x.p$ to its parent and a pointer $x.child$ to any one of its children.

The children of x are linked together in a circular, doubly linked list. Each child y in a child list has pointers $y.left$ and $y.right$ that point to y's left and right siblings. If node y is an only child, then $y.left = y.right = y$.

Each node has two other attributes. We store the number of children in the child list of node x in $x.degree$. The boolean-valued attribute $x.mark$ indicates whether node x has lost a child since the last time x was made the child of another node.

Structure of Fibonacci Heaps

Newly created nodes are unmarked, and a node x becomes unmarked whenever it is made the child of another node. Until we look at the Decrease-Key operation later, we will just set all mark attributes to FALSE.

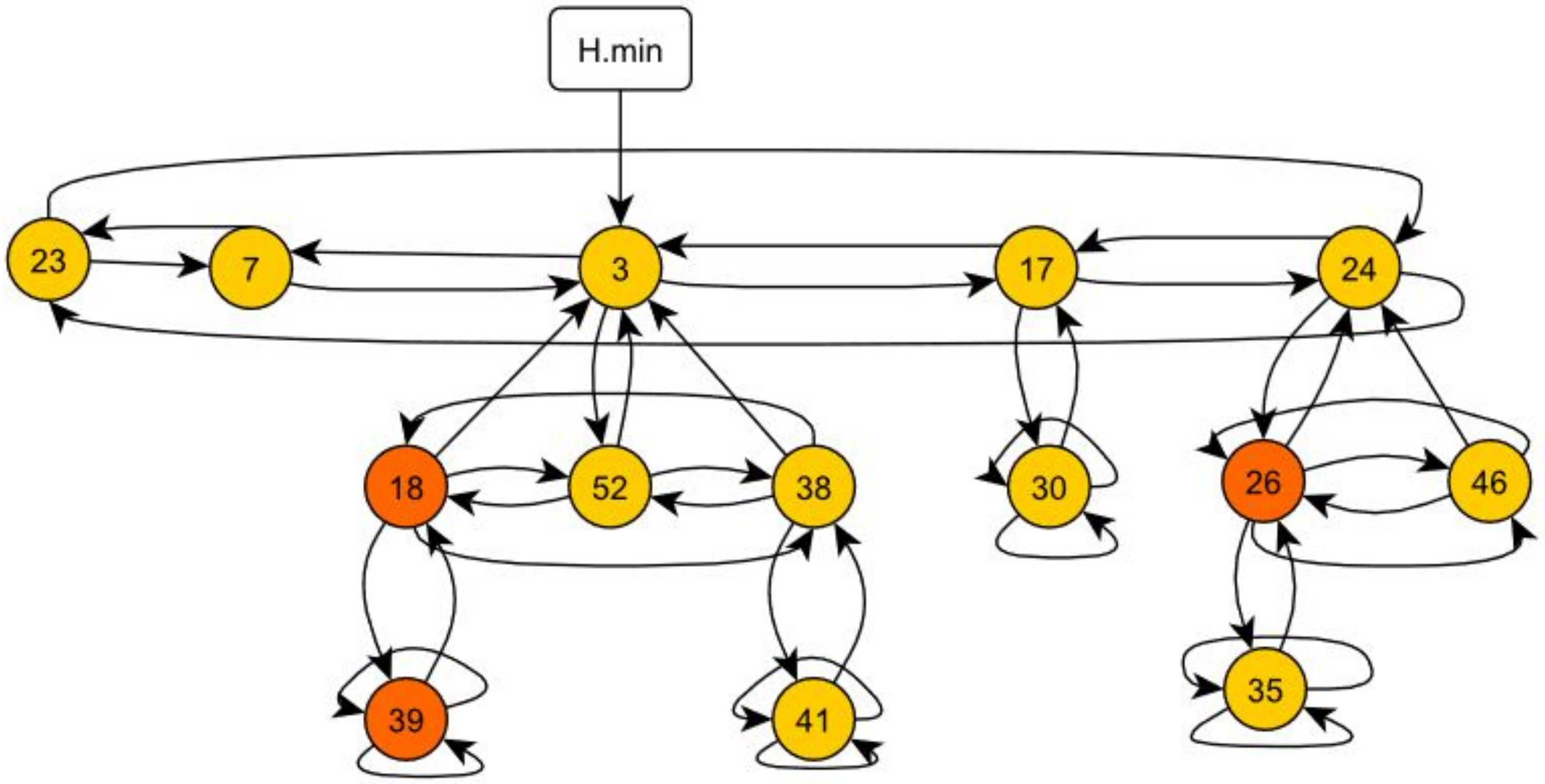
We access a given Fibonacci heap H by a pointer $H.min$ to the root of a tree containing the minimum key; we call this node the minimum node of the Fibonacci heap. If more than one root has a key with the minimum value, then any such root may serve as the minimum node. When a Fibonacci heap H is empty, $H.min$ is NIL.

Structure of Fibonacci Heaps

The roots of all the trees in a Fibonacci heap are linked together using their left and right pointers into a circular, doubly linked list called the **root list** of the Fibonacci heap. The pointer $H.min$ thus points to the node in the root list whose key is minimum. Trees may appear in any order within a root list. We rely on one other attribute for a Fibonacci heap H : $H.n$, the number of nodes currently in H.

