Prepared by: [Arun Konagurthu]

FIT3155: Advanced Algorithms and Data Structures

Week 6: **Fibonacci heaps**

Faculty of Information Technology, Monash University

What is covered in this lecture?

Fibonacci heaps

Original reference

Michael Fredman and Robert Tarjan, Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms
Journal of the ACM, 34(3) 596-615 (1987). [link]

Source material and recommended reading

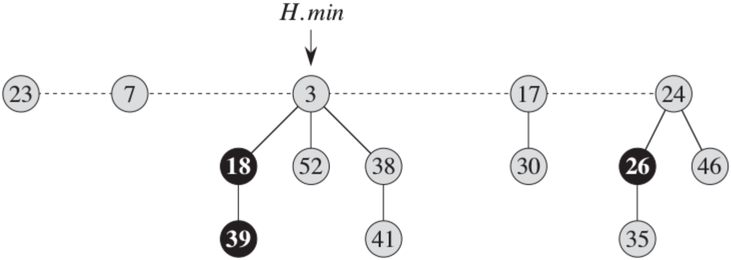
- CLRS, Introduction to Algorithms (Chapter 19):
  Fibonacci heaps [online link]

Motivation for Fibonacci heaps

- Improve run-time complexity of **`Dijkstra's`** shortest path algorithm
  - Recall this from FIT2004?
- Similar to a **`binomial heap`**, a **`Fibonacci heap`** maintains a collection of (min-heap ordered) trees, however..
  - ...the trees in the collection are **`less stringent`** in their definitions, and..
    - ...while in a **`binomial heap`** merging/consolidation of trees is performed eagerly after each **`extract-min`** or **`insert`** operation...
    - ...in a **`Fibonacci heap`** the consolidation/merging is performed lazily, by deferring until **`extract-min`** operation is next invoked.

Example of a Fibonacci heap

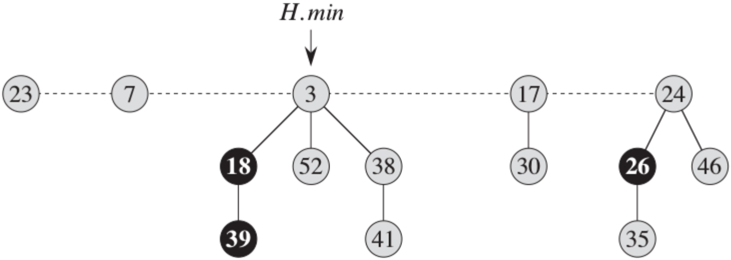
- A Fibonacci heap H containing 5 trees, with total 14 elements.



- $H.min$ is a pointer to root node (of a tree in the collection) with the minimum element.
- In a **Fibonacci heap**, each node/element is:
  - either **marked** (shown above as **black coloured nodes**)...
  - ...or **unmarked**/regular (shown as the grey coloured nodes above)
  - We will examine in later slides what this 'marking' means/does.

Example of a Fibonacci heap

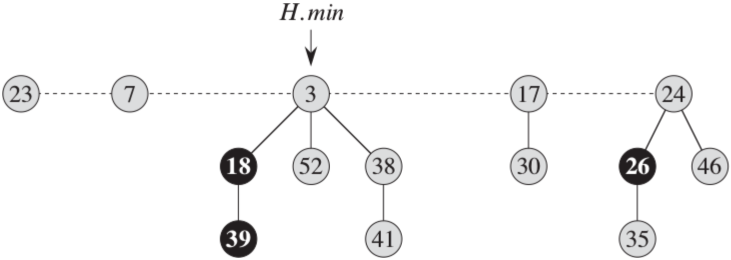
- A Fibonacci heap H containing 5 trees, with total 14 elements.



- $H.min$ is a pointer to root node (of a tree in the collection) with the minimum element.
- In a **Fibonacci heap**, each node/element is:
  - either **marked** (shown above as **black coloured nodes**)...
  - ...or **unmarked**/regular (shown as the grey coloured nodes above)
  - We will examine in later slides what this 'marking' means/does.

Example of a Fibonacci heap

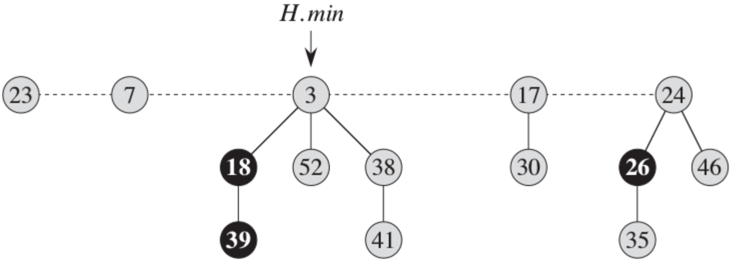
- A Fibonacci heap H containing 5 trees, with total 14 elements.



- $H.min$ is a pointer to root node (of a tree in the collection) with the minimum element.
- In a **Fibonacci heap**, each node/element is:
  - either **marked** (shown above as **black coloured nodes**)...
  - ...or **unmarked**/regular (shown as the grey coloured nodes above)
  - We will examine in later slides what this 'marking' means/does.

Example of a Fibonacci heap

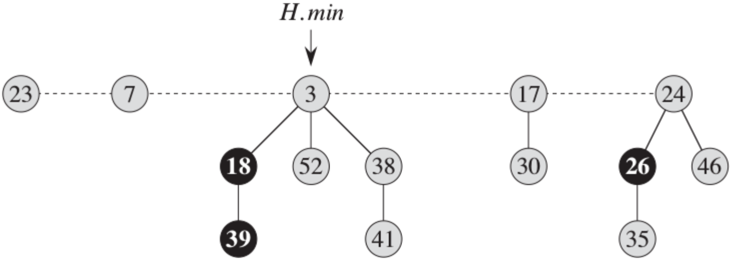
- A Fibonacci heap H containing 5 trees, with total 14 elements.



- $H.min$ is a pointer to root node (of a tree in the collection) with the minimum element.
- In a **Fibonacci heap**, each node/element is:
  - either **marked** (shown above as **black coloured nodes**)...
  - ...or **unmarked**/regular (shown as the grey coloured nodes above)
  - We will examine in later slides what this 'marking' means/does.

Example of a Fibonacci heap

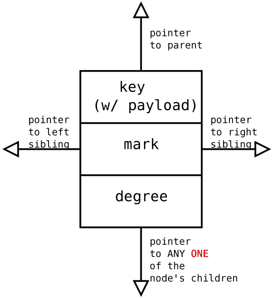
- A Fibonacci heap H containing 5 trees, with total 14 elements.



- $H.min$ is a pointer to root node (of a tree in the collection) with the minimum element.
- In a **Fibonacci heap**, each node/element is:
  - either **marked** (shown above as **black coloured nodes**)...
  - ...or **unmarked**/regular (shown as the grey coloured nodes above)
  - We will examine in later slides what this 'marking' means/does.

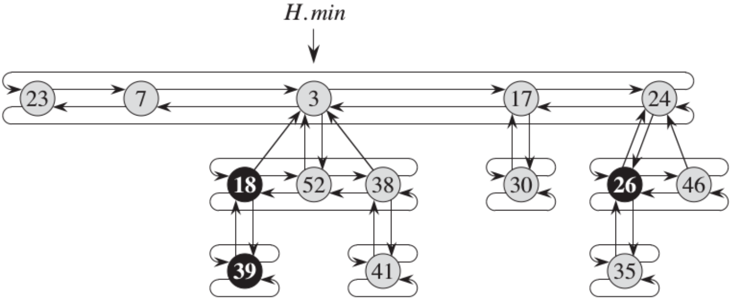
Representation of a node in a Fibonacci heap

- Each node x:
  - stores a **key**, and has associated **payload** information;
  - stores the number of its children: $x.degree$;
  - stores if the node is marked or not: $x.mark$;
  - has a pointer $x.parent$ to its parent node;
  - has a pointer $x.child$ to ANY ONE of its children.
  - x and its siblings form a **circular doubly linked list**:
    - ★ So, x a pointer to its left sibling $x.left$...
    - ★ ...and its right sibling $x.right$.



