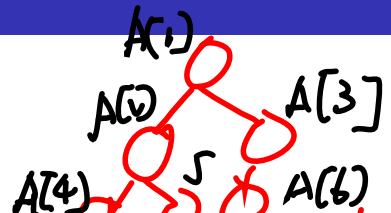
Heaps, Heapsort, and Fibonacci Heaps

Congduan Li

Chinese University of Hong Kong, Shenzhen

congduan.li@gmail.com

Oct 31 & Nov 2, 2017



All nodes except leaves have 2 children

Heap data structure

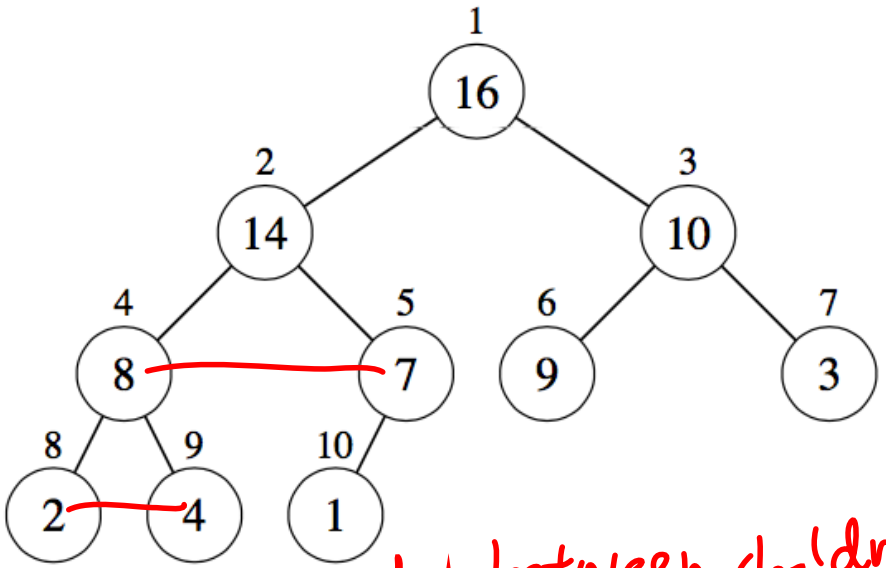
- Heap A (*not* garbage-collected storage) is a nearly complete binary tree.
  - ***Height*** of node = # of edges on a longest simple path from the node down to a leaf.
  - ***Height*** of heap = height of root = $\Theta(\lg n)$.
- A heap can be stored as an array A.
  - Root of tree is $A[1]$.
  - Parent of $A[i] = A[\lfloor i/2 \rfloor]$.
  - Left child of $A[i] = A[2i]$.
  - Right child of $A[i] = A[2i+1]$.
  - Computing is fast with binary representation implementation.

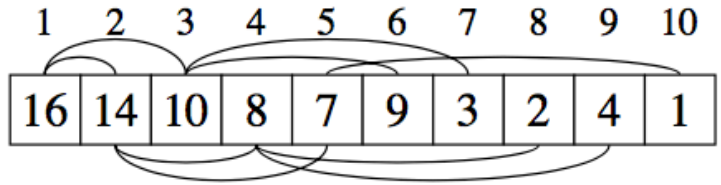
for $2i > n$, $A[2i] = \phi$ (leaves)

$i > \frac{n}{2}$

$\times 2 / -2$

Example





No strict order between children
8 > 7 but 2 < 4
max-heap max value at root
(relationship between parents & child)

Heaps property

- For max-heaps (largest element at root), ***max-heap property:*** for all nodes i, excluding the root, $A[\text{PARENT}(i)] \geq A[i]$.
- For min-heaps (smallest element at root), ***min-heap property:*** for all nodes i, excluding the root, $A[\text{PARENT}(i)] \leq A[i]$.

By induction and transitivity of $\leq$, the max-heap property guarantees that the maximum element of a max-heap is at the root. Similar argument for min-heaps.

The heapsort algorithm we'll show uses max-heaps.

Note: In general, heaps can be k-ary tree instead of binary.

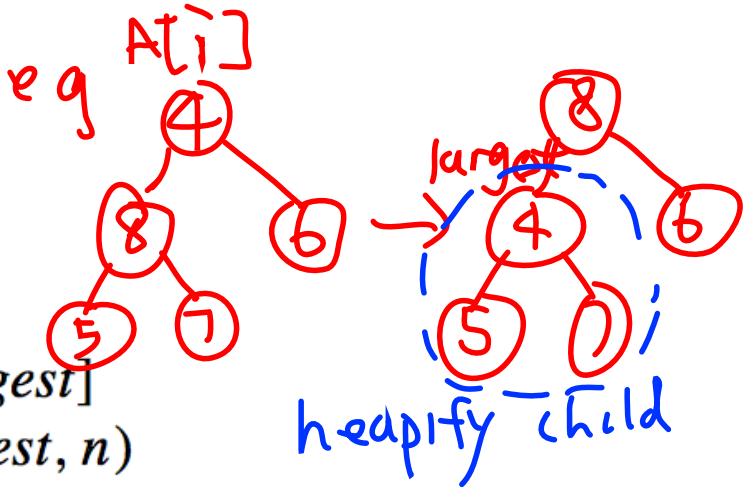
Maintaining heaps property

MAX-HEAPIFY is important for manipulating max-heaps. It is used to maintain the max-heap property.

- Before MAX-HEAPIFY, $A[i]$ may be smaller than its children.
- Assume left and right subtrees of i are max-heaps.
- After MAX-HEAPIFY, subtree rooted at i is a max-heap.

MAX-HEAPIFY(A, i, n) max index of the array cannot go out of the index

$l \leftarrow$ LEFT(i)
$r \leftarrow$ RIGHT(i)
if $l \leq n$ and $A[l] > A[i]$
  then $largest \leftarrow l$
  else $largest \leftarrow i$
if $r \leq n$ and $A[r] > A[largest]$
  then $largest \leftarrow r$
if $largest \neq i$
  then exchange $A[i] \leftrightarrow A[largest]$
    MAX-HEAPIFY$(A, largest, n)$

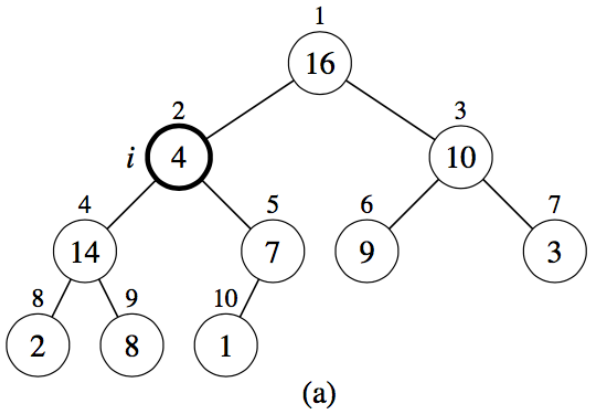


How it works

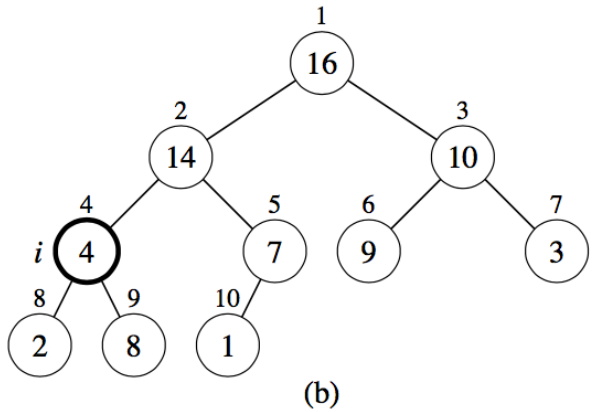
The way MAX-HEAPIFY works:

- Compare $A[i]$, $A[\text{LEFT}(i)]$, and $A[\text{RIGHT}(i)]$.
- If necessary, swap $A[i]$ with the larger of the two children to preserve heap property.
- Continue this process of comparing and swapping down the heap, until subtree rooted at i is max-heap. If we hit a leaf, then the subtree rooted at the leaf is trivially a max-heap.