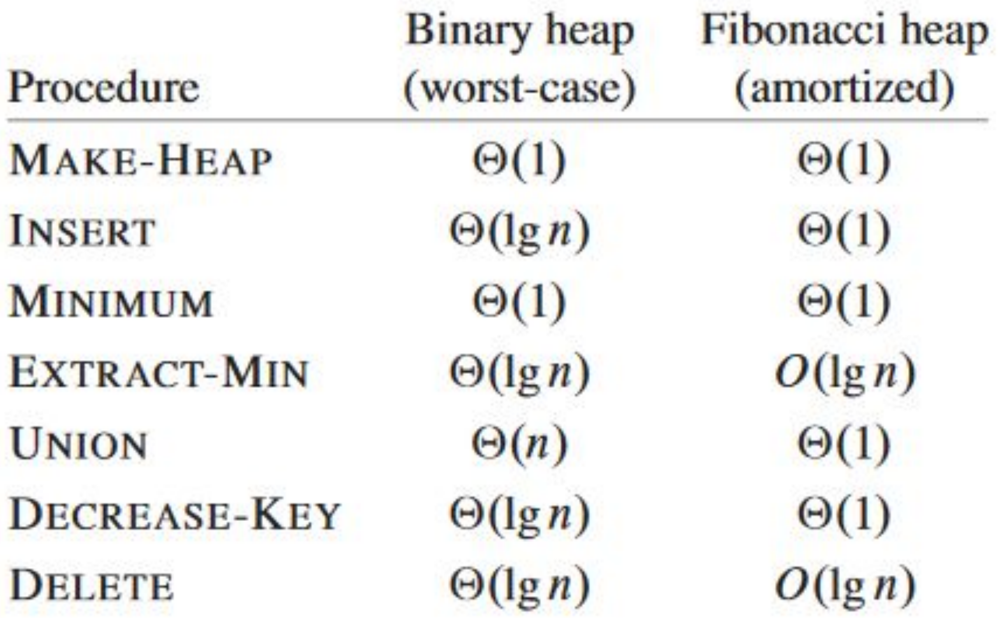
An example is displayed in the example in the next slide. In this example, the Fibonacci heap consists 5 mini-heap ordered trees. The roots of these trees are 23, 7, 3, 17, 24.

Structure of Fibonacci Heaps



Structure of Fibonacci Heaps

Circular, doubly linked lists have two advantages for use in Fibonacci heaps. Firstly, we can insert a node into any location or remove a node from anywhere in a circular in O(1) time. Secondly, given two such lists, we can concatenate them (or "splice" them together) into one circular, doubly linked list in O(1) time.



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

Binary Heaps v.s. Fibonacci Heaps

A Binary Heap is a complete binary tree, i.e., all levels are completely filled except possibly the last level and the last level has all keys as left as possible. This property of Binary Heap makes them suitable to be stored in an array. A Binary Heap is either Min Heap or Max Heap. In a Min Binary Heap, the key at root must be minimum among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary Heap.

A Fibonacci Heap is a collection of trees with Min-Heap or Max-Heap property. In Fibonacci Heap, trees can have any shape even all trees can be single nodes (This is unlike Binomial Heap where every tree has to be a Binomial Tree). Fibonacci Heap maintains a pointer to a minimum value (which is the root of a tree). All tree roots are connected using a circular doubly linked list, so all of them can be accessed using a single 'min' pointer.

Potential Function

For a given Fibonacci heap H, we indicate by $t(H)$ the number of trees in the root list of H and by $m(H)$ the number of marked nodes in H. We then define the potential $\Phi(H)$ of Fibonacci heap H by

$$\Phi(H) = t(H) + 2m(H) \qquad \text{(Book Ch19.1)}$$

The intuition will be discussed later.

The potential of a set of Fibonacci heaps is the sum of the potentials of its constituent Fibonacci heaps. We shall assume that a unit of potential can pay for a constant amount of work, where the constant is sufficiently large to cover the cost of any of the specific constant-time pieces of work that we might encounter.

Potential Function

We assume that a Fibonacci heap application begins with no heaps. The initial potential, therefore, is 0, and by equation (Ch19.1), the potential of H is nonnegative at all subsequent times.

From the equation of the sum of amort. costs (in Ch 17), an upper bound on the total amortized cost provides an upper bound on the total actual cost for the sequence of operations.

Maximum Degree

We assume that we know an upper bound $D(n)$ on the maximum degree of any node in an n-node Fibonacci heap and $D(n) \leq \lfloor lgn \rfloor$ (this property can be proved).

Mergeable-Heap Operations

The mergeable-heap operations on Fibonacci heaps delay work as long as possible.

If we were to start with an empty Fibonacci heap and then insert k nodes, the Fibonacci heap would consist of just a root list of k nodes. If we then perform an Extract-Min operation on Fibonacci heap H, after removing the node that H.min points to, we would have to look through each of the remaining k − 1 nodes in the root list to find the new minimum node.

When we go through the entire root list during the Extract-Min operation, we also consolidate nodes into min-heap-ordered trees to reduce the size of the root list.

After Extract-Min operation, each node in the root list has a degree that is unique within the root list, which leads to a root list of size at most D(n) + 1.

Mergeable-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 H.n = 0 and H.min = NIL; there are no trees in H. Because t(H) = 0 and m(H) = 0, the potential of the empty Fibonacci heap is Φ(H) = 0. The amortized cost of Make-Fib-Heap is thus equal to its O(1) actual cost.

Next we concern insertion.

Lakehead UNIVERSITY

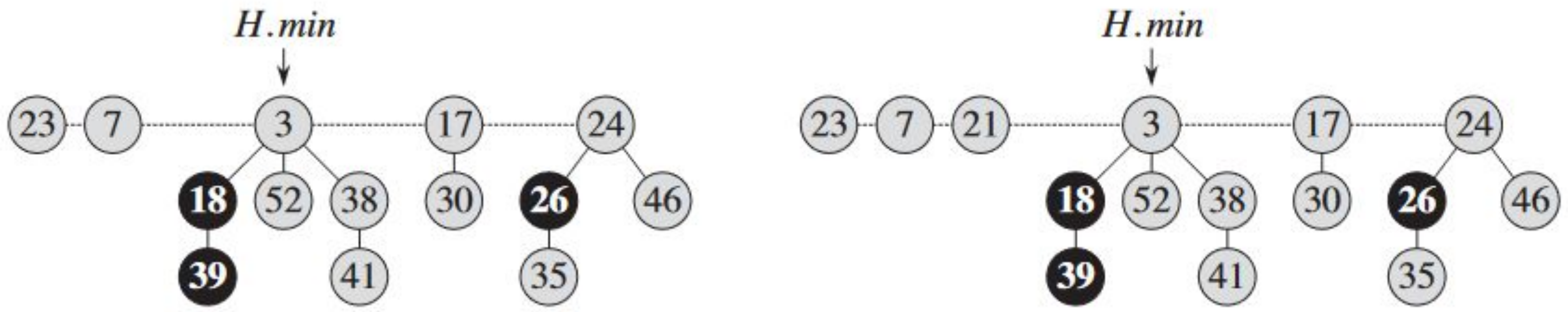
Mergeable-Heap Operations

Inserting a node

The following procedure inserts node x into Fibonacci heap H , assuming that the node has already been allocated and that $x{:}key$ has already been filled in.

```
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
```

Mergeable-Heap Operations



To determine the amortized cost, 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), so the increase in potential is ((t(H) + 1) + 2m(H)) − (t(H) + 2m(H)) = 1.