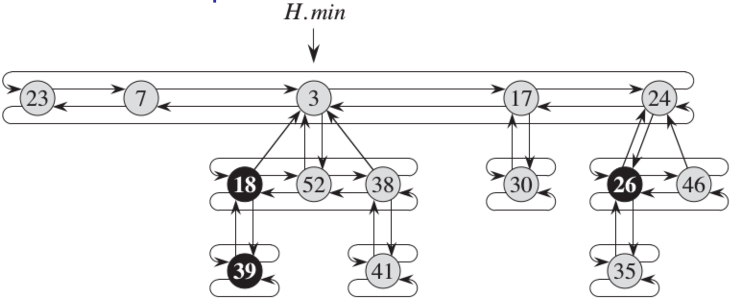
Fibonacci heaps are represented using circular doubly linked lists

- Below is a visualization of circular doubly linked list representation (and other pointers) for the example Fibonacci heap shown in slide # 6.



- This has several advantages:
  - This allows **insert** operations into any location in $O(1)$ time.
  - This allows **delete** operations from any location in $O(1)$ time.
  - This allows joining elements in one list to another in $O(1)$ time.

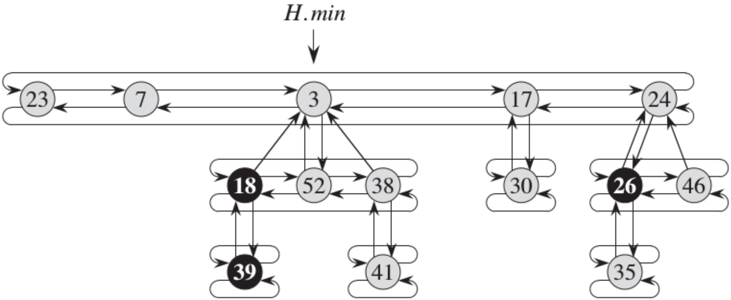
Fibonacci heaps – Attributes



Associated with each node/element x in a Fibonacci heap H, is:

- the **number of children** in the child list:
  - we will call the $degree$ of a node ($x.degree$).
  - Eg: 24 has $degree$=2. 7 has $degree$=0.
- whether a node is marked or not – $x.mark$
  - It will become clear in **decrease-key** operation what this means...
  - .. but quickly, '**marked**' implies the node has lost a child; **unmarked**' implies it hasn't lost a child. Details when slides #26-29 are covered.
  - Eg: 18 is '**marked**'. 30 is '**unmarked**'.

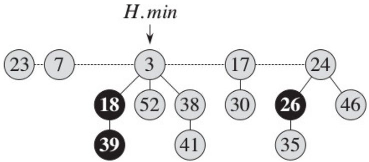
Fibonacci heaps – Attributes (continued)



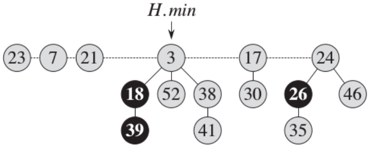
- Access to the Fibonacci heap H is via the pointer to the **minimum** (key) node in the entire heap, denoted by $H.min$.
- Roots of all trees in the Fibonacci heap are connected by a **root_list**,
- ...where each tree's root can be accessed via $left$ and $right$ pointers, starting from $H.min$.

Fibonacci heaps – **insert** operation

insert(x) into a Fibonacci heap H. (In this example, $x =$ 21.)



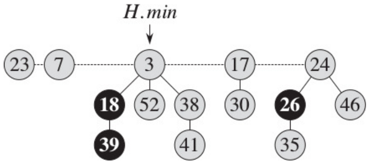
↓ **insert** 21



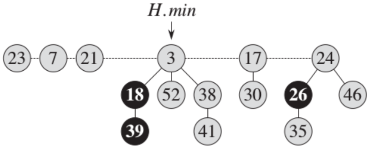
- Access H via the pointer $H.min$.
- **insert** x into the **root_list**, making it the left sibling of $H.min$ element/root.
- If x < $H.min$ (i.e., comparing their respective priorities/keys), update $H.min$ to point to x in the root level.
- Clearly, **insert**(x) is $O(1)$-time operation.

Fibonacci heaps – **insert** operation

insert(x) into a Fibonacci heap H. (In this example, $x =$ 21.)



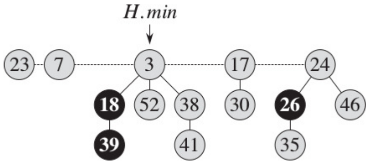
↓ **insert** 21



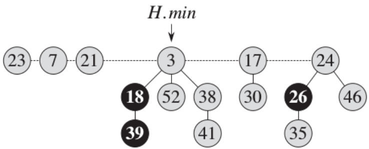
- Access H via the pointer $H.min$.
- **insert** x into the **root_list**, making it the left sibling of $H.min$ element/root.
- If x < $H.min$ (i.e., comparing their respective priorities/keys), update $H.min$ to point to x in the root level.
- Clearly, **insert**(x) is $O(1)$-time operation.

Fibonacci heaps – insert operation

insert(x) into a Fibonacci heap H. (In this example, $x =$ 21.)



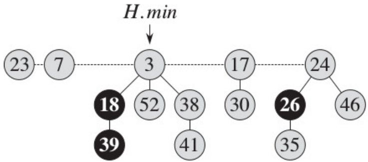
↓ **insert** 21



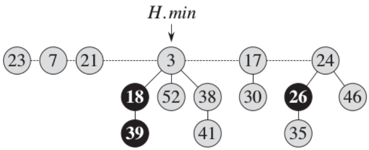
- Access H via the pointer $H.min$.
- **insert** x into the **root_list**, making it the left sibling of $H.min$ element/root.
- If x $< H.min$ (i.e., comparing their respective priorities/keys), update $H.min$ to point to x in the root level.
- Clearly, **insert**(x) is $O(1)$-time operation.

Fibonacci heaps – **insert** operation

insert(x) into a Fibonacci heap H. (In this example, $x = 21$.)



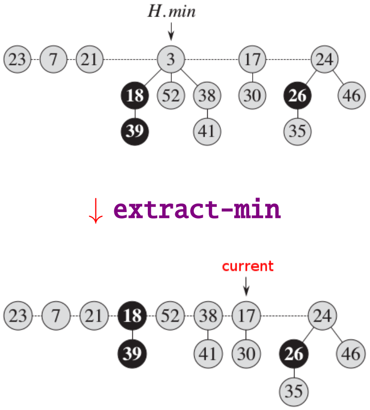
↓ **insert** 21



- Access H via the pointer $H.min$.
- **insert** x into the **root_list**, making it the left sibling of $H.min$ element/root.
- If $x < H.min$ (i.e., comparing their respective priorities/keys), update $H.min$ to point to x in the root level.
- Clearly, **insert**(x) is $O(1)$-time operation.

Fibonacci heaps – extract-min operation

Identify and delete the minimum (key) node in the heap



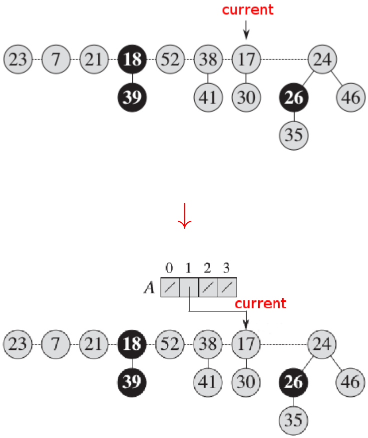
- Identify minimum element via the pointer $H.min$.
- To extract (and delete) minimum element (which is 3 in this running example),...
- ...set the current pointer to the $right$ sibling of $H.min$.
- ...promote/add all children (subtrees) to the root list, and
- IMPORTANT: Now run **consolidate** (or merge) operation. See next slides 13-18

NOTE!!!

consolidate operation ensures that no two nodes in the root level have the same degree (i.e., number of children).