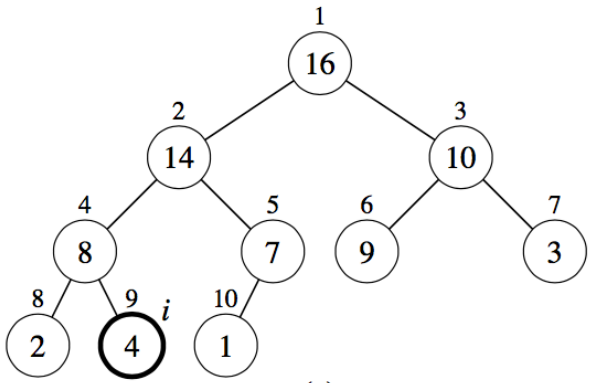
Example



(a)



(b)



(c)

- Node 2 violates the max-heap property.
- Compare node 2 with its children, and then swap it with the larger of the two children.
- Continue down the tree, swapping until the value is properly placed at the root of a subtree that is a max-heap. In this case, the max-heap is a leaf.

Time: $O(\lg n)$.

Builde a heap

BUILD-MAX-HEAP(A, n) Given an array with unsorted n elements
for $i \leftarrow \lfloor n/2 \rfloor$ **downto** 1
    do MAX-HEAPIFY(A, i, n)

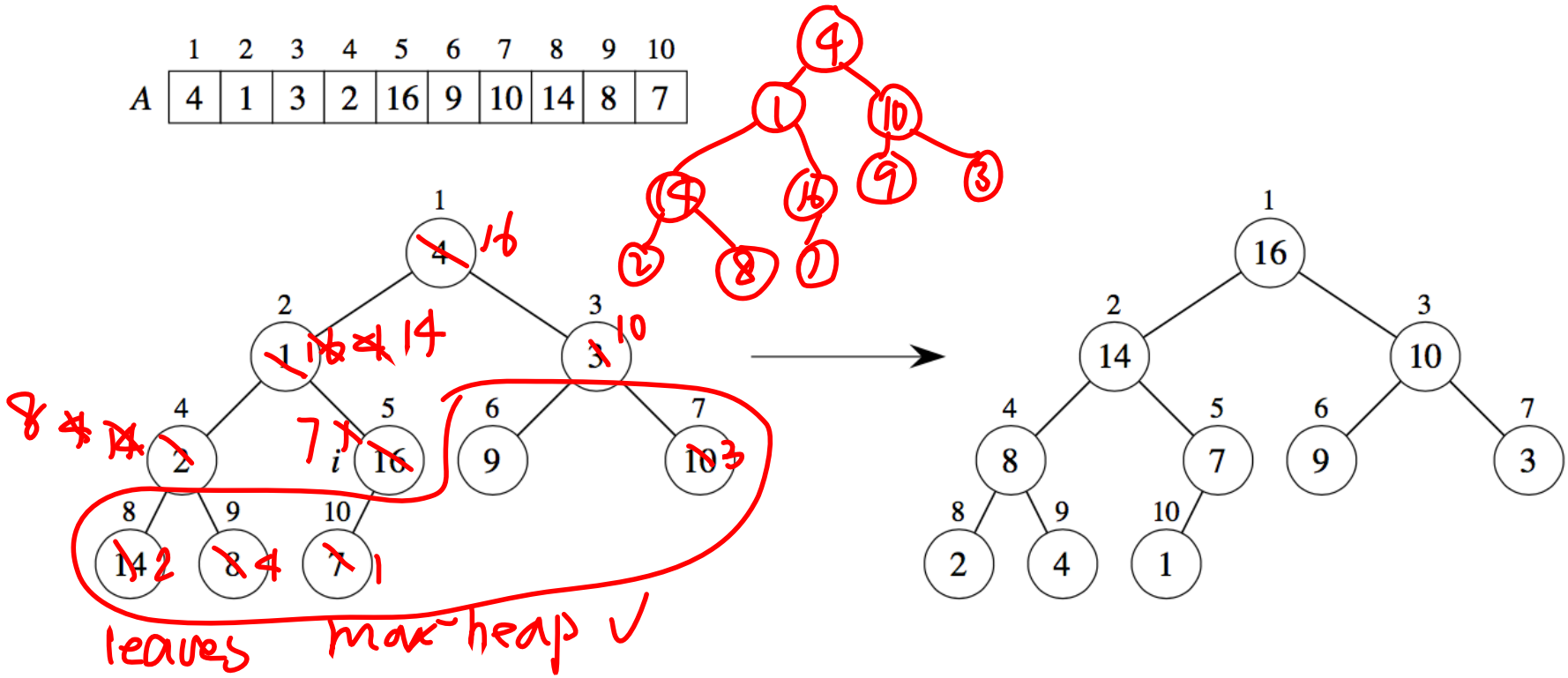
Why from $\lfloor n/2 \rfloor$ down to 1?
$A[i], i \geq \lfloor n/2 \rfloor + 1$ are leaves.

Build a heap

Example: Building a max-heap from the following unsorted array results in the first heap example.

- i starts off as 5.
- MAX-HEAPIFY is applied to subtrees rooted at nodes (in order): 16, 2, 3, 1, 4.

	1	2	3	4	5	6	7	8	9	10
A	4	1	3	2	16	9	10	14	8	7



Builde a heap

Correctness

Loop invariant: At start of every iteration of **for** loop, each node $i+1$, $i+2, \ldots, n$ is root of a max-heap.

Initialization: By Exercise 6.1-7, we know that each node $\lfloor n/2 \rfloor + 1$, $\lfloor n/2 \rfloor + 2$, $\ldots, n$ is a leaf, which is the root of a trivial max-heap. Since $i = \lfloor n/2 \rfloor$ before the first iteration of the **for** loop, the invariant is initially true.

Maintenance: Children of node i are indexed higher than i, so by the loop invariant, they are both roots of max-heaps. Correctly assuming that $i+1, i+2, \ldots, n$ are all roots of max-heaps, MAX-HEAPIFY makes node i a max-heap root. Decrementing i reestablishes the loop invariant at each iteration.

Termination: When $i = 0$, the loop terminates. By the loop invariant, each node, notably node 1, is the root of a max-heap.