Since the actual cost is O(1), the amortized cost is O(1) + 1 = O(1).

Mergeable-Heap Operations

Finding the minimum node

The minimum node of a Fibonacci heap H is given by the pointer H:min, 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).

Uniting two Fibonacci heaps

Uniting Fibonacci heaps H_1 and H_2 simply concatenates the their root lists and then determines the new minimum node.

Mergeable-Heap Operations

```
FIB-HEAP-UNION($H_1, H_2$)
$H$ = MAKE-FIB-HEAP()
$H.min = H_1.min$
concatenate the root list of $H_2$ with the root list of $H$
if ($H_1.min$ == NIL) or ($H_2.min \neq$ NIL and $H_2.min.key < H_1.min.key$)
    $H.min = H_2.min$
$H.n = H_1.n + H_2.n$
return $H$
```

The change in potential is

$\Phi(H) - (\Phi(H_1) + \Phi(H_2))$
$= (t(H) + 2m(H)) - ((t(H_1) + 2m(H_1)) + (t(H_2) + 2m(H_2)))$
$= 0,$
because $t(H) = t(H_1) + t(H_2)$ and $m(H) = m(H_1) + m(H_2)$. The amortized cost of Fib-Heap-Union is therefore equal to its $O(1)$ actual cost.

Mergeable-Heap Operations

Extracting the minimum node

It is where the delayed work of consolidating trees in the root list finally occurs.

```
FIB-HEAP-EXTRACT-MIN($H$)
$z = H.min$
if $z \neq$ 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 $z$
```

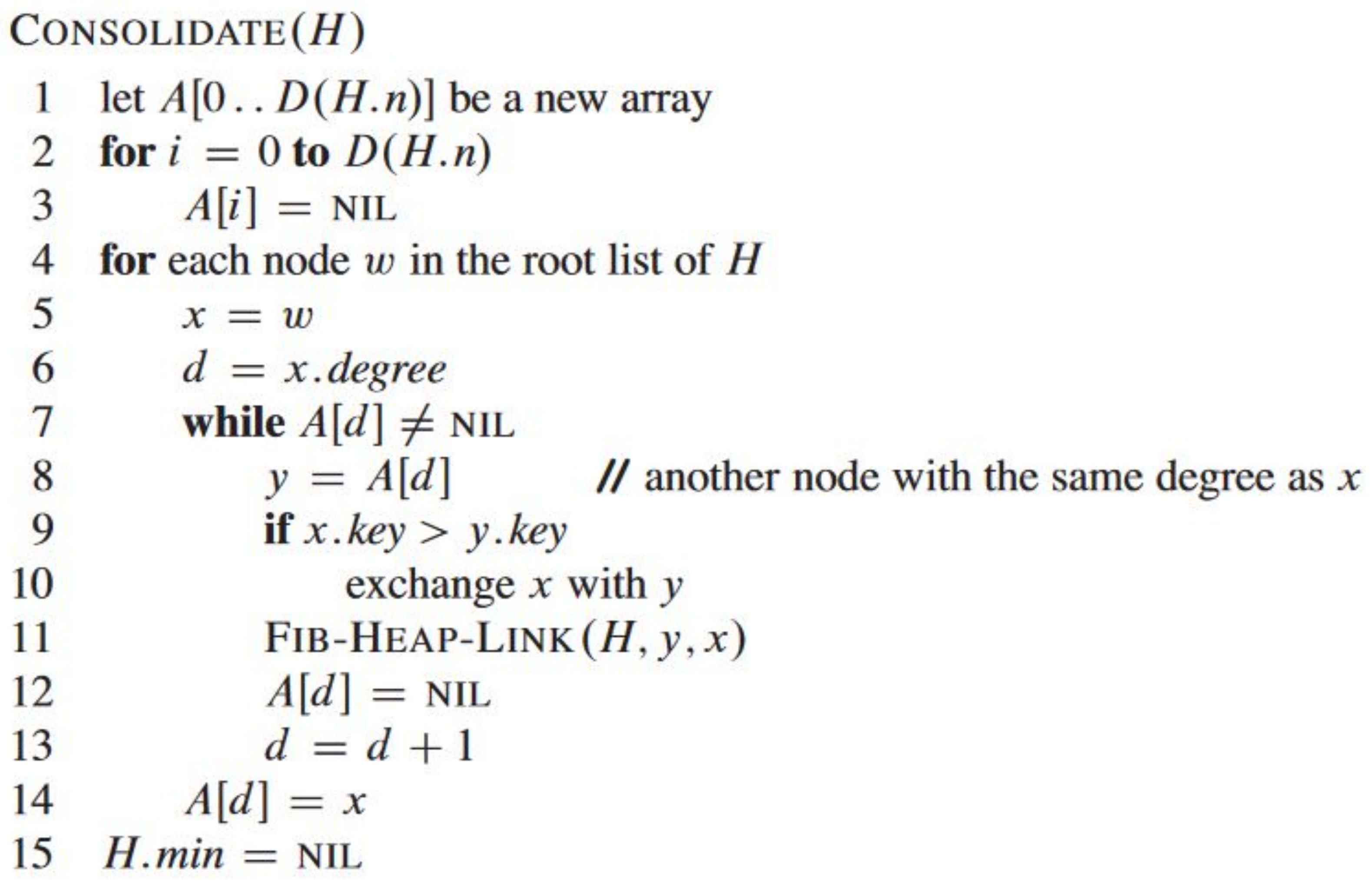
Mergeable-Heap Operations

The main steps of the procedure are as follows.

- Move all the children of z to the root list (lines 3 - 5).
- Remove z from the root list.
- If z is the only node in H, then set H as empty heap (lines 7-8).
- Otherwise, let H.min = z.right. But then H.min is not necessary the minimum node of H. So the procedure calls Consolidate to fix that problem.

The consolidate procedure not only fix the minimum problem but also reduced the number of trees in the Fibonacci heap.