Fibonacci heaps – **consolidate** operation

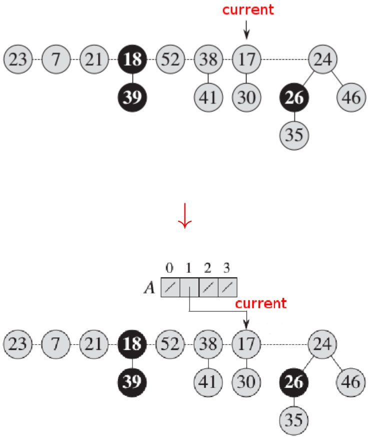
Fibonacci heaps run a **consolidate** operation only after a call to **extract-min**.



- Starting from current which is pointing to 17...
- Maintain an auxiliary array A to keep track of the root nodes indexed by their *degrees* (i.e., number of children). Initially, A is empty.
- Since the root node at current=17 has $degree = 1$...
- ...and $A[1]$ slot is empty, so...
- ...get $A[1]$ to point to the root node at current=17.
- Next, move current to the right sibling, i.e. current=24

Fibonacci heaps – consolidate operation

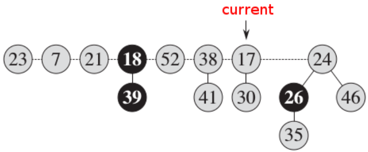
Fibonacci heaps run a **consolidate** operation only after a call to **extract-min**.

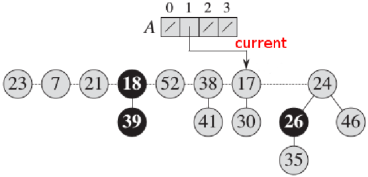


- Starting from current which is pointing to 17...
- Maintain an auxiliary array A to keep track of the root nodes indexed by their *degrees* (i.e., number of children). Initially, A is empty.
- Since the root node at current=17 has $degree = 1$...
- ...and $A[1]$ slot is empty, so...
- ...get $A[1]$ to point to the root node at current=17.
- Next, move current to the right sibling, i.e. current=24

Fibonacci heaps – consolidate operation

Fibonacci heaps run a **consolidate** operation only after a call to **extract-min**.

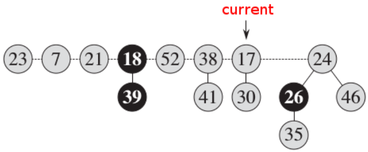


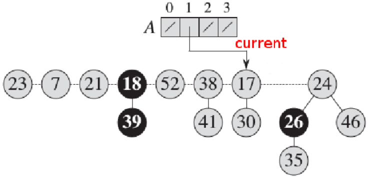


- Starting from current which is pointing to 17...
- Maintain an auxiliary array A to keep track of the root nodes indexed by their $degrees$ (i.e., number of children). Initially, A is empty.
- Since the root node at current=17 has $degree = 1$...
- ...and $A[1]$ slot is empty, so...
- ...get $A[1]$ to point to the root node at current=17.
- Next, move current to the right sibling, i.e. current=24

Fibonacci heaps – consolidate operation

Fibonacci heaps run a **consolidate** operation only after a call to **extract-min**.

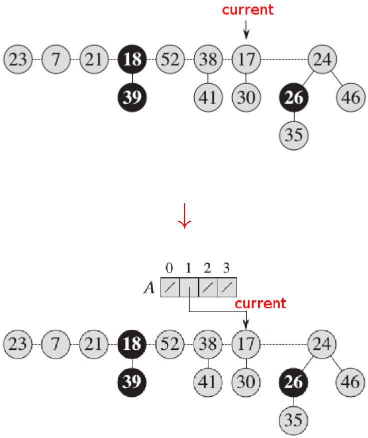




- Starting from current which is pointing to 17...
- Maintain an auxiliary array A to keep track of the root nodes indexed by their $degrees$ (i.e., number of children). Initially, A is empty.
- Since the root node at current= 17 has $degree = 1$...
- ...and $A[1]$ slot is empty, so...
- ...get $A[1]$ to point to the root node at current= 17.
- Next, move current to the right sibling, i.e. current= 24

Fibonacci heaps – consolidate operation

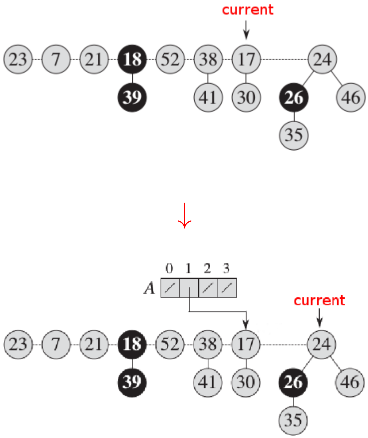
Fibonacci heaps run a **consolidate** operation only after a call to **extract-min**.



- Starting from current which is pointing to 17...
- Maintain an auxiliary array A to keep track of the root nodes indexed by their *degrees* (i.e., number of children). Initially, A is empty.
- Since the root node at current= 17 has $degree = 1$...
- ...and $A[1]$ slot is empty, so...
- ...get $A[1]$ to point to the root node at current= 17.
- Next, move current to the right sibling, i.e. current= 24

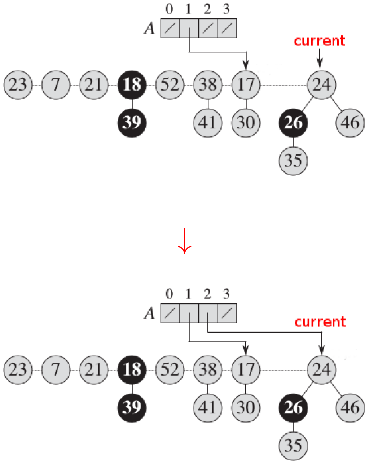
Fibonacci heaps – **consolidate** operation

Fibonacci heaps run a **consolidate** operation only after a call to **extract-min**.



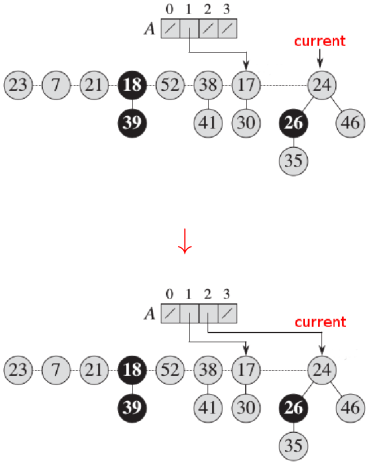
- Starting from current which is pointing to 17...
- Maintain an auxiliary array A to keep track of the root nodes indexed by their $degrees$ (i.e., number of children). Initially, A is empty.
- Since the root node at current=17 has $degree = 1$...
- ...and $A[1]$ slot is empty, so...
- ...get $A[1]$ to point to the root node at current=17.
- Next, move current to the right sibling, i.e. current=24

Fibonacci heaps – **consolidate** operation ...continued



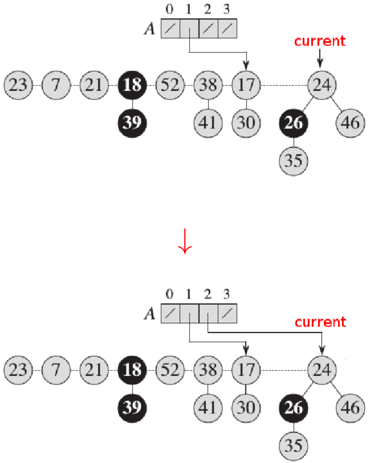
- Now, current=24 has *degree*=2.
- Again, $A[2]$ is empty, so...
- ...get $A[2]$ to point to the root node at current =24.
- Next, move current to the right sibling, i.e. move current from 24 to 23. Why?
  - root_list is a circular doubly linked list...
  - ...so the right sibling of 24 is (circularly) 23.
  - therefore, current=23.