Time analysis for building a heap

- ***Simple bound:*** $O(n)$ calls to MAX-HEAPIFY, each of which takes $O(\lg n)$ time $\Rightarrow O(n \lg n)$. (Note: A good approach to analysis in general is to start by proving easy bound, then try to tighten it.)
- ***Tighter analysis:*** Observation: Time to run MAX-HEAPIFY is linear in the height of the node it's run on, and most nodes have small heights. Have $\leq \lceil n/2^{h+1} \rceil$ nodes of height h (see Exercise 6.3-3), and height of heap is $\lfloor \lg n \rfloor$ (Exercise 6.1-2). $\lceil n/2 \rceil$ of height $=0$ (leaves)

  The time required by MAX-HEAPIFY when called on a node of height h is $O(h)$, so the total cost of BUILD-MAX-HEAP is

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) .$$

$h=0$ $n/2$ nodes
$h=1$ $n/2^2$
h $n/2^{h+1}$

$= O\left(\sum \frac{n}{2^{h+1}} \cdot c \cdot h\right)$ // $\hookrightarrow O(n)$

Time analysis for building a heap

$n \to \infty$ $\sum_{h=0}^{\lg n} \frac{h}{2^{h+1}} = \sum_{h=0}^{\infty} \left(\frac{1}{2}\right)^{h+1} \cdot h = \frac{1}{2} \sum_{h=0}^{\infty} \left(\frac{1}{2}\right)^{h} \cdot h = \frac{1}{2} \times \frac{\frac{1}{2}}{\left(1-\frac{1}{2}\right)^2} = 1$

$\left(\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2} \right) \sum_{k=0}^{\infty} x^k = \frac{1}{1-x} \Rightarrow \sum_{k=0}^{\infty} kx^{k-1} = \frac{1}{(1-x)^2}$

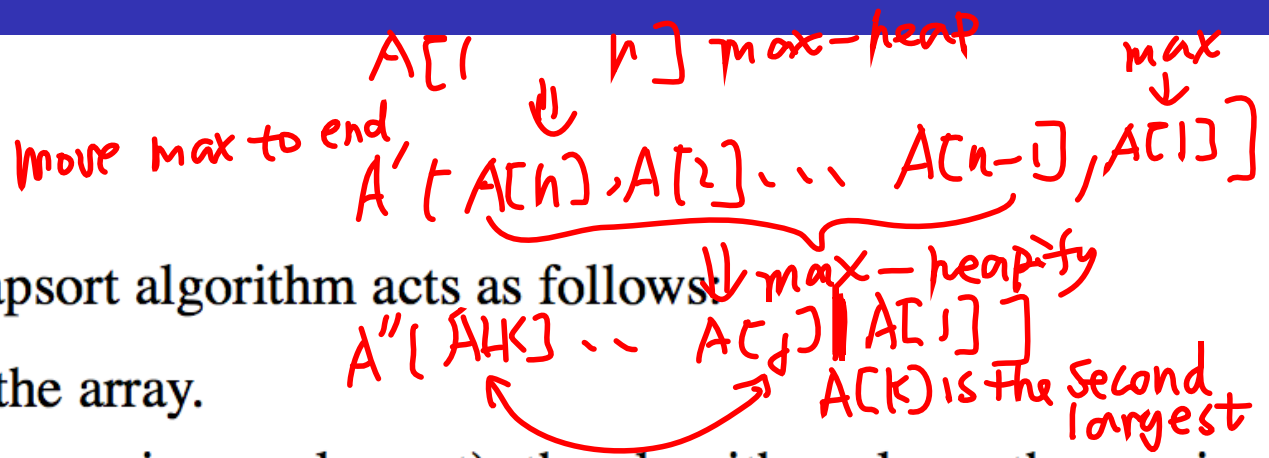
Evaluate the last summation by substituting $x = 1/2$ in the formula (A.8) ($\sum_{k=0}^{\infty} kx^k$), which yields

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2 .$$

Thus, the running time of BUILD-MAX-HEAP is $O(n)$.

Building a min-heap from an unordered array can be done by calling MIN-HEAPIFY instead of MAX-HEAPIFY, also taking linear time.

Heapsort



Given an input array, the heapsort algorithm acts as follows:

- Builds a max-heap from the array.
- Starting with the root (the maximum element), the algorithm places the maximum element into the correct place in the array by swapping it with the element in the last position in the array.
- "Discard" this last node (knowing that it is in its correct place) by decreasing the heap size, and calling MAX-HEAPIFY on the new (possibly incorrectly-placed) root.
- Repeat this "discarding" process until only one node (the smallest element) remains, and therefore is in the correct place in the array.

Heapsort

HEAPSORT(A, n) — final exam

BUILD-MAX-HEAP(A, n)

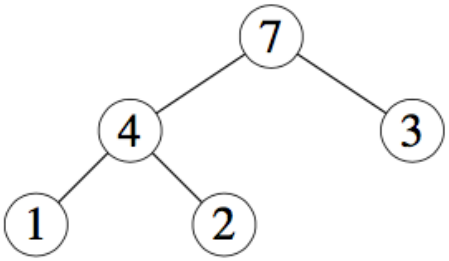
for $i \leftarrow n$ **downto** 2

  do exchange $A[1] \leftrightarrow A[i]$

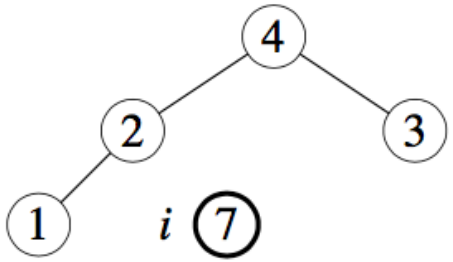
    MAX-HEAPIFY$(A, 1, i-1)$

min ?

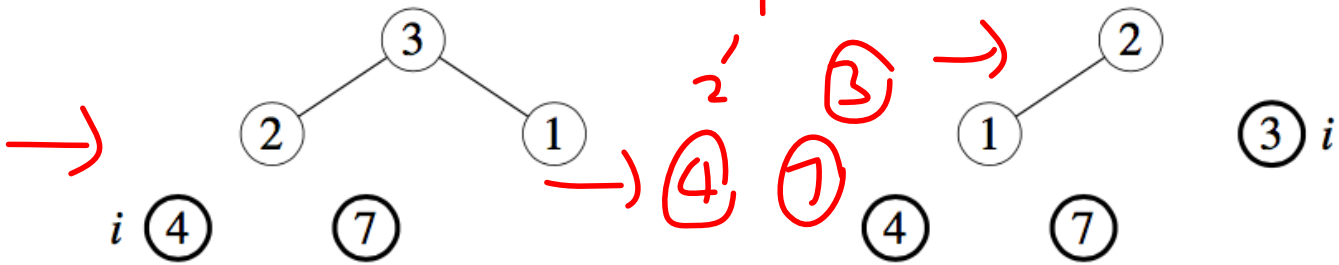
Heapsort example



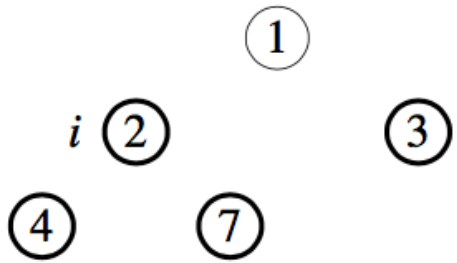
(a)



(b)



(c) (d)



(e)

A	1	2	3	4	7

Heapsort analysis

Analysis

- BUILD-MAX-HEAP: $O(n)$
- **for** loop: $n - 1$ times
- exchange elements: $O(1)$
- MAX-HEAPIFY: $O(\lg n)$

Total time: $O(n \lg n)$.

Heap implementation of priority queue

- Maintains a dynamic set S of elements.
- Each set element has a ***key***—an associated value.
- Max-priority queue supports dynamic-set operations:
  - INSERT(S, x): inserts element x into set S.
  - MAXIMUM(S): returns element of S with largest key.
  - EXTRACT-MAX(S): removes and returns element of S with largest key.
  - INCREASE-KEY(S, x, k): increases value of element x's key to k. Assume $k \geq x$'s current key value.
- Example max-priority queue application: schedule jobs on shared computer.