Mergeable-Heap Operations



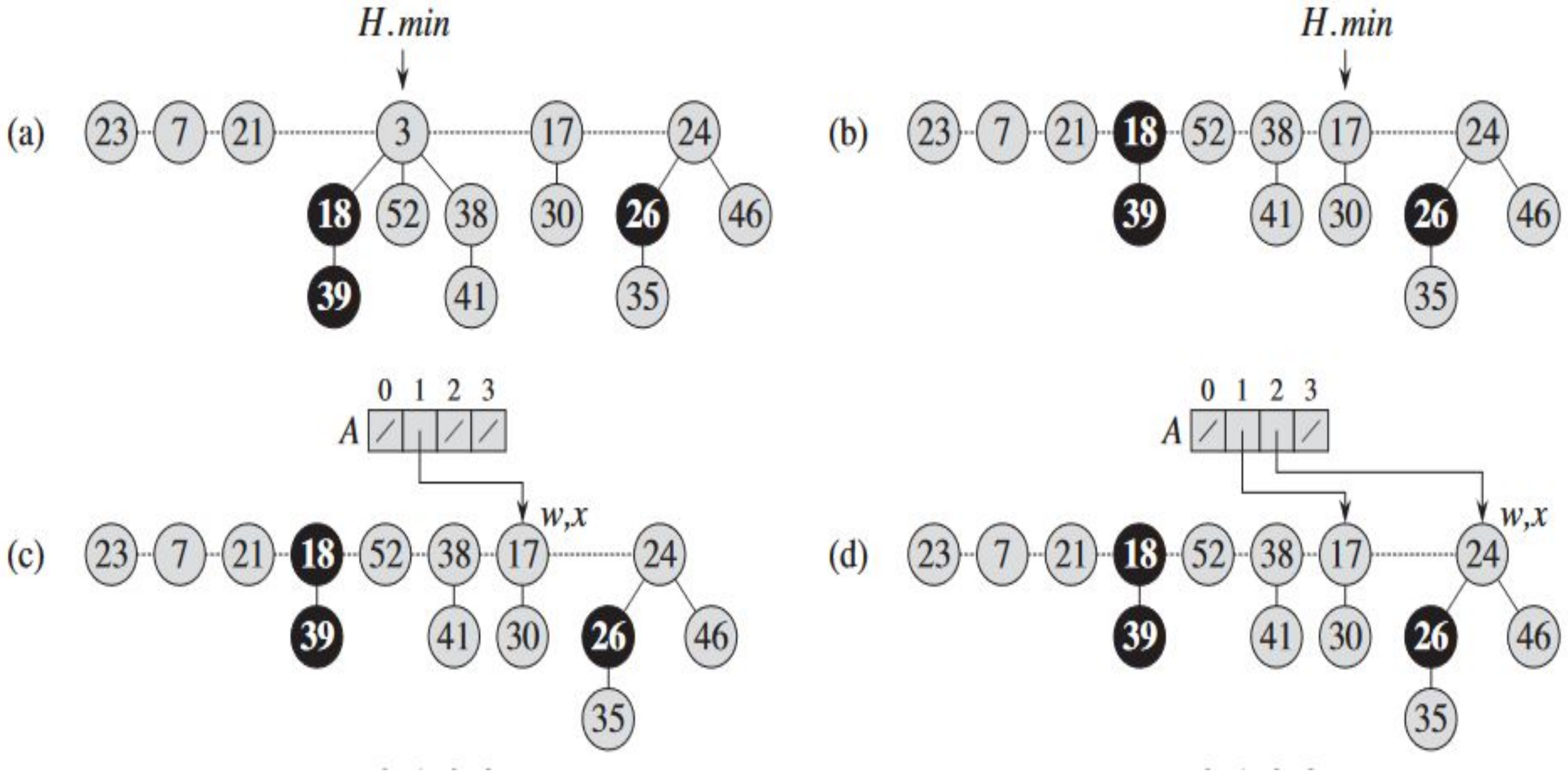
```
CONSOLIDATE($H$)
let $A[0\,..\,D(H.n)]$ be a new array
for $i = 0$ to $D(H.n)$
    $A[i] = \text{NIL}$
for each node $w$ in the root list of $H$
    $x = w$
    $d = x.degree$
    while $A[d] \neq \text{NIL}$
        $y = A[d]$            // another node with the same degree as $x$
        if $x.key > y.key$
            exchange $x$ with $y$
        FIB-HEAP-LINK($H, y, x$)
        $A[d] = \text{NIL}$
        $d = d + 1$
    $A[d] = x$
$H.min = \text{NIL}$
```