Fibonacci heaps – **consolidate** operation ...continued



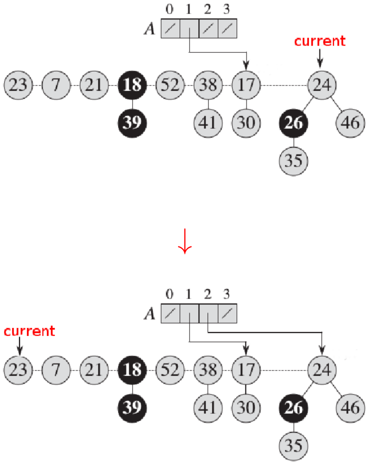
- Now, current=24 has *degree*=2.
- Again, $A[2]$ is empty, so...
- ...get $A[2]$ to point to the root node at current =24.
- Next, move current to the right sibling, i.e. move current from 24 to 23. Why?
  - `root_list` is a circular doubly linked list...
  - ...so the right sibling of 24 is (circularly) 23.
  - therefore, current=23.

Fibonacci heaps – **consolidate** operation ...continued



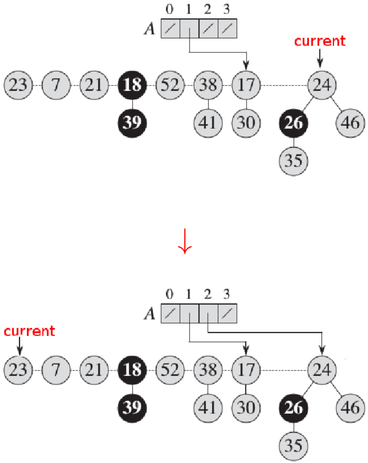
- Now, current= 24 has $degree$=2.
- Again, $A[2]$ is empty, so...
- ...get $A[2]$ to point to the root node at current = 24.
- Next, move current to the right sibling, i.e. move current from 24 to 23. Why?
  - `root_list` is a circular doubly linked list...
  - ...so the right sibling of 24 is (circularly) 23.
  - therefore, current= 23.

Fibonacci heaps – **consolidate** operation ...continued



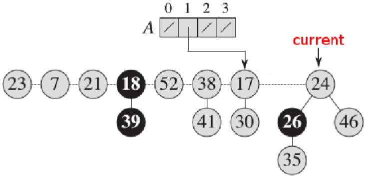
- Now, current=24 has $degree$=2.
- Again, $A[2]$ is empty, so...
- ...get $A[2]$ to point to the root node at current =24.
- Next, move current to the right sibling, i.e. move current from 24 to 23. Why?
  - **root_list** is a circular doubly linked list...
  - ...so the right sibling of 24 is (circularly) 23.
  - therefore, current=23.

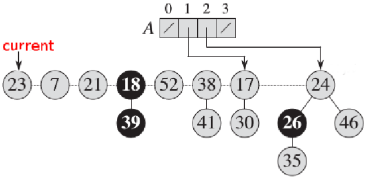
Fibonacci heaps – **consolidate** operation ...continued



- Now, current=24 has $degree$=2.
- Again, $A[2]$ is empty, so...
- ...get $A[2]$ to point to the root node at current =24.
- Next, move current to the right sibling, i.e. move current from 24 to 23. Why?
  - **root_list** is a circular doubly linked list...
  - ...so the right sibling of 24 is (circularly) 23.
  - therefore, current=23.

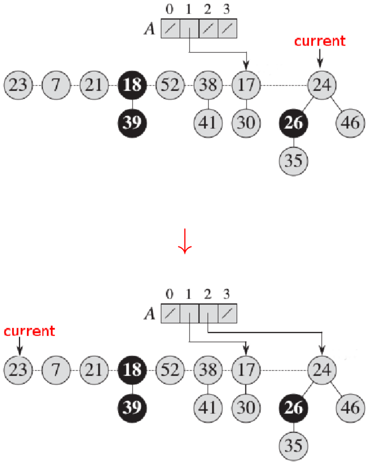
Fibonacci heaps – **consolidate** operation ...continued





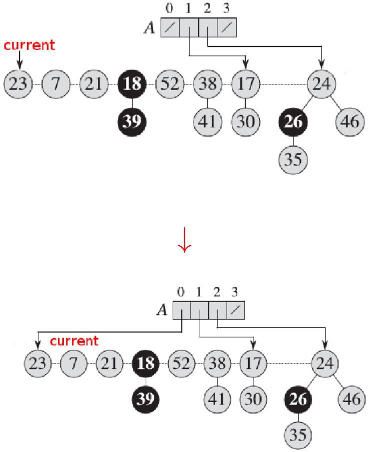
- Now, current=24 has *degree*=2.
- Again, $A[2]$ is empty, so...
- ...get $A[2]$ to point to the root node at current =24.
- Next, move current to the right sibling, i.e. move current from 24 to 23. Why?
  - **root_list** is a circular doubly linked list...
  - ...so the right sibling of 24 is (circularly) 23.
  - therefore, current=23.

Fibonacci heaps – **consolidate** operation ...continued



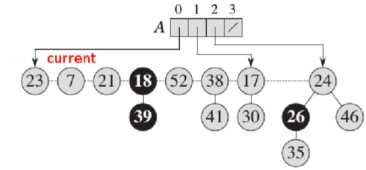
- Now, current= 24 has $degree$=2.
- Again, $A[2]$ is empty, so...
- ...get $A[2]$ to point to the root node at current = 24.
- Next, move current to the right sibling, i.e. move current from 24 to 23. Why?
  - **root_list** is a circular doubly linked list...
  - ...so the right sibling of 24 is (circularly) 23.
  - therefore, current= 23.

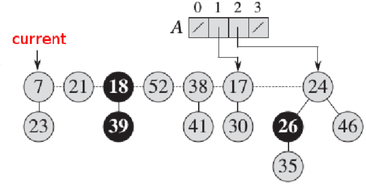
Fibonacci heaps – **consolidate** operation ...continued



- Now, current=23 has $degree$=0.
- $A[0]$ is empty, so...
- ...get $A[0]$ to point to the root node at current =23.
- Next, move current to the right sibling, i.e. current=7.

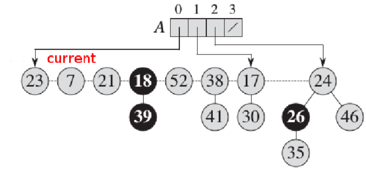
Fibonacci heaps – **consolidate** operation ...continued

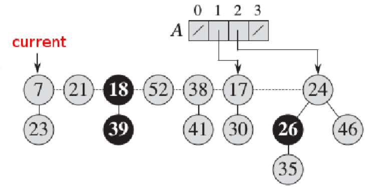




- Now, current=7 has *degree*=0.
- But $A[0]$ is already **occupied** with a pointer to 23.
- Therefore, resolve this clash by **merging** (consolidating) trees with roots 7 and 23, and set $A[0]$ to empty.
- To maintain the (min-)heap property, root node 23 becomes the **child** of root node 7.
- current now points to the root of this merged tree, 7.
- Note: 7.*degree* goes up from 0 to 1.

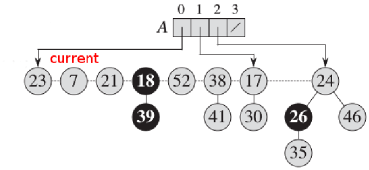
Fibonacci heaps – consolidate operation ...continued

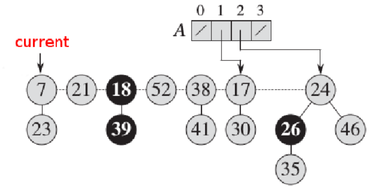




- Now, current=7 has $degree$=0.
- But $A[0]$ is already **occupied** with a pointer to 23.
- Therefore, resolve this clash by **merging** (consolidating) trees with roots 7 and 23, and set $A[0]$ to empty.
- To maintain the (min-)heap property, root node 23 becomes the **child** of root node 7.
- current now points to the root of this merged tree, 7.
- Note: 7.$degree$ goes up from 0 to 1.