Heap implementation of priority queue

- Min-priority queue supports similar operations:
  - INSERT(S, x): inserts element x into set S.
  - MINIMUM(S): returns element of S with smallest key.
  - EXTRACT-MIN(S): removes and returns element of S with smallest key.
  - DECREASE-KEY(S, x, k): decreases value of element x's key to k. Assume $k \leq x$'s current key value.
- Example min-priority queue application: event-driven simulator.

Find the max

Finding the maximum element

Getting the maximum element is easy: it's the root.

HEAP-MAXIMUM(A)
return $A[1]$

Time: $\Theta(1)$.

Extract the max

Given the array A:

- Make sure heap is not empty.
- Make a copy of the maximum element (the root).
- Make the last node in the tree the new root.
- Re-heapify the heap, with one fewer node.
- Return the copy of the maximum element.

HEAP-EXTRACT-MAX(A, n)
if $n < 1$
  then error "heap underflow"
$max \leftarrow A[1]$
$A[1] \leftarrow A[n]$
MAX-HEAPIFY$(A, 1, n-1)$ ▷ remakes heap
return max

assumption: children are heapified

Analysis of extracting max

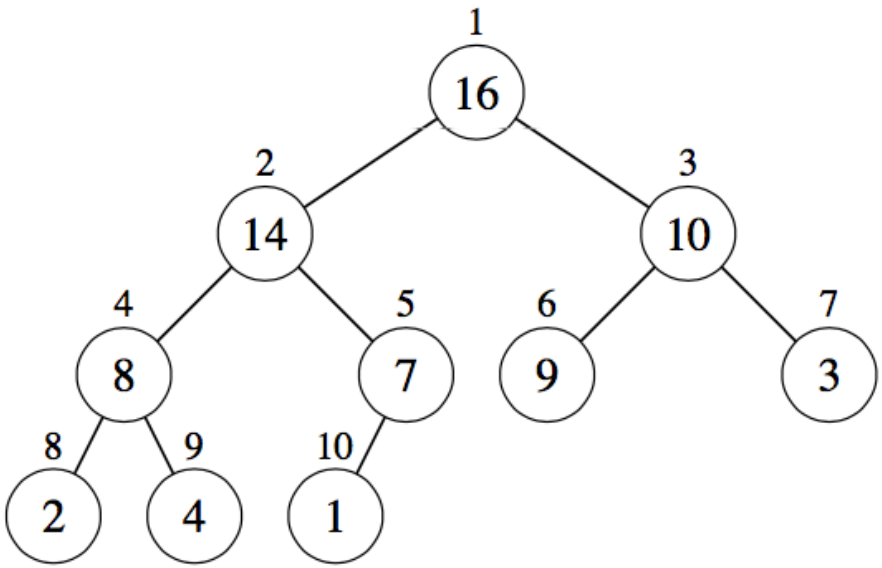
Analysis: constant time assignments plus time for MAX-HEAPIFY.

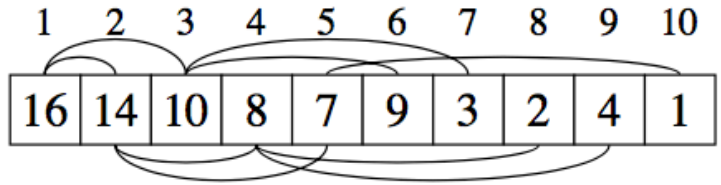
Time: $O(\lg n)$.

Example: Run HEAP-EXTRACT-MAX on first heap example.

- Take 16 out of node 1.
- Move 1 from node 10 to node 1. keep structure of other elements unchanged
- Erase node 10.
- MAX-HEAPIFY from the root to preserve max-heap property.
- Note that successive extractions will remove items in reverse sorted order.

Example





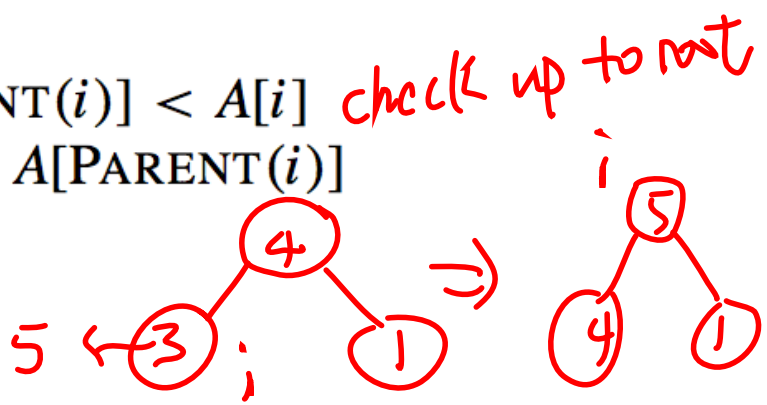
Inscrease key

Given set S, element x, and new key value k:

- Make sure $k \geq x$'s current key.
- Update x's key value to k.
- Traverse the tree upward comparing x to its parent and swapping keys if necessary, until x's key is smaller than its parent's key.

HEAP-INCREASE-KEY(A, i, key)

if $key < A[i]$
  then error "new key is smaller than current key"
$A[i] \leftarrow key$
while $i > 1$ and $A[\text{PARENT}(i)] < A[i]$
  do exchange $A[i] \leftrightarrow A[\text{PARENT}(i)]$
    $i \leftarrow \text{PARENT}(i)$



Heaps

Analysis: Upward path from node i has length $O(\lg n)$ in an n-element heap.

Time: $O(\lg n)$.

Example: Increase key of node 9 in first heap example to have value 15. Exchange keys of nodes 4 and 9, then of nodes 2 and 4.

Heaps: insertion

Given a key k to insert into the heap:

- Insert a new node in the very last position in the tree with key $-\infty$.
- Increase the $-\infty$ key to k using the HEAP-INCREASE-KEY procedure defined above.

MAX-HEAP-INSERT(A, key, n)
$A[n+1] \leftarrow -\infty$ the smallest
HEAP-INCREASE-KEY$(A, n+1, key)$

Analysis: constant time assignments + time for HEAP-INCREASE-KEY.

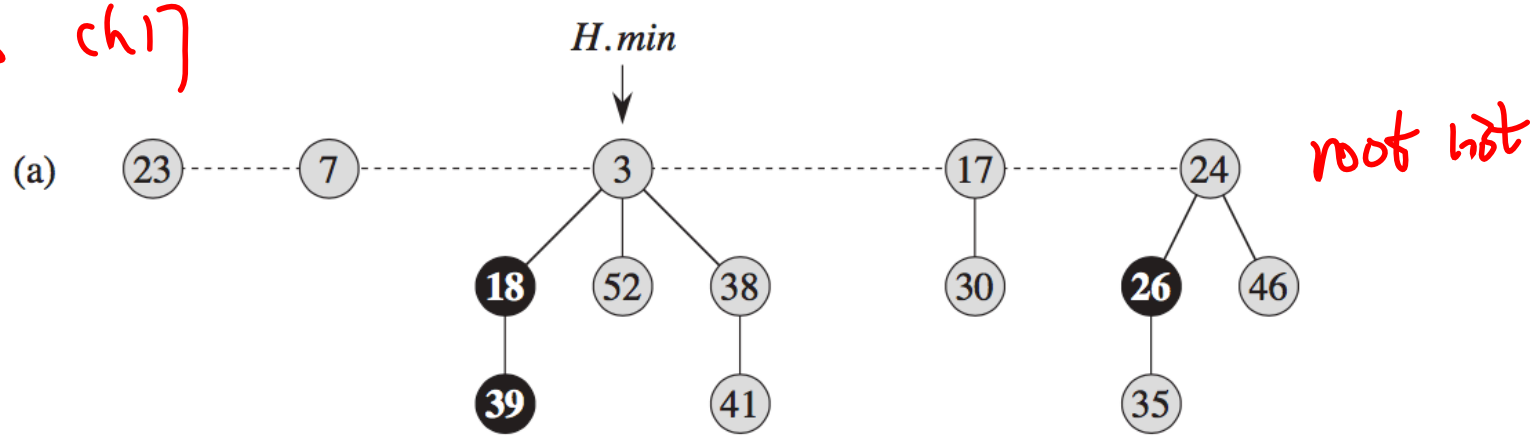
Time: $O(\lg n)$.

