Prepared by: [Arun Konagurthu]

FIT3155: Advanced Algorithms and Data Structures

Weeks 5,6: **Binomial and Fibonacci heaps**

Faculty of Information Technology, Monash University

What is covered in these?

Binomial heap and Fibonacci heap

Source material and recommended reading

- Weiss, Data Structures and Algorithm Analysis (Chapters 6.8, 11.1, 11.2)
- Cormen et al., Introduction to Algorithms (Chapter 19):
  Binomial heaps [online link]
  Fibonacci heaps [online link]

Priority queues (implemented using heaps)

Recall from FIT2004 that the heap data structure was used in several applications:

- Heap sort
- Dijkstra's shortest path algorithm
- Prim's algorithm

Recall also that this data structure supports the following operations*:

- **insert** a new element (key/priority+payload) into a heap
- identify the **min** element in an existing heap
- **extract-min** (identify and delete min) element in an existing heap
- **decrease-priority** of an element in an existing heap

*As with these slides, default heap operations are defined over a min-heap. One could alternatively define **max**, **extract-max**, **increase-priority** operations on a max-heap.

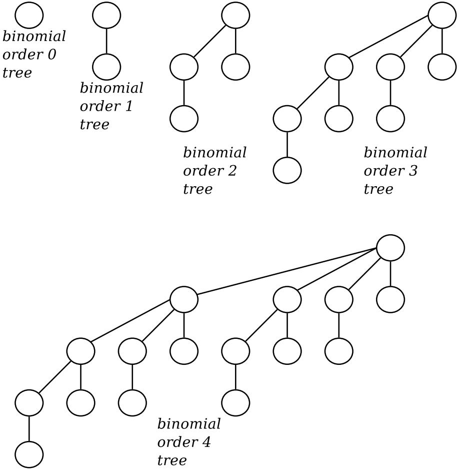
Mergeable heaps

Today (binomial heap) and next start of next lecture (Fibonacci heap), we will learn about mergeable heaps that support (at least) the following operations:

- **insert**: inserts a new element into the existing heap
- **min**: finds the min element in the heap
- **extract-min**: finds and deletes the min element in the heap
- **merge/union**: combines two heaps into one
- **decrease-priority**: decreases the elements key/priority
- **delete**: removes an element from the heap

Part-1: Binomial heaps

Before Binomial **heap**, let us define a binomial **tree**



Binomial trees are defined recursively:

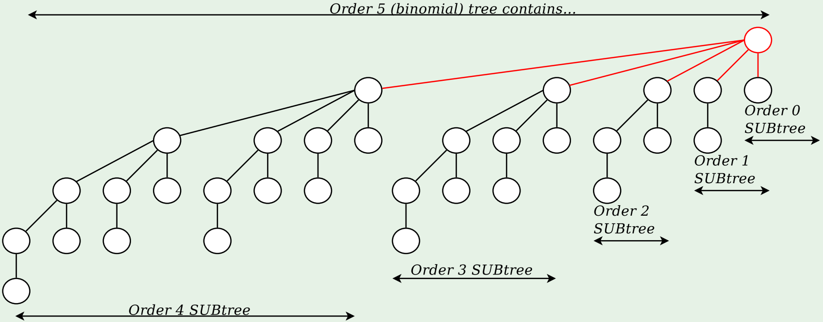
- The binomial tree of order 0 (or B_0 in short) is a single node tree
- The binomial tree of order 1 (B_1) is created from two B_0 trees, by making one B_0 tree the child of the other.
- The binomial tree of order 2 (B_2) is created from two B_1 trees, by making one B_1 tree the child of the other.
- and so on...

Properties of a Binomial **tree**

Any binomial tree of order k has the following properties:

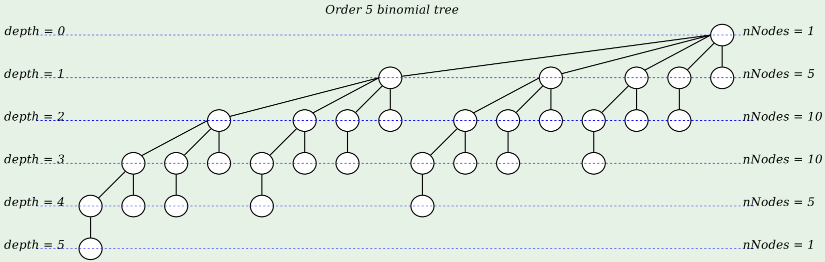
- The number of nodes in any B_k is 2^k.
- The height of any B_k is k.
- The root node of any B_k tree has k subtrees as children.
- Deleting the root node of B_k (with its edges/links) yields k independent lower order binomial trees $B_{k-1}, B_{k-2}, \ldots, B_0$.