Mergeable-Heap Operations

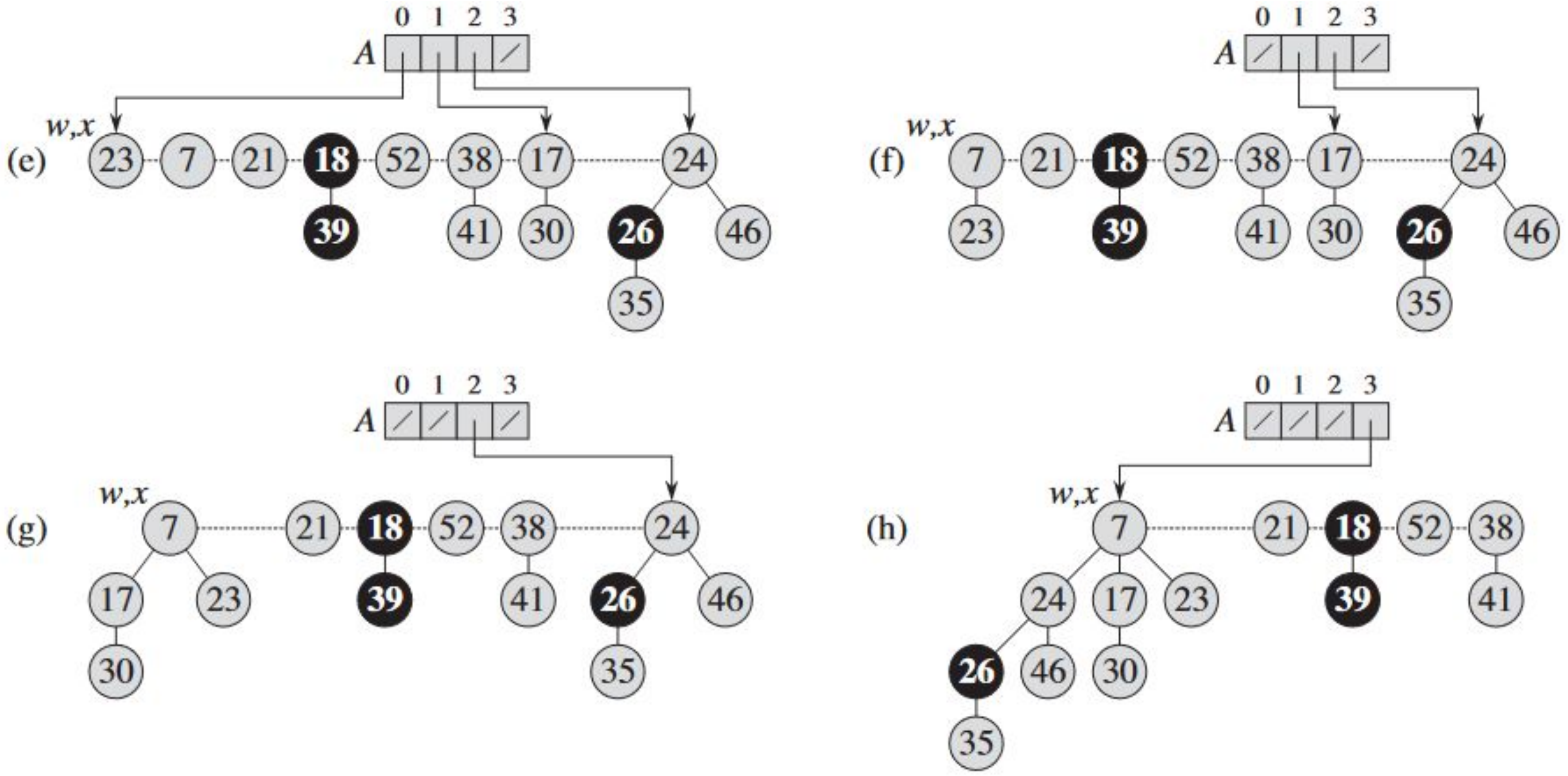
```
for $i = 0$ to $D(H.n)$
    if $A[i] \neq$ NIL
        if $H.min ==$ NIL
            create a root list for $H$ containing just $A[i]$
            $H.min = A[i]$
        else insert $A[i]$ into $H$'s root list
            if $A[i].key < H.min.key$
                $H.min = A[i]$
```

```
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
```

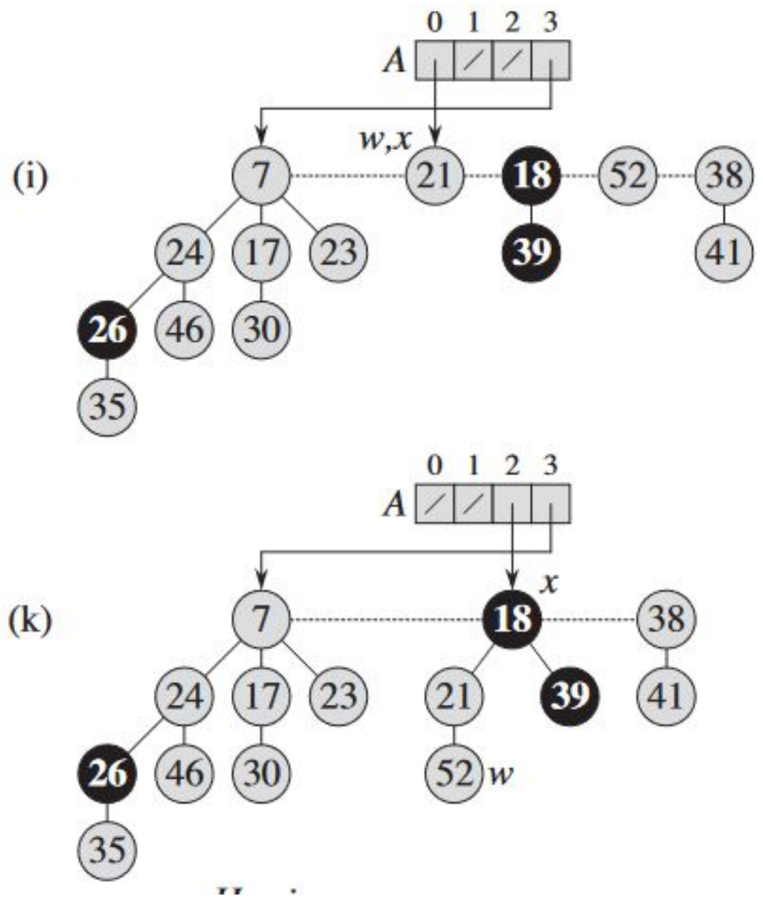
Delete Min Example (1)

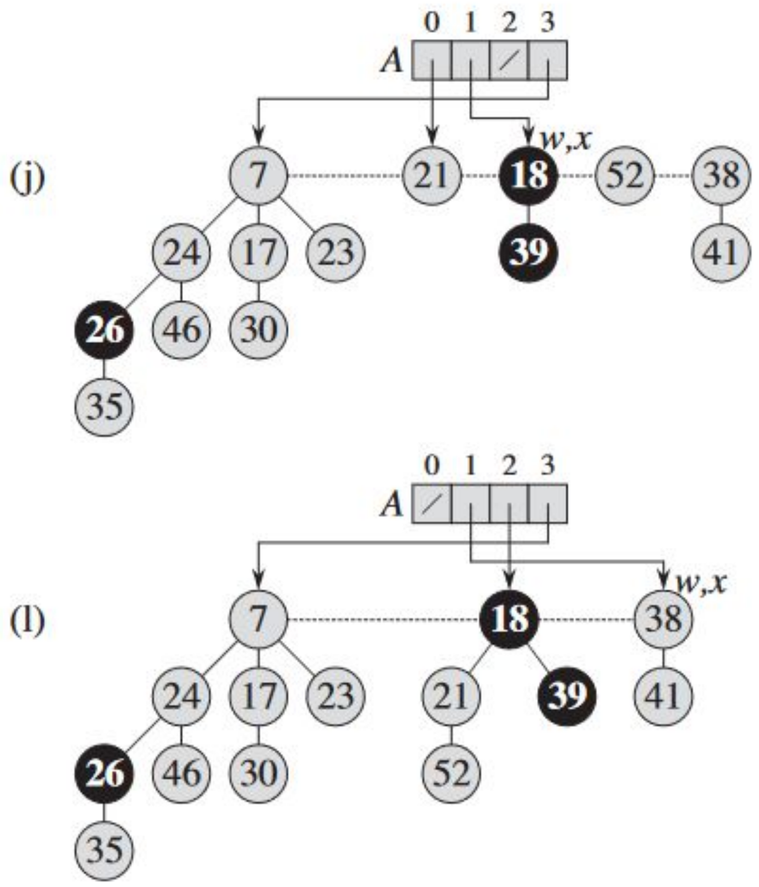


Delete Min Example (2)