Min-priority queue operations are implemented similarly with min-heaps.

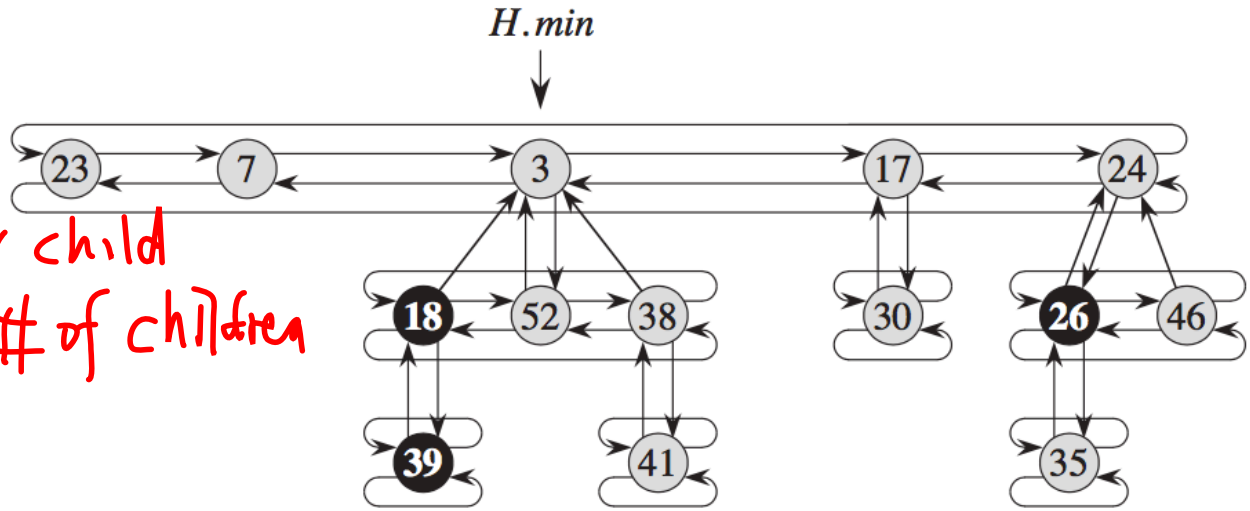
Fibonacci Heaps: collection of min-heap ordered trees

root list and children of a node are doubly circular linked lists,
attributes in a node x: $x.p$, $x.child$, $x.left$, $x.right$, $x.degree$, $x.mark$
minimum of the heap: $H.min$

Analysis ch17



root list



root p=NIL

x.child → any child

x.degree = # of children

Fibonacci Heaps: insertion, $O(1)$

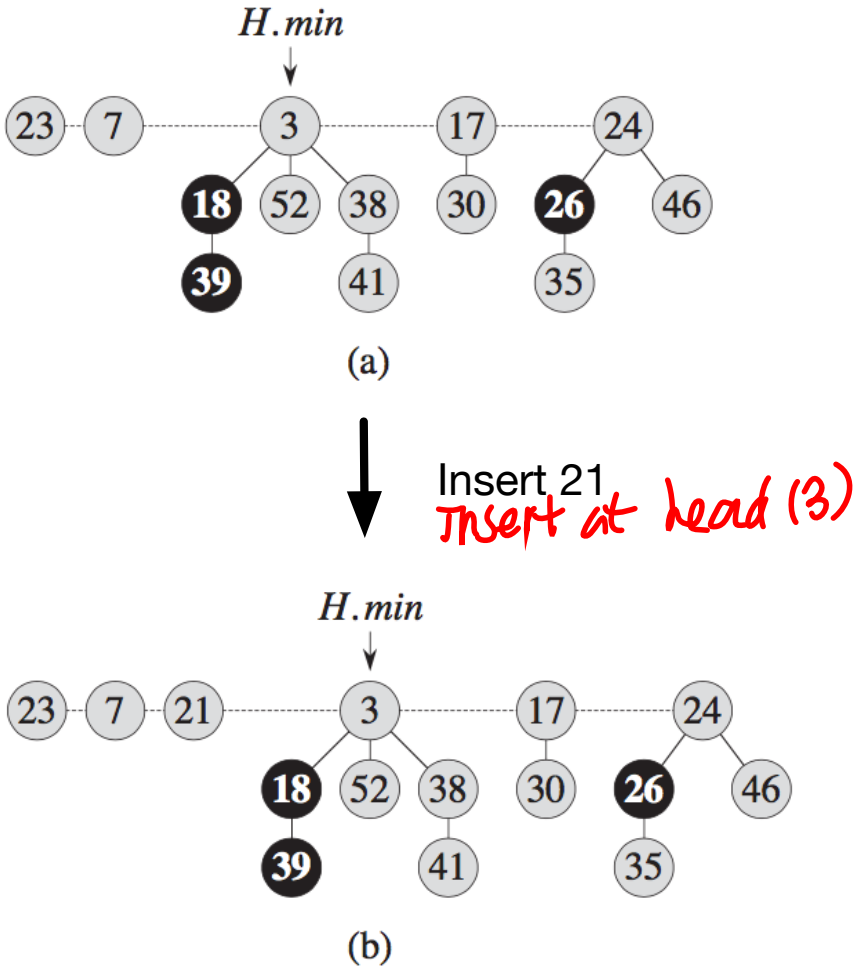
FIB-HEAP-INSERT(H, x)

```
1   x.degree = 0
2   x.p = NIL
3   x.child = NIL
4   x.mark = FALSE
5   if H.min == NIL
6       create a root list for H containing just x
7       H.min = x
8   else insert x into H's root list
9       if x.key < H.min.key
10          H.min = x
11  H.n = H.n + 1
```

} insert into root list

empty

Insertion demo



Fibonacci Heaps: union, $O(1)$

FIB-HEAP-UNION(H_1, H_2)

1 H = MAKE-FIB-HEAP()
2 $H.min = H_1.min$
3 concatenate the root list of H_2 with the root list of H
4 **if** ($H_1.min$ == NIL) or ($H_2.min \neq$ NIL and $H_2.min.key < H_1.min.key$)
5 $H.min = H_2.min$
6 $H.n = H_1.n + H_2.n$
7 **return** H