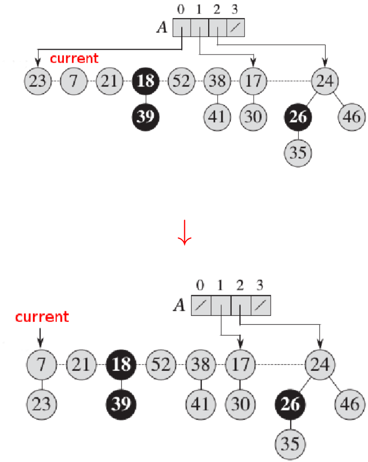
Fibonacci heaps – **consolidate** operation ...continued





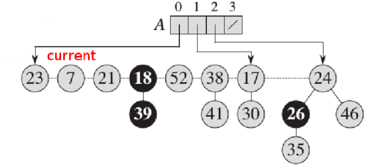
- Now, current=7 has *degree*=0.
- But $A[0]$ is already **occupied** with a pointer to 23.
- Therefore, resolve this clash by **merging** (consolidating) trees with roots 7 and 23, and set $A[0]$ to empty.
- To maintain the (min-)heap property, root node 23 becomes the **child** of root node 7.
- current now points to the root of this merged tree, 7.
- Note: 7.*degree* goes up from 0 to 1.

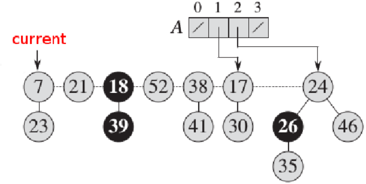
Fibonacci heaps – **consolidate** operation ...continued



- Now, current=7 has $degree$=0.
- But $A[0]$ is already **occupied** with a pointer to 23.
- Therefore, resolve this clash by **merging** (consolidating) trees with roots 7 and 23, and set $A[0]$ to empty.
- To maintain the (min-)heap property, root node 23 becomes the **child** of root node 7.
- current now points to the root of this merged tree, 7.
- Note: 7.$degree$ goes up from 0 to 1.

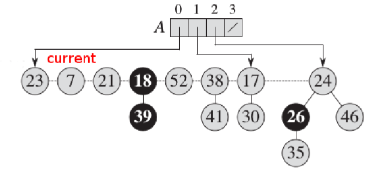
Fibonacci heaps – **consolidate** operation ...continued

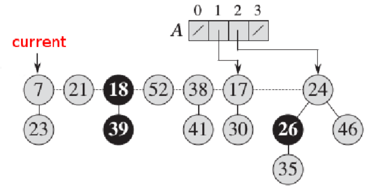




- Now, current=7 has *degree*=0.
- But $A[0]$ is already **occupied** with a pointer to 23.
- Therefore, resolve this clash by **merging** (consolidating) trees with roots 7 and 23, and set $A[0]$ to empty.
- To maintain the (min-)heap property, root node 23 becomes the **child** of root node 7.
- current now points to the root of this merged tree, 7.
- Note: 7.*degree* goes up from 0 to 1.

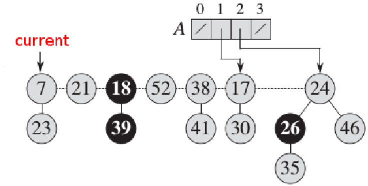
Fibonacci heaps – **consolidate** operation ...continued

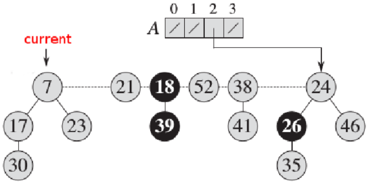




- Now, current=7 has $degree$=0.
- But $A[0]$ is already **occupied** with a pointer to 23.
- Therefore, resolve this clash by **merging** (consolidating) trees with roots 7 and 23, and set $A[0]$ to empty.
- To maintain the (min-)heap property, root node 23 becomes the **child** of root node 7.
- current now points to the root of this merged tree, 7.
- Note: 7.$degree$ goes up from 0 to 1.

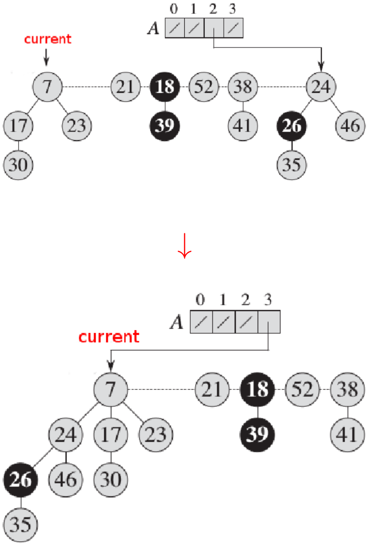
Fibonacci heaps – **consolidate** operation ...continued





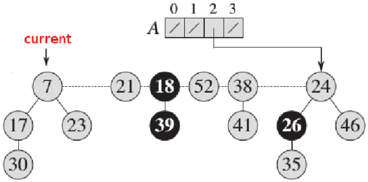
- Repeat: current=(7) has $degree$=1.
- But $A[1]$ is already **occupied** with a pointer to (17).
- Therefore, resolve this clash by **merging** (consolidating) trees with roots (7) and (17), and set $A[1]$ to empty.
- To maintain the (min-)heap property, root node (17) becomes the **child** of root node (7).
- current now points to the root of this merged tree, (7).
- Note: (7).$degree$ goes up from 1 to 2.

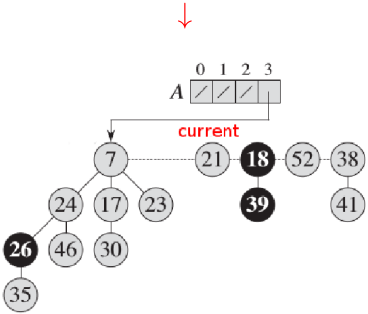
Fibonacci heaps – **consolidate** operation ...continued



- Repeat: current=7 has $degree$=2.
- But $A[2]$ is already **occupied** with a pointer to 24.
- Therefore, resolve this clash by **merging** (consolidating) trees with roots 7 and 24, and set $A[2]$ to empty.
- To maintain the (min-)heap property, root node 24 becomes the **child** of root node 7.
- current now points to the root of this merged tree, 7.
- Note: 7.$degree$ goes up from 2 to 3.
- Since, $A[3]$ is empty, get $A[3]$ to point to the root node at current=7.
- Next, move current to the right sibling, i.e. current=21

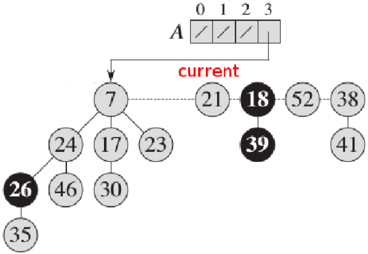
Fibonacci heaps – **consolidate** operation ...continued



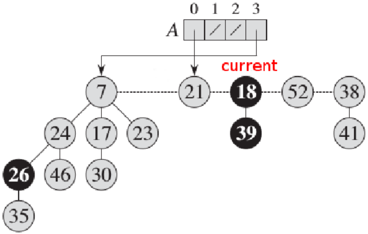


- Repeat: current=7 has $degree=2$.
- But $A[2]$ is already **occupied** with a pointer to 24.
- Therefore, resolve this clash by **merging** (consolidating) trees with roots 7 and 24, and set $A[2]$ to empty.
- To maintain the (min-)heap property, root node 24 becomes the **child** of root node 7.
- current now points to the root of this merged tree, 7.
- Note: 7.$degree$ goes up from 2 to 3.
- Since, $A[3]$ is empty, get $A[3]$ to point to the root node at current=7.
- Next, move current to the right sibling, i.e. current=21

Fibonacci heaps – **consolidate** operation ...continued

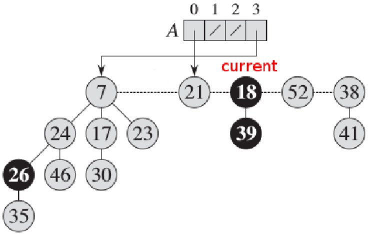


↓

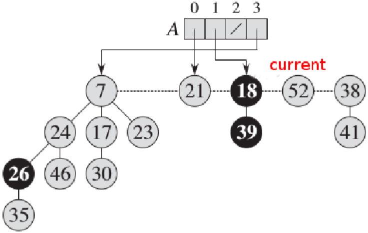


- Now, current=21 has $degree$=0.
- $A[0]$ is empty, so...
- ...get $A[0]$ to point to the root node at current=21.
- Next, move current to the right sibling, i.e current=18.