Example



Why are these trees called **binomial**?

Example: B_5



Main property

A main property of any B_k tree is that the **number of nodes** at any given depth d is given by the **binomial coefficient** $\binom{k}{d}$, that is "k-choose-d"

What is a binomial **heap**?

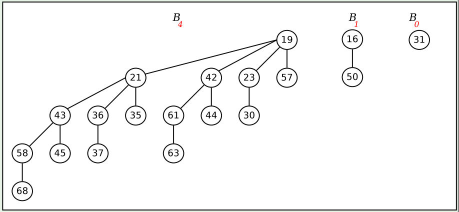
A binomial **heap** is a collection/set of binomial **trees** such that:

- each binomial tree in the set satisfies the heap property – i.e., each tree-node's key/priority is $\leq$ its children's keys/priorities.
- There is **at most** one (i.e. either 0 or 1) binomial tree of any given order in that set.

Example

On the right is a binomial **heap** that contains a collection/set of binomial **trees**:

- one B_4 tree
- zero B_3 tree
- zeroB_2 tree
- one B_1 tree
- one B_0 tree



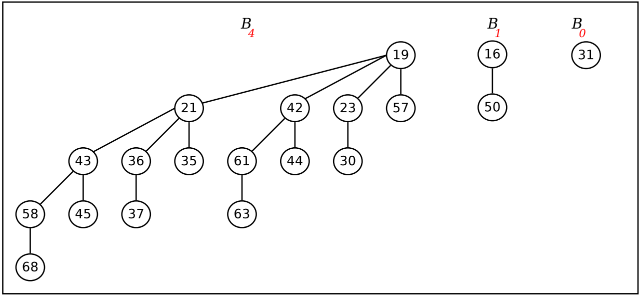
Binomial heap properties

Properties

For any binomial **heap** containing N elements, the following properties hold:

- There are at most $\lfloor \log_2 N \rfloor + 1$ binomial **trees**
- The height of each binomial **tree** is $\leq \lfloor \log_2 N \rfloor$
- The '1's in the **binary representation** of N tell us which order binomial **trees** are present in the collection forming this binomial **heap** of N elements.
- the element with **minimum** key is one of the of root nodes of the **trees** in the collection.

Binomial heap properties – Example



Example

For the above binomial **heap**:

- $N = 19$.
- Number of trees is 3
- binary representation of 19 is: $\overset{4}{1}\ \overset{3}{0}\ \overset{2}{0}\ \overset{1}{1}\ \overset{0}{1}$ (therefore contains B_4, B_1, B_0)

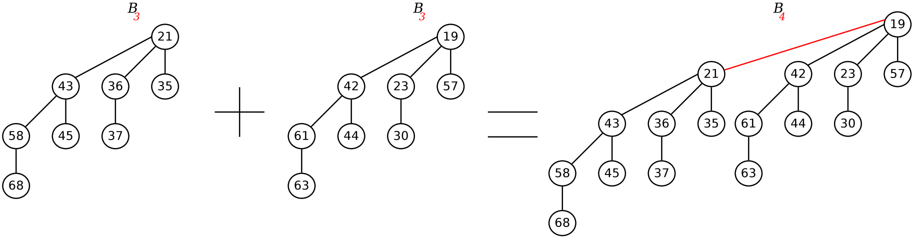
Representing a binomial heap

- Unlike **binary** heaps, **binomial** heaps are stored explicitly using a **tree** data structure.
- Each node x:
  - is denoted by a **key**,
  - has associated **payload** information
  - has a pointer **parent**$[x]$ to its parent node
  - has a pointer **child**$[x]$ to its **leftmost** child node
    - If node x has zero children, then **child**$[x] = nil$
  - has a pointer **sibling**$[x]$ to the immediate **sibling** of x to its right.
    - If node x is the rightmost child of its parent, then **sibling**$[x] = nil$
  - stores **degree**$[x]$ which is the number of children of x (i.e., same as the **order** of the binomial tree rooted at x)
- Finally, the roots of the binomial tress within a binomial heap are organized in a linked list, referred to as the **root list**.

operations on a binomial heap

Merging two binomial **trees** into one

- First, merging two binomial **trees**, each of the **same** order (say) k results in an order $k+1$ binomial tree, where:
  - the two roots are linked, such that...
  - ...the root containing the **larger** key becomes the **child** of the smaller root.