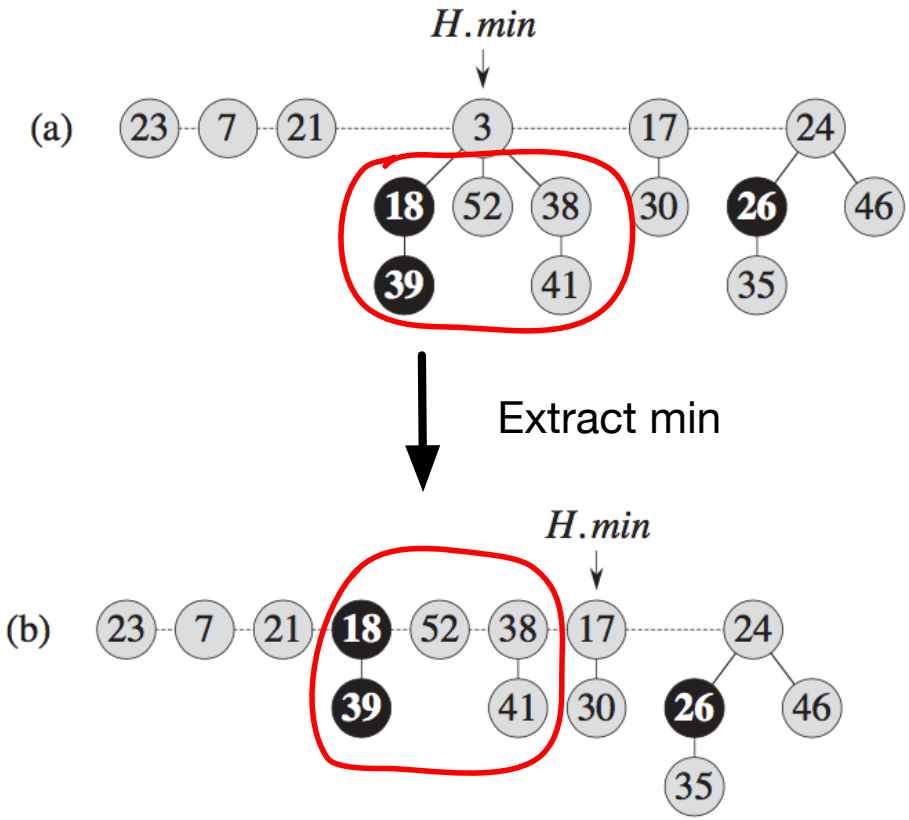
Fibonacci Heaps: exact min, $O(\lg n)$

FIB-HEAP-EXTRACT-MIN(H)

```
1   z = H.min
2   if z ≠ NIL
3       for each child x of z
4           add x to the root list of H
5           x.p = NIL
6       remove z from the root list of H
7       if z == z.right
8           H.min = NIL
9       else H.min = z.right
10          CONSOLIDATE(H)
11      H.n = H.n − 1
12  return z
```

z = nil → empty

Extraction demo



Consolidation

The next step, in which we reduce the number of trees in the Fibonacci heap, is ***consolidating*** the root list of H, which the call CONSOLIDATE(H) accomplishes. Consolidating the root list consists of repeatedly executing the following steps until every root in the root list has a distinct *degree* value:

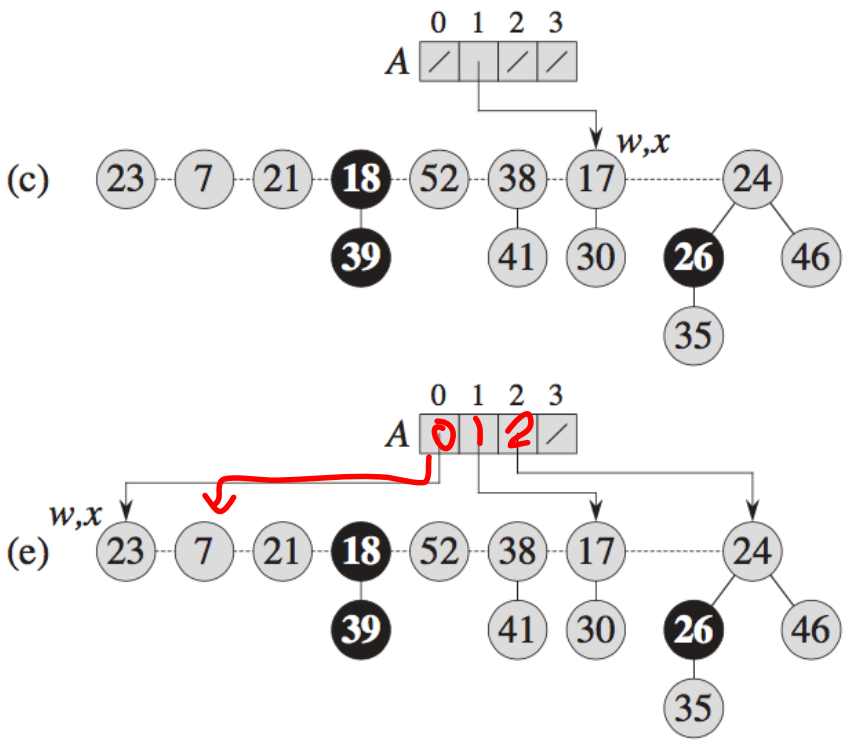
1. Find two roots x and y in the root list with the same degree. Without loss of generality, let $x.key \leq y.key$.
2. ***Link*** y to x: remove y from the root list, and make y a child of x by calling the FIB-HEAP-LINK procedure. This procedure increments the attribute $x.degree$ and clears the mark on y.

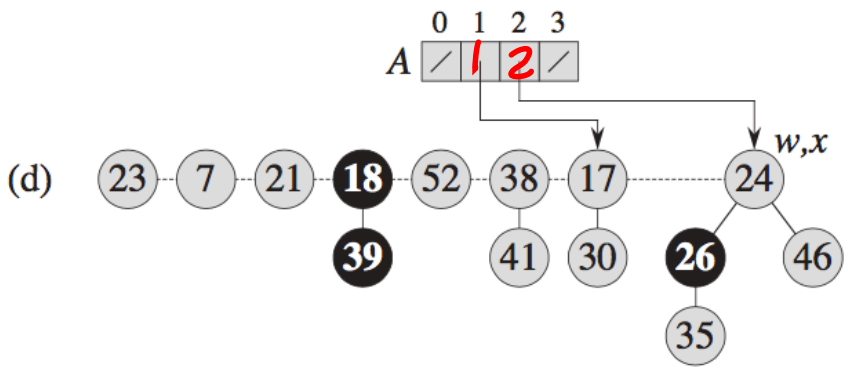
Consolidation

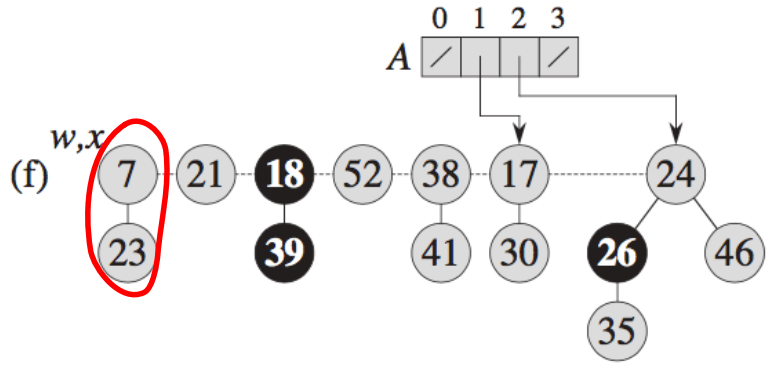
```
CONSOLIDATE(H)
 1  let A[0..D(H.n)] be a new array
 2  for i = 0 to D(H.n)
 3      A[i] = NIL
 4  for each node w in the root list of H
 5      x = w
 6      d = x.degree
 7      while A[d] ≠ NIL
 8          y = A[d]        // another node with the same degree as x
 9          if x.key > y.key
10              exchange x with y
11          FIB-HEAP-LINK(H, y, x)
12          A[d] = NIL
13          d = d + 1
14      A[d] = x
15  H.min = NIL
16  for i = 0 to D(H.n)
17      if A[i] ≠ NIL
18          if H.min == NIL
19              create a root list for H containing just A[i]
20              H.min = A[i]
21          else insert A[i] into H's root list
22              if A[i].key < H.min.key
23                  H.min = 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 x.degree
3  y.mark = FALSE
```