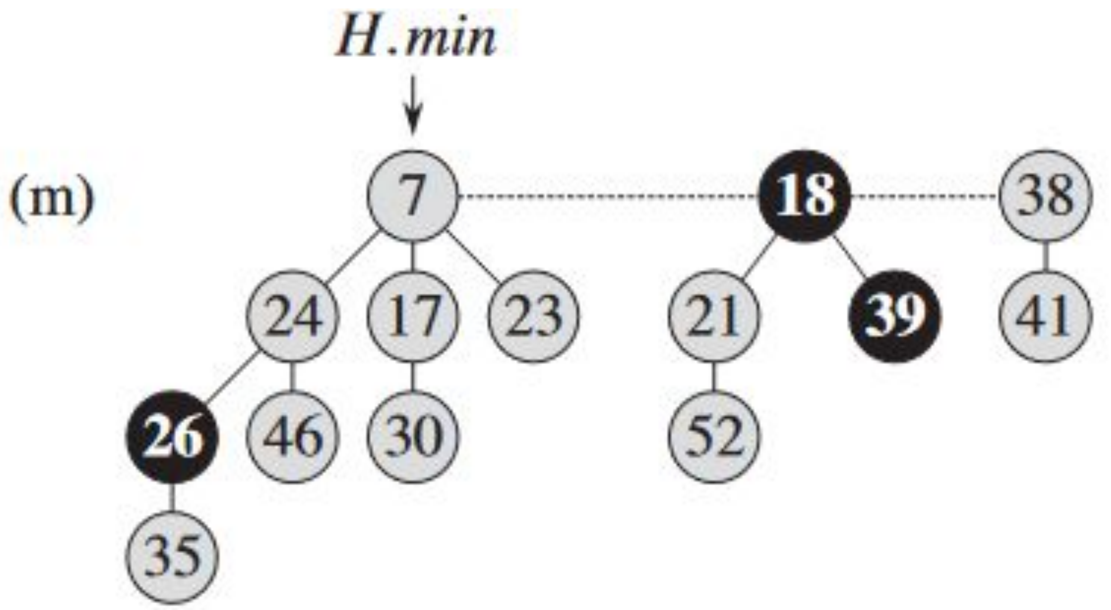


Delete Min Example (3)





Delete Min Example (4)



Mergeable-Heap Operations

The procedure Consolidate uses an auxiliary array A[0 . . . D(H.n)] to keep track of roots according to their degree. The main steps of the procedure are as follows.

- From line 4, let A[i] point to the node in the root list, that has degree i. If there are two nodes in the root list having same degree, then we put one node as a child of another node, that is done by the while loop in line 7. The if block in line 9 is used to keep the node with smaller key to remain in the root list.
- The Fib-Heap-Link is used to link one node as a child of another node.
- Line 15 empties the root list, and the remains are construct the renewed root list.

Mergeable-Heap Operations

Now we consider the amortized cost of extracting the minimum node of an n-node Fibonacci heap. Let H denote the Fibonacci heap just prior to the Fib-Heap-Extract-Min operation. We start by accounting for the actual cost of extracting the minimum node. An O(D(n)) contribution comes from Fib-Heap-Extract-Min processing at most D(n) children of the minimum node and from the work in lines 2-4 and 15-23 of Consolidate. It remains to analyze the contribution from the for loop of lines 4 -14 in Consolidate, for which we use an aggregate analysis. The size of the root list upon calling Consolidate is at most D(n) + t(H) − 1, since it consists of the original t(H) root-list nodes, minus the extracted root node, plus the children of the extracted node, which number is at most D(n).

Mergeable-Heap Operations

Within a given iteration of the for loop of lines 4 - 14, the number of iterations of the while loop of lines 7 - 13 depends on the root list.

But we know that every time through the while loop, one of the roots is linked to another, and thus the total number of iterations of the while loop over all iterations of the for loop is at most the number of roots in the root list.

Hence, the total amount of work performed in the for loop is at most proportional to D(n) + t(H). Thus, the total actual work in extracting the minimum node is O(D(n) + t(H)).

Mergeable-Heap Operations

The potential before extracting the minimum node is t(H)+2m(H), and the potential afterward is at most (D(n)+1)+2m(H), since at most D(n)+1 roots remain and no nodes become marked during the operation. The amortized cost is thus at most

$$\begin{aligned} & O(D(n)+t(H))+((D(n)+1)+2m(H))-(t(H)+2m(H)) \\ &= O(D(n))+O(t(H))-t(H) \\ &= O(D(n)), \end{aligned}$$

since we can scale up the units of potential to dominate the constant hidden in O(t(H)). We shall see later that D(n) = $O(lg\ n)$, so that the amortized cost of extracting the minimum node is $O(lg\ n)$.