Fibonacci heaps – **consolidate** operation ...continued

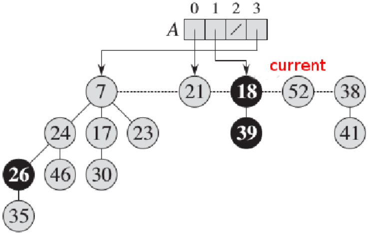


↓

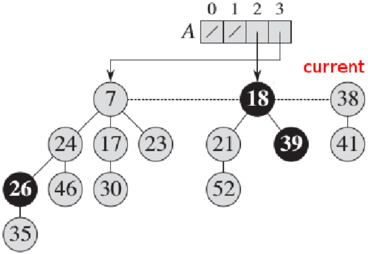


- Now, current=18 has $degree$=1.
- $A[1]$ is empty, so...
- ...get $A[1]$ to point to the root node at current=18.
- Next, move current to the right sibling, i.e current=52.

Fibonacci heaps – **consolidate** operation ...continued

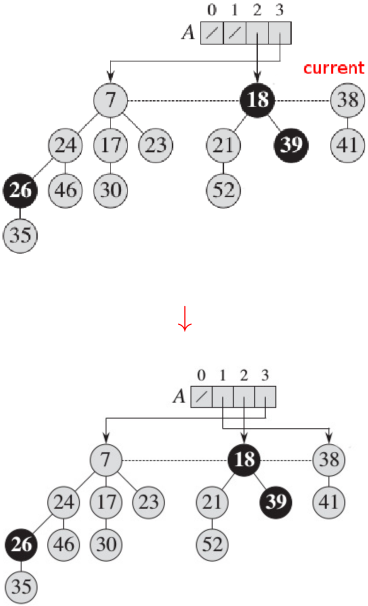


↓



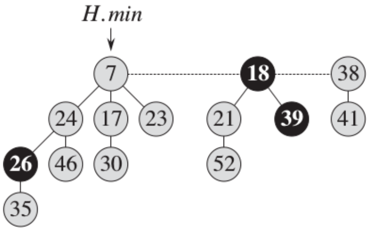
- Now, current=52 has *degree*=0, but $A[0]$ is occupied.
- So, in operations similar to those on slides #16-18...
- ...we get to the state shown in the figure on the left (below).
- current now points to root 38.

Fibonacci heaps – **consolidate** operation …continued



- Now, current=38 has *degree*=1.
- $A[1]$ is empty, so…
- …get $A[1]$ to point to the root node at current=38.
- **consolidate** has now completed one full cycle on the doubly linked list. So, consolidation STOPS!

Fibonacci heaps – **consolidate** operation ...continued



- **extract-min** operation (and consolidation) is now complete.
- Note: during the process of cycling through the **root-list** (during consolidation), we can keep track of the minimum root encountered, and update $H.min$.

Run-time complexity is $O(\log(N))$ amortized.[a]

[a] Skipping amortized analysis for lack of time. See non-examinable hand-scribbled proof/analysis uploaded on Moodle under week 6 topics.

Fibonacci heaps – **decrease-key** operation

We want to decrease key of any node x in a Fibonacci heap.*

- This can be handled in two cases:

  case 1: When this operation does not violate the heap property (slide #25)

  case 2: When it does!

  - We will handle this over subcases, Case 2a (slide #26) and Case 2b (slide #27-29).

*Note: as with binary heaps, Fibonacci heaps are inefficient to **search** for any node x; For this reason, **decrease-key** on x assumes a pointer to x as part of its input.

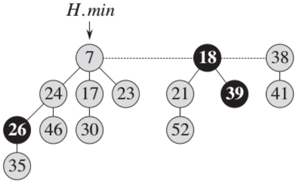
Fibonacci heaps – **decrease-key**: Case 1

When **decrease-key** does not violate the heap property. Simply decrease the key on the node, and we are done!

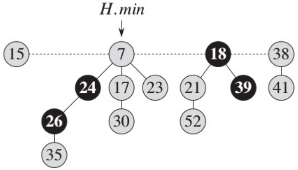
*Note: as with binary heaps, Fibonacci heaps are inefficient to **search** for any node x; For this reason, **decrease-key** on x assumes a pointer to x as part of its input.

Fibonacci heaps – **decrease-key**: Case 2a

Consider an example where we want to **decrease-key** of node with key= 46 to key= 15.



↓**decrease-key**(46 → 15)

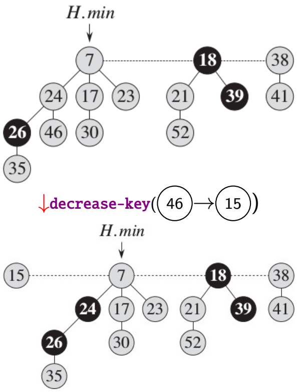


- Decreasing 46 to 15 violates the heap property because its parent 24 > 15.
- To address this violation:
  - Cut the subtree rooted at 15 (prev. 46)...
  - ...and promote it into the root list. (Update $H.min$, if necessary.)
  - If necessary, update the mark of 15 to 'unmarked' after promoting to the root level.
  - Since parent 24 was originally **unmarked** (i.e., hasn't yet lost a child), **mark it** (i.e., has lost a child);

*Note: as with binary heaps, Fibonacci heaps are inefficient to **search** for any node x; For this reason, **decrease-key** on x assumes a pointer to x as part of its input.

Fibonacci heaps – **decrease-key**: Case 2a

Consider an example where we want to **decrease-key** of node with key= 46 to key= 15.

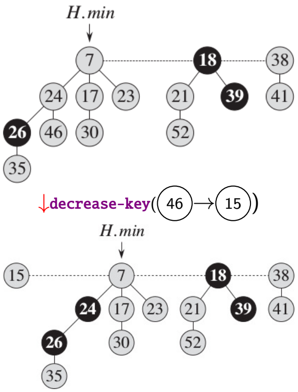


- Decreasing 46 to 15 violates the heap property because its parent 24 > 15.
- To address this violation:
  - Cut the subtree rooted at 15 (prev. 46)...
  - ...and promote it into the root list. (Update $H.min$, if necessary.)
  - If necessary, update the mark of 15 to 'unmarked' after promoting to the root level.
  - Since parent 24 was originally **unmarked** (i.e., hasn't yet lost a child), **mark it** (i.e., has lost a child);

*Note: as with binary heaps, Fibonacci heaps are inefficient to **search** for any node x; For this reason, **decrease-key** on x assumes a pointer to x as part of its input.

Fibonacci heaps – **decrease-key**: Case 2a

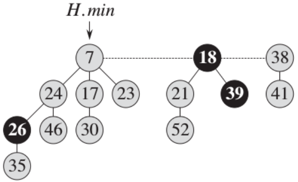
Consider an example where we want to **decrease-key** of node with key= 46 to key= 15.



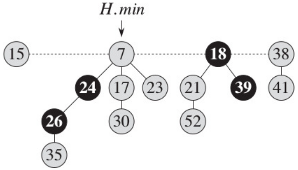
- Decreasing 46 to 15 violates the heap property because its parent 24 > 15.
- To address this violation:
  - Cut the subtree rooted at 15 (prev. 46)...
  - ...and promote it into the root list. (Update $H.min$, if necessary.)
  - If necessary, update the mark of 15 to 'unmarked' after promoting to the root level.
  - Since parent 24 was originally **unmarked** (i.e., hasn't yet lost a child), **mark it** (i.e., has lost a child);

*Note: as with binary heaps, Fibonacci heaps are inefficient to **search** for any node x; For this reason, **decrease-key** on x assumes a pointer to x as part of its input.