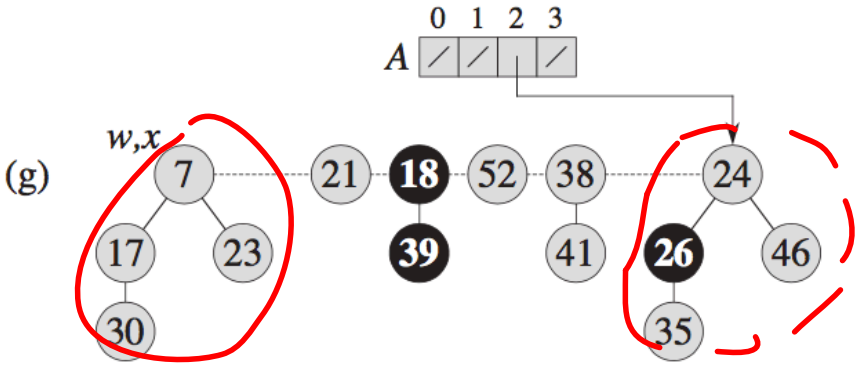
Consolidation demo: start with $H.min$

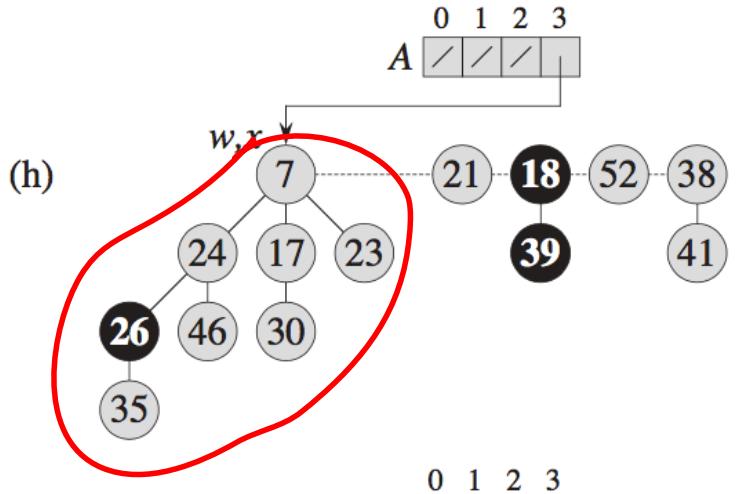


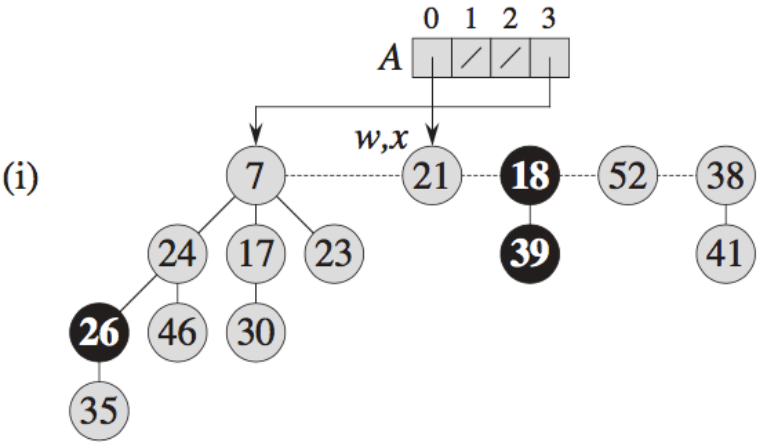


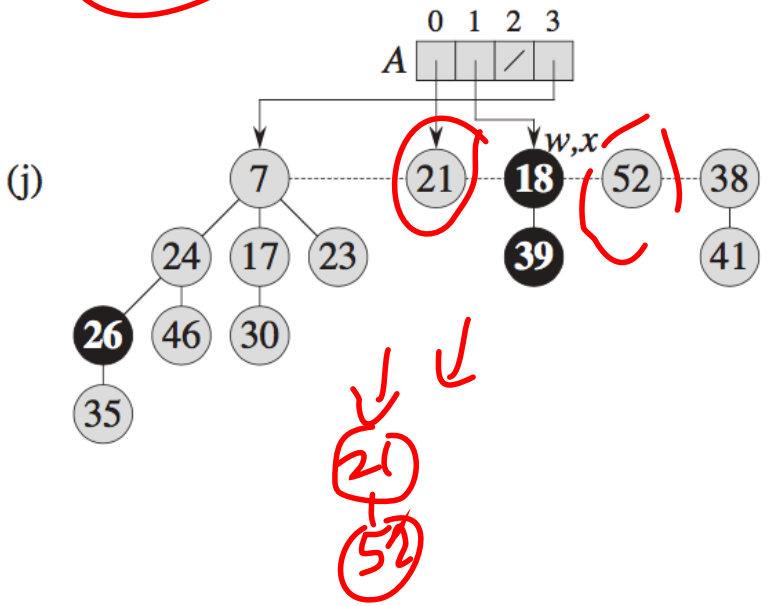


Consolidation demo

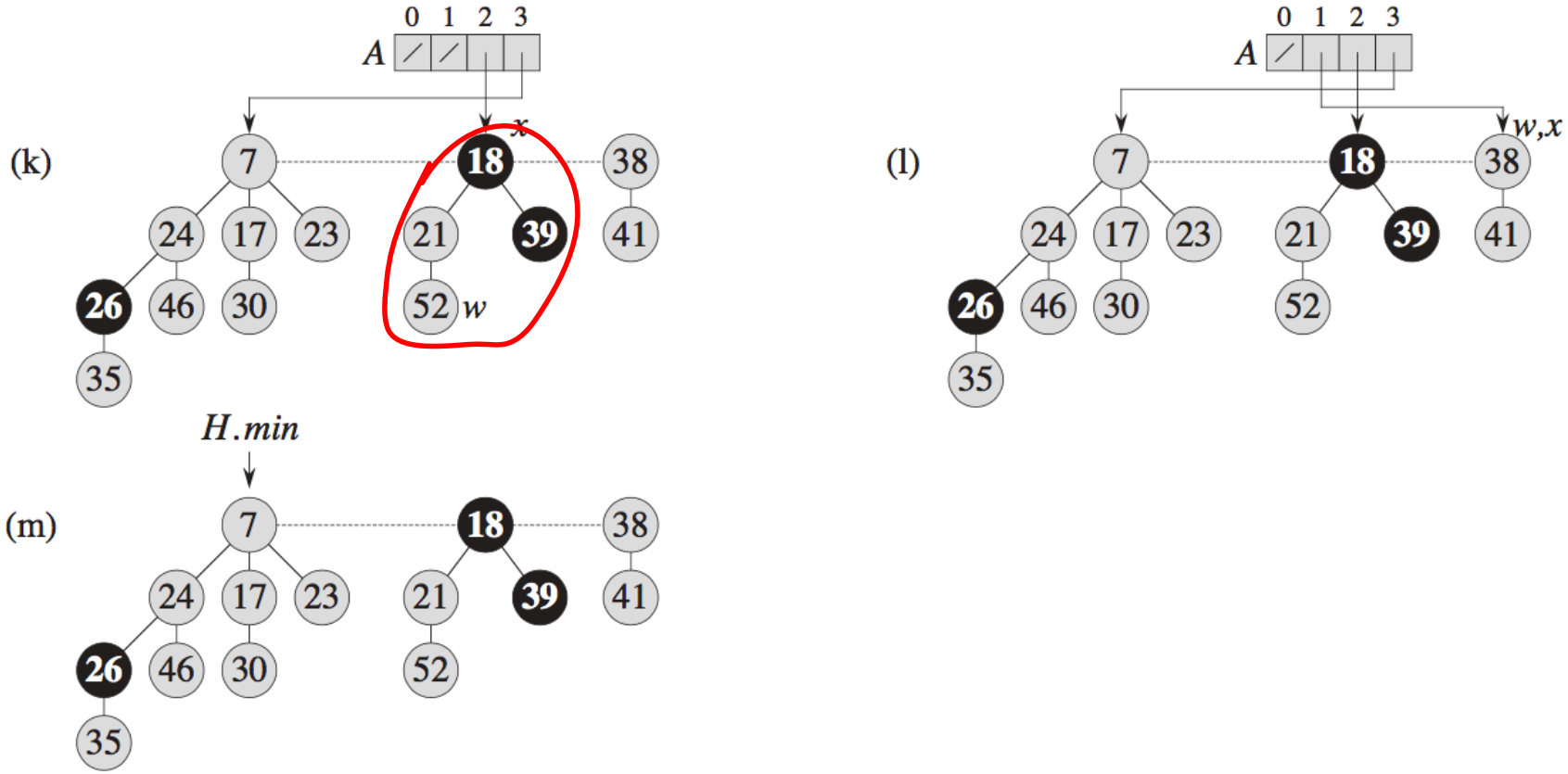








Consolidation demo

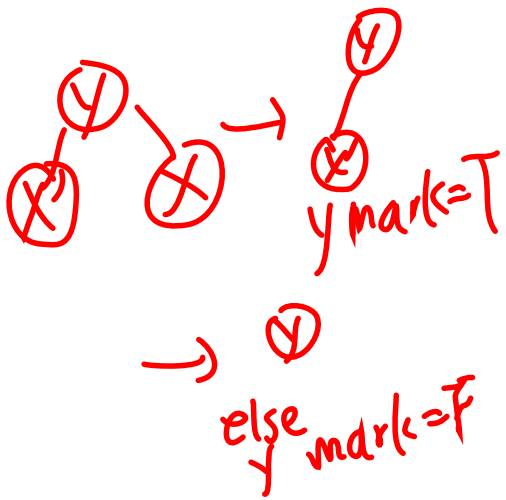


Fibonacci Heaps: decrease key

```
FIB-HEAP-DECREASE-KEY(H, x, k)
1  if k > x.key
2      error "new key is greater than current key"
3  x.key = k
4  y = x.p
5  if y ≠ NIL and x.key < y.key
6      CUT(H, x, y)
7      CASCADING-CUT(H, y)
8  if x.key < H.min.key
9      H.min = x
```

```