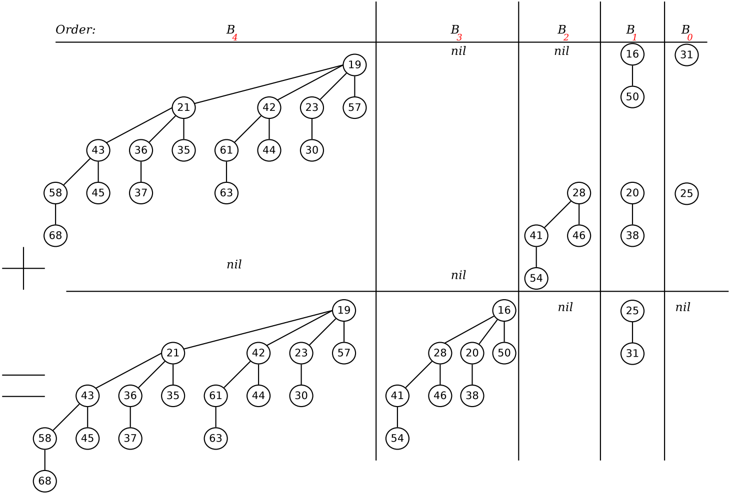
Binomial **heap** operation – **merge**/**consolidate** two binomial **heaps** into one

- With merging of two binomial **trees** established (see previous slide), we can now define **merge**/**consolidate** operation on two binomial **heaps**.
- Heaps are merged in a way that is reminiscent of how we add two numbers in binary:

Example: addition of $19 + 7 = 26$ in binary

Order:	4	3	2	1	0
carry:		1	1	1	
	1	0	0	1	1
$+$	0	0	1	1	1
Result:	1	1	0	1	0

Example of merging 2 binomial heaps containing 19 and 7 elements each



(To be discussed during the lecture)

Running time of `merge` operation between 2 binomial heaps

- Running time is $O(\log N)$ worst-case – why?
  - time is bounded by maximum number of possible merges between trees of the same order within the heaps.
  - the number of trees in each heap containing N elements is bounded by $\lfloor \log N \rfloor + 1$
  - merging two heaps in worst case requires $2 \times (\lfloor \log N \rfloor + 1)$ tree merges

Binomial **heap** operation – **extract-min**

We use this to identify and delete the minimum element among all **root nodes** of the trees in the heap.

- Identify the **min** root node among the trees in the heap.
- From slide #9, we know that deleting the root node of any B_k tree yields: $B_{k-1}, B_{k-2}, \ldots, B_0$.
- If we promote these subtrees to the root level of the existing binomial heap...
- ...this might create multiple trees of the same order (violating the definition of a binomial heap – see slide #11).
- So, progressively **merge** the binomial trees of the same order (starting from 0) until the binomial heap definition is satisfied.

(Example will be handled during the lecture)

Running time of **extract-min** operation

- Running time is $O(\log N)$ worst-case – why?
- Effort required to find the **min** is $O(\log N)$. (see slide $\#12$)
- Effort required to promote subtrees formed upon deletion to root level is $O(\log N)$ – the number of these subtrees is bounded by $\lfloor \log N \rfloor$.
- Effort required to merge multiple trees into a binomial heap is also $O(\log N)$. (see slide $\#19$)
- Total effort: $O(\log N)$

Binomial **heap** operation – **decrease-priority**

We want to decrease priority of any node x in a binomial heap containing N elements. †

- decrease priority of node x.
- if min-heap property is violated (i.e. $x < parent[x]$), bubble up node x.
- Running time (worst-case): $O(\log N)$. Note: the depth of the binomial tree in which x resides is bounded above by $\lfloor \log N \rfloor$

(Example will be handled during the lecture)

†Note: as with binary heaps, binomial heaps are inefficient to **search** for any node x (except the root); For this reason, **decrease-priority** on x assumes a pointer to x as part of its input.