Consider an example where we want to **decrease-key** of node with key= 46 to key= 15.



↓**decrease-key**(46 → 15)

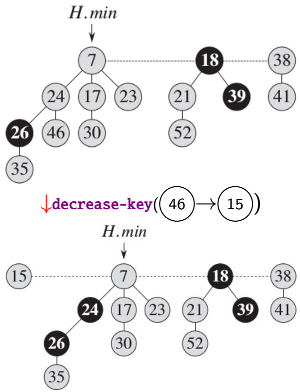


- Decreasing 46 to 15 violates the heap property because its parent 24 > 15.
- To address this violation:
  - Cut the subtree rooted at 15 (prev. 46)...
  - ...and promote it into the root list. (Update $H.min$, if necessary.)
  - If necessary, update the mark of 15 to 'unmarked' after promoting to the root level.
  - Since parent 24 was originally **unmarked** (i.e., hasn't yet lost a child), **mark it** (i.e., has lost a child);

*Note: as with binary heaps, Fibonacci heaps are inefficient to **search** for any node x; For this reason, **decrease-key** on x assumes a pointer to x as part of its input.

Fibonacci heaps – **decrease-key**: Case 2a

Consider an example where we want to **decrease-key** of node with key= 46 to key= 15.

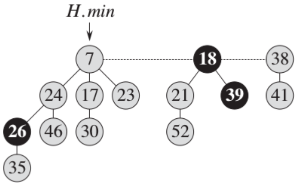


- Decreasing 46 to 15 violates the heap property because its parent 24 > 15.
- To address this violation:
  - Cut the subtree rooted at 15 (prev. 46)...
  - ...and promote it into the root list. (Update $H.min$, if necessary.)
  - If necessary, update the mark of 15 to 'unmarked' after promoting to the root level.
  - Since parent 24 was originally **unmarked** (i.e., hasn't yet lost a child), **mark it** (i.e., has lost a child);

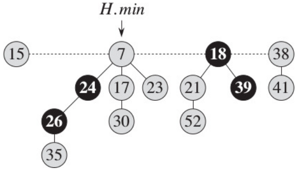
*Note: as with binary heaps, Fibonacci heaps are inefficient to **search** for any node x; For this reason, **decrease-key** on x assumes a pointer to x as part of its input.

Fibonacci heaps – **decrease-key**: Case 2a

Consider an example where we want to **decrease-key** of node with key= 46 to key= 15.



↓**decrease-key**(46 → 15)

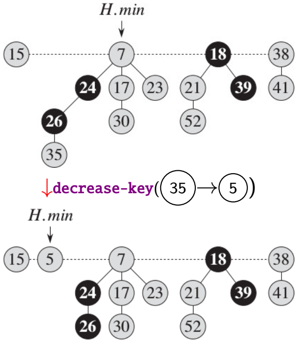


- Decreasing 46 to 15 violates the heap property because its parent 24 > 15.
- To address this violation:
  - Cut the subtree rooted at 15 (prev. 46)...
  - ...and promote it into the root list. (Update $H.min$, if necessary.)
  - If necessary, update the mark of 15 to 'unmarked' after promoting to the root level.
  - Since parent 24 was originally **unmarked** (i.e., hasn't yet lost a child), **mark it** (i.e., has lost a child);

*Note: as with binary heaps, Fibonacci heaps are inefficient to **search** for any node x; For this reason, **decrease-key** on x assumes a pointer to x as part of its input.

Fibonacci heaps – **decrease-key**: Case 2b

Another case arises, when the original parent of the subtree, promoted to the root level, is already ‘**marked**’ – consider the example where we want to **decrease-key** of node (with key=) 35 to 5.

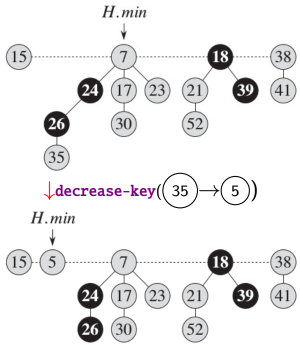


- Decreasing 35 to 5 violates the heap property...
- ...because its parent 26 > 5 (previously 35).
- To address this violation:
  - Cut the subtree rooted at 5 (prev. 35)...
  - ...and promote it into the root list. (Update $H.min$, if necessary.)
  - If necessary, update the mark of 5 to ‘**unmarked**’ after promoting to the root level.
  - But parent 26 is already ‘**marked**’ (i.e., has lost one child previously);
  - so, repeat this cut-and-promote-to-root process for 26...

*Note: as with binary heaps, Fibonacci heaps are inefficient to **search** for any node x; For this reason, **decrease-key** on x assumes a pointer to x as part of its input.

Fibonacci heaps – **decrease-key**: Case 2b

Another case arises, when the original parent of the subtree, promoted to the root level, is already '**marked**' – consider the example where we want to **decrease-key** of node (with key=) 35 to 5.

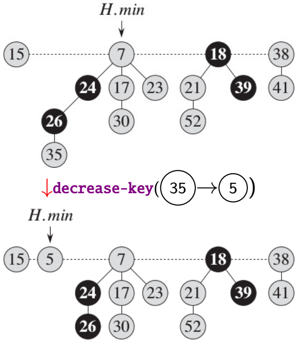


- Decreasing 35 to 5 violates the heap property...
- ...because its parent 26 > 5 (previously 35).
- To address this violation:
  - Cut the subtree rooted at 5 (prev. 35)...
  - ...and promote it into the root list. (Update $H.min$, if necessary.)
  - If necessary, update the mark of 5 to '**unmarked**' after promoting to the root level.
  - But parent 26 is already '**marked**' (i.e., has lost one child previously);
  - so, repeat this cut-and-promote-to-root process for 26...

*Note: as with binary heaps, Fibonacci heaps are inefficient to **search** for any node x; For this reason, **decrease-key** on x assumes a pointer to x as part of its input.

Fibonacci heaps – **decrease-key**: Case 2b

Another case arises, when the original parent of the subtree, promoted to the root level, is already '**marked**' – consider the example where we want to **decrease-key** of node (with key=) 35 to 5.

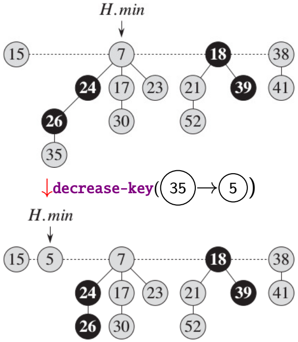


- Decreasing 35 to 5 violates the heap property...
- ...because its parent 26 > 5 (previously 35).
- To address this violation:
  - Cut the subtree rooted at 5 (prev. 35)...
  - ...and promote it into the root list. (Update $H.min$, if necessary.)
  - If necessary, update the mark of 5 to '**unmarked**' after promoting to the root level.
  - But parent 26 is already '**marked**' (i.e., has lost one child previously);
  - so, repeat this cut-and-promote-to-root process for 26...

*Note: as with binary heaps, Fibonacci heaps are inefficient to **search** for any node x; For this reason, **decrease-key** on x assumes a pointer to x as part of its input.

Fibonacci heaps – **decrease-key**: Case 2b

Another case arises, when the original parent of the subtree, promoted to the root level, is already '**marked**' – consider the example where we want to **decrease-key** of node (with key=) 35 to 5.

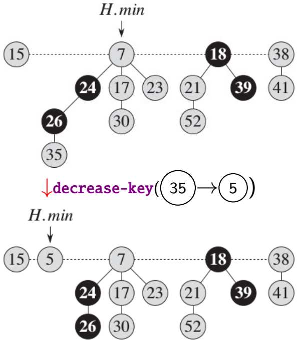


- Decreasing 35 to 5 violates the heap property...
- ...because its parent 26 > 5 (previously 35).
- To address this violation:
  - Cut the subtree rooted at 5 (prev. 35)...
  - ...and promote it into the root list. (Update $H.min$, if necessary.)
  - If necessary, update the mark of 5 to '**unmarked**' after promoting to the root level.
  - But parent 26 is already '**marked**' (i.e., has lost one child previously);
  - so, repeat this cut-and-promote-to-root process for 26...