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 ≠ NIL
3      if y.mark == FALSE
4          y.mark = TRUE
5      else CUT(H, y, z)
6          CASCADING-CUT(H, z)
```

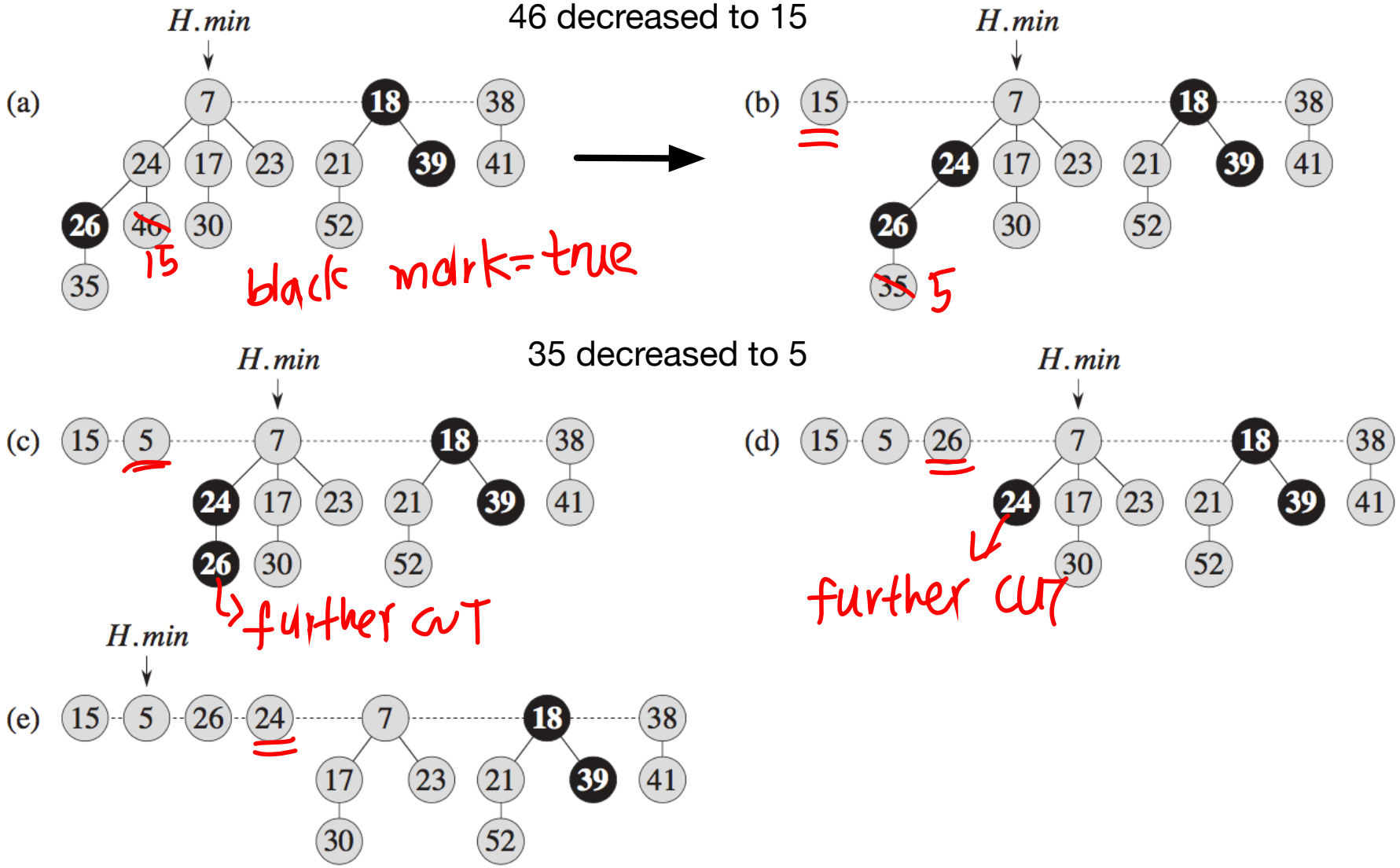


is x.key > y key
fine

x is in root list
(肯定false)

mark 用来计算 running time
每两次

Decrease key demo



Delete a node

FIB-HEAP-DELETE(H, x)

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

Self study after class: amortized time analysis in Chapter 17, then, the time analysis for Fibonacci Heaps in Chapter 19.

Homework 4

1. Exercise 6.4-1
2. Exercise 6.5-1
3. Exercise 6.5-2
4. Problem 6-1
5. Exercise 19.2-1

Due date: Nov 16, Thu