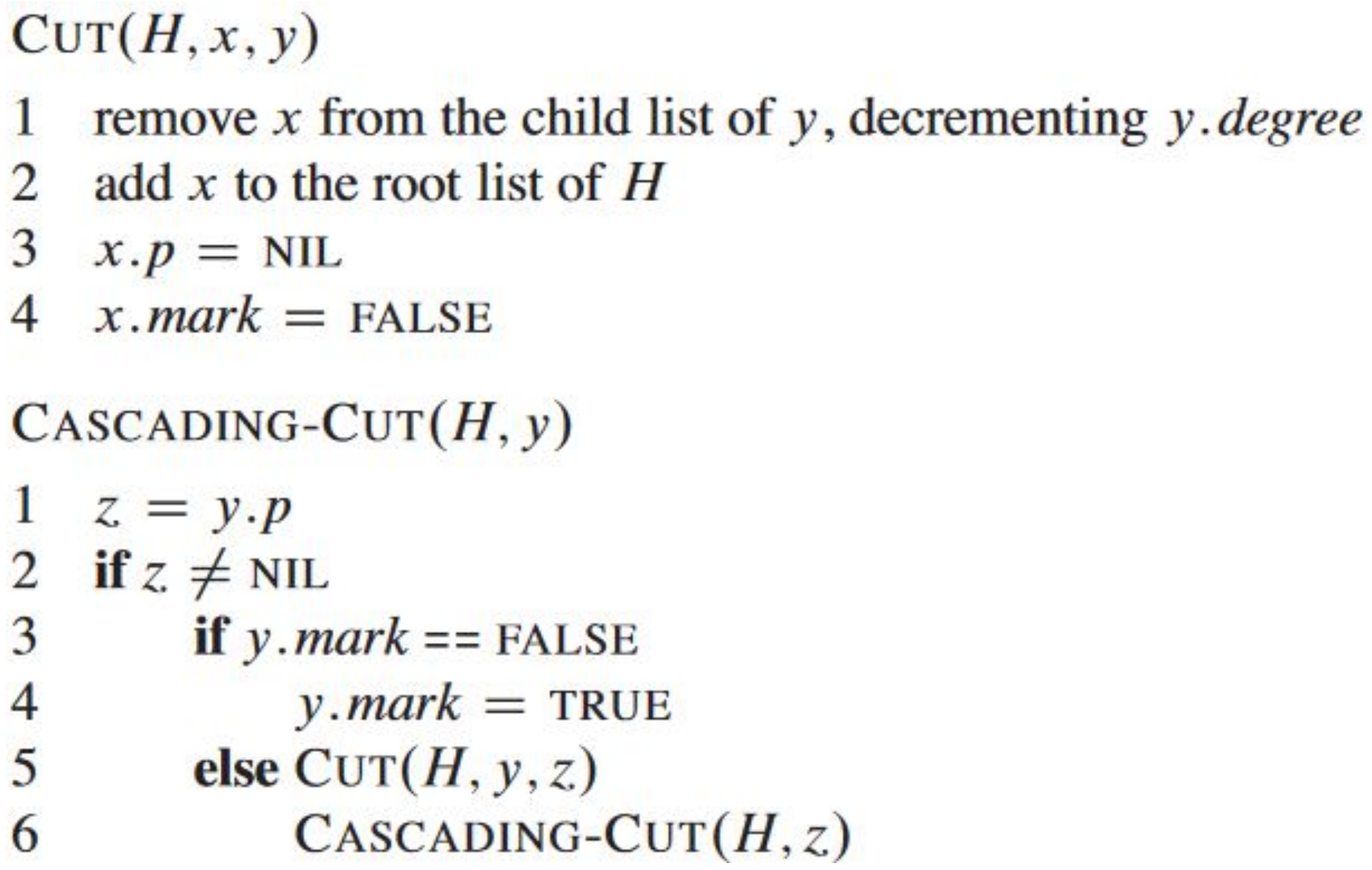
Fibonacci Heap Operations

Decreasing a key

We assume as before that removing a node from a linked list does not change any of the structural attributes in the removed node.

```
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 $y \neq$ NIL and $x.key < y.key$
    CUT$(H, x, y)$
    CASCADING-CUT$(H, y)$
if $x.key < H.min.key$
    $H.min = x$
```

Fibonacci Heap Operations



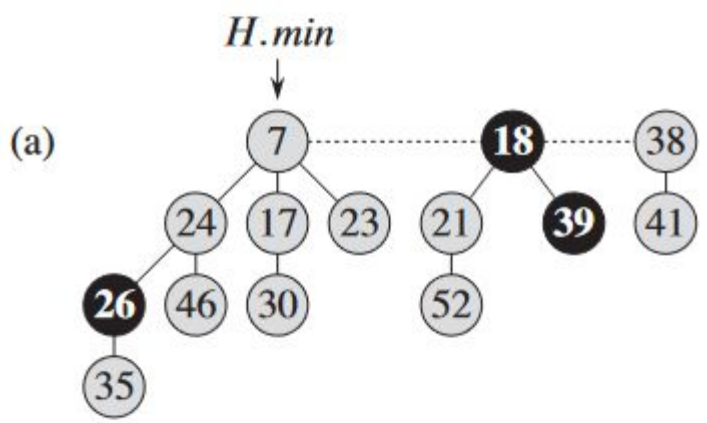
```
CUT($H, x, y$)
1  remove $x$ from the child list of $y$, decrementing $y.degree$
2  add $x$ to the root list of $H$
3  $x.p$ = NIL
4  $x.mark$ = FALSE

CASCADING-CUT($H, y$)
1  $z = y.p$
2  if $z \neq$ NIL
3      if $y.mark$ == FALSE
4          $y.mark$ = TRUE
5      else CUT($H, y, z$)
6          CASCADING-CUT($H, z$)
```

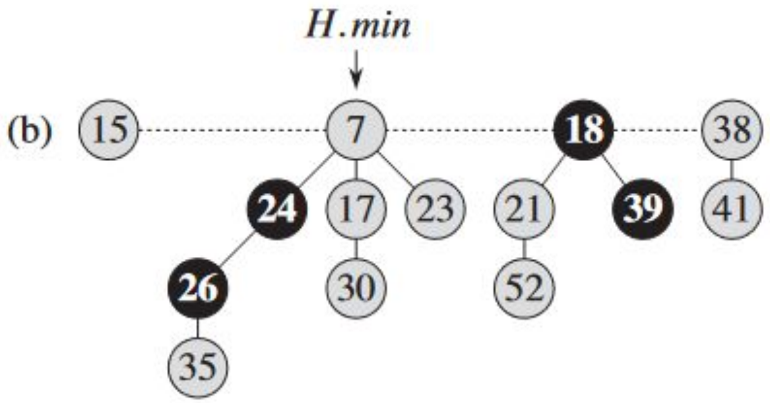
Fibonacci Heap Operations

The algorithm is working as follows.