*Note: as with binary heaps, Fibonacci heaps are inefficient to **search** for any node x; For this reason, **decrease-key** on x assumes a pointer to x as part of its input.

Fibonacci heaps – **decrease-key**: Case 2b

Another case arises, when the original parent of the subtree, promoted to the root level, is already '**marked**' – consider the example where we want to **decrease-key** of node (with key=) 35 to 5.

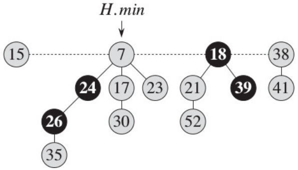


- Decreasing 35 to 5 violates the heap property...
- ...because its parent 26 > 5 (previously 35).
- To address this violation:
  - Cut the subtree rooted at 5 (prev. 35)...
  - ...and promote it into the root list. (Update $H.min$, if necessary.)
  - If necessary, update the mark of 5 to '**unmarked**' after promoting to the root level.
  - But parent 26 is already '**marked**' (i.e., has lost one child previously);
  - so, repeat this cut-and-promote-to-root process for 26...

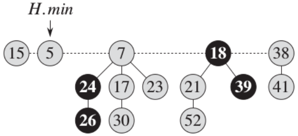
*Note: as with binary heaps, Fibonacci heaps are inefficient to **search** for any node x; For this reason, **decrease-key** on x assumes a pointer to x as part of its input.

Fibonacci heaps – **decrease-key**: Case 2b

Another case arises, when the original parent of the subtree, promoted to the root level, is already '**marked**' – consider the example where we want to **decrease-key** of node (with key=) 35 to 5.



↓**decrease-key**(35→5)

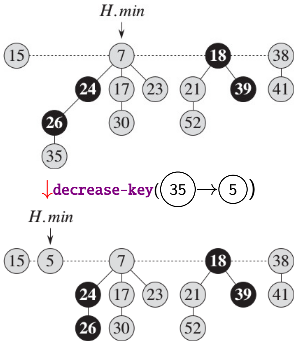


- Decreasing 35 to 5 violates the heap property...
- ...because its parent 26 > 5 (previously 35).
- To address this violation:
  - Cut the subtree rooted at 5 (prev. 35)...
  - ...and promote it into the root list. (Update $H.min$, if necessary.)
  - If necessary, update the mark of 5 to '**unmarked**' after promoting to the root level.
  - But parent 26 is already '**marked**' (i.e., has lost one child previously);
  - so, repeat this cut-and-promote-to-root process for 26...

*Note: as with binary heaps, Fibonacci heaps are inefficient to **search** for any node x; For this reason, **decrease-key** on x assumes a pointer to x as part of its input.

Fibonacci heaps – **decrease-key**: Case 2b

Another case arises, when the original parent of the subtree, promoted to the root level, is already '**marked**' – consider the example where we want to **decrease-key** of node (with key=) 35 to 5.

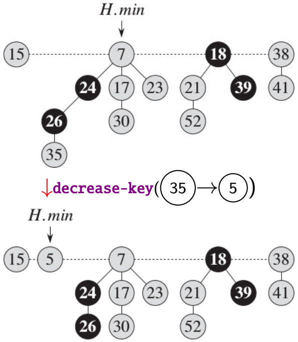


- Decreasing 35 to 5 violates the heap property...
- ...because its parent 26 $>$ 5 (previously 35).
- To address this violation:
  - Cut the subtree rooted at 5 (prev. 35)...
  - ...and promote it into the root list. (Update $H.min$, if necessary.)
  - If necessary, update the mark of 5 to '**unmarked**' after promoting to the root level.
  - But parent 26 is already '**marked**' (i.e., has lost one child previously);
  - so, repeat this cut-and-promote-to-root process for 26...

*Note: as with binary heaps, Fibonacci heaps are inefficient to **search** for any node x; For this reason, **decrease-key** on x assumes a pointer to x as part of its input.

Fibonacci heaps – **decrease-key**: Case 2b

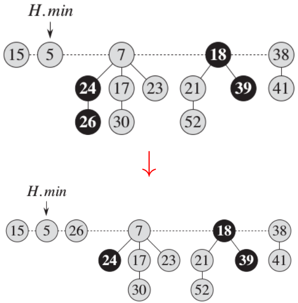
Another case arises, when the original parent of the subtree, promoted to the root level, is already '**marked**' – consider the example where we want to **decrease-key** of node (with key=) 35 to 5.



- Decreasing 35 to 5 violates the heap property...
- ...because its parent 26 $>$ 5 (previously 35).
- To address this violation:
  - Cut the subtree rooted at 5 (prev. 35)...
  - ...and promote it into the root list. (Update $H.min$, if necessary.)
  - If necessary, update the mark of 5 to '**unmarked**' after promoting to the root level.
  - But parent 26 is already '**marked**' (i.e., has lost one child previously);
  - so, repeat this cut-and-promote-to-root process for 26...

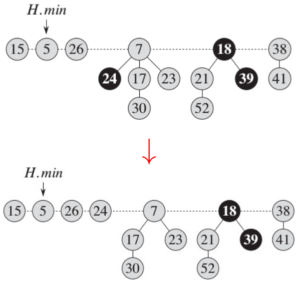
*Note: as with binary heaps, Fibonacci heaps are inefficient to **search** for any node x; For this reason, **decrease-key** on x assumes a pointer to x as part of its input.

Fibonacci heaps – **decrease-key**: Case 2b (continued)



- ... so, cut the subtree rooted at 26.
- ...and promote it into the root list.
- If necessary, update the mark of 26 to '**unmarked**' after promoting to the root level.
- But its parent 24 is again already '**marked**' (i.e., has lost one child previously);
- so, repeat this cut-and-promote-to-root process for 24...

Fibonacci heaps – **decrease-key**: Case 2b (continued)



- ...now, cut the subtree rooted at 24.
- ...and promote it into the root list.
- If necessary, update the mark of 24 to 'unmarked' after promoting to the root level.
- Finally, since its original parent 7 is 'unmarked', STOP!
  - Btw, we do not have to 'mark' a previously unmarked root (when it is a parent of a child that is cut and promoted).

Run-time complexity of **decrease-key** is $O(1)$ amortized. Unfortunately, we are short of time to prove this, so we will omit it for now.

Fibonacci heaps – **Union** operation:

Union operation involves combining two Fibonacci heaps, H_1 and H_2 into one (used during **consolidation**):

- Takes $O(1)$ time. Why?
  - This involves combining two root lists...
  - ...each represented by a circular doubly-linked lists,
  - ... and accessible via their respective minimum (root) elements, $H_1.min$ and $H_2.min$...
  - ...before linking them into a single heap.
  - (reason this fully during self-study)

Fibonacci heaps – **delete** operations:

delete operation deletes some specified node x. This can be composed using the following two operations, which we already discussed:

- **decrease-key** of x to $-\infty$.
- **extract-min**.

*Note: as with binary heaps, Fibonacci heaps are inefficient to **search** for any node x; For this reason, **delete**(x) assumes a pointer to x as part of its input.

Summary of Fibonacci heaps

Operation	Fibonacci heap	Binomial heap
make-new-heap	$O(1)$	$O(1)$
min	$O(1)$	$O(\log N)$
extract-min	$O(\log N)$ (amortized)	$O(\log N)$
merge	$O(1)$	$O(\log N)$
decrease-key	$O(1)$ (amortized)	$O(\log N)$
delete	$O(\log N)$ (amortized)	$O(\log N)$
insert	$O(1)$	$O(\log N)$ worst-case $O(1)$ amortized

In the next lecture...

B-Trees

-=o0o=-
END
-=o0o=-