- First lines 1- 2 check if k is less than the key of x. Then change the key of x to k in lines 4 - 5.
- If x is in the root list, or x is not a root, but x.key is larger then its parent y's key, then the change is fine.
- If x.key < y.key, then the min-heap order is violated. In this case, we cut x from y and put x into the root list.
- Then we use the mark attribute to obtain the desired time bound. If a node x was linked to (made the child of) another node and then two children of x were removed by cuts, then we move x to the root list.

Decreasing Key Example

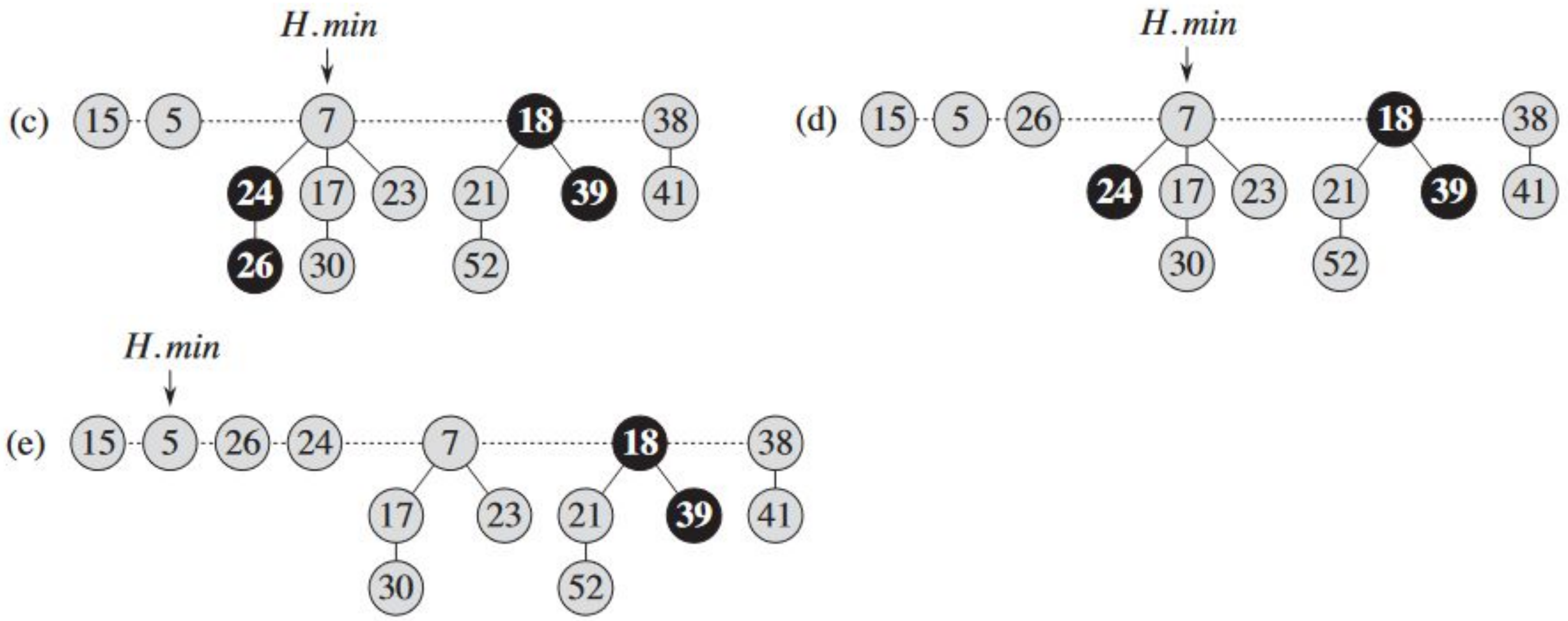




a) Original heap

b) Decrease key 46 to 15 (24 is marked)

Decreasing Key Example



c) - e) Key 35 decresed to 5. In part (c), the node with key 5 becomes a root. Its parent, with key 26, is marked, so a cascading cut occurs. The node with key 26 is cut from its parent and made an unmarked root in (d). Another cascading cut occurs in (e), since the node with key 24 is marked as well.

Fibonacci Heap Operations

We now consider the amortized cost of the operation Fib-Heap-Decrease-Key.

We start by determining its actual cost. The Fib-Heap-Decrease-Key procedure takes O(1) time, plus the time to perform the cascading cuts. Suppose that a given invocation of Fib-Heap-Decrease-Key results in c calls of Cascading-Cut (recursively called). Each call of Cascading-Cut takes O(1) exclusive of recursive calls. Thus, the actual cost of Fib-Heap-Decrease-Key including all recursive calls, is O(c).

Fibonacci Heap Operations

We next compute the change in potential. Let H denote the Fibonacci heap just prior to the Fib-Heap-Decrease-Key operation. The call to Cut in Fib-Heap-Decrease-Key creates a new tree rooted at node x and clears x's mark bit (which may have already been FALSE). Each call of Cascading-Cut, except for the last one, cuts a marked node and clears the mark bit. Afterward, the Fibonacci heap contains t(H) + c trees and at most m(H) − c + 2 marked nodes (c − 1 were unmarked by cascading cuts and the last call of Cascading-Cut may have marked a node). The change in potential is therefore at most

$$((t(H) + c) + 2(m(H) - c + 2)) - (t(H) + 2m(H)) = 4 - c \; .$$

Fibonacci Heap Operations

So the amortized cost of Fib-Heap-Decrease-Key is at most

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

since we can scale up the units of potential to dominate the constant hidden in O(c).

When a marked node y is cut by a cascading cut, its mark bit is cleared, which reduces the potential by 2. One unit of potential pays for the cut and the clearing of the mark bit, and the other unit compensates for the unit increase in potential due to node y becoming a root.

That is why we defined the potential function to include a term that is twice the number of marked nodes.

Fibonacci Heap Operations

Deleting a node

To delete a node from a Fibonacci heap, we use the following.

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

The amortized time of Fib-Heap-Delete is the sum of the O(1) amortized time of Fib-Heap-Decrease-Key and the O(D(n)) amortized time of Fib-Heap-Extract-Min. Since we shall see later that D(n) = O(lg n), the amortized time of Fib-Heap-Delete is O(lg n).

After Class

After class reading: Part V 19.1-19.3.