Binomial **heap** operation – **delete**

We want to delete any node x in a binomial heap containing N elements. ‡

- run **decrease-priority** by setting x to $-\infty$.
- run **extract-min**.
- Running time (worst-case): $O(\log n)$.

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

Binomial **heap** operation – **insert**

We want to insert a new element x into an existing binomial heap H_1

- Make a new binomial heap H_2 with x as its only element.
- run **merge**(H_1, H_2).
- At face value, the runtime per single **insert** takes $O(\log N)$ effort.

Amortized analysis of **insert** operation

Consider the problem of building a **binomial** heap of N elements:

- From FIT2004, we know that at least a **binary** heap of N elements can be built in $O(N)$ time.
- What about a **binomial** heap then?

claim

A **binomial** heap of N elements can be built by N successive inserts in $O(N)$-time.

Amortized analysis of **insert** operation ...continued(2)

- Time required for inserting **each** element x into a heap H_1 (starting from an empty heap) involves:
  - time to create a new binomial heap H_2 containing only 1 element x – which requires constant effort, **plus**
  - time to merge H_2 into H_1. It isn't fully clear yet how many merges (between same-order binomial trees) will be required in each insert operation.
- Total over N insertions requires:
  - $O(N)$ **plus**
  - total merging time.

Amortized analysis of **insert** operation ...continued(3)

It is easy to see (by beholding how the numbers starting from 0 change when 1 is added each time):

- the first insertion into an empty H_1 heap requires zero merges. Why?
- the second insertion involves exactly one merge between two B_0 binomial trees, yielding a heap containing one B_1 tree.
- the third insertion involves zero merges
  - H_1 before insertion contains 2 elements (contained in 1 B_1 tree).
  - merging the new inserted element into H_1 adds only a new B_0 tree to the existing B_1 tree. Therefore no merges.
- the fourth insertion involves exactly two merges – why?.
- the fifth insertion involves zero merges – why?
- the sixth insertion involves one merge – why?
- $\vdots$

Amortized analysis of **insert** operation ...continued(3)

When inserting N elements, if the binary representation of number elements in H_1 before each insertion ends in

- `.......0`, the effort takes only 1 unit of time.
- `......01`, the effort takes only 2 units of time.
- `.....011`, the effort takes only 3 units of time.
- `....0111`, the effort takes only 4 units of time.
- `...01111`, the effort takes only 5 units of time.
- $\vdots$

Total time over N insertions

- $T = \frac{N}{2} \times 1 + \frac{N}{4} \times 2 + \frac{N}{8} \times 3 \ldots \leq 2N$
- Such series is called an Arithmetico-Geometric series.

Thus total time is bounded by $O(N)$, implying that each **insert** into a binomial heap is $O(1)$ amortized!

Summary of Binomial heaps

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

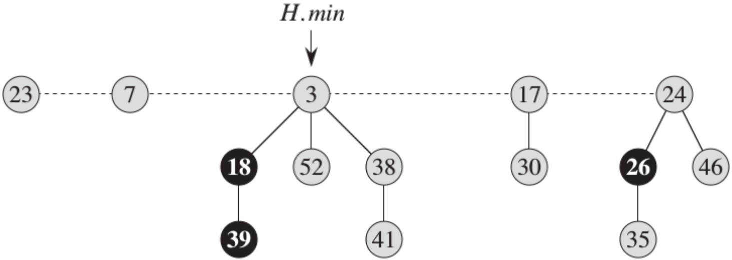
Part-2: Fibonacci heaps

Motivation for Fibonacci heaps

- Improve complexity of **Dijkstra's** shortest path algorithm – Recall this from FIT2004?
- Maintains a collection of trees (much like Binomial heaps), however:
  - ...trees in the collection are **less stringent** in their definitions.
    - While a **binomial heap** performs eager merging/consolidation of trees after each and every **extract-min** or **insert** operations...
    - ...**Fibonacci heap**, on the other hand, does a lazy consolidation/merging, by deferring any merging/consolidation until next **extract-min** operation.

Example of a Fibonacci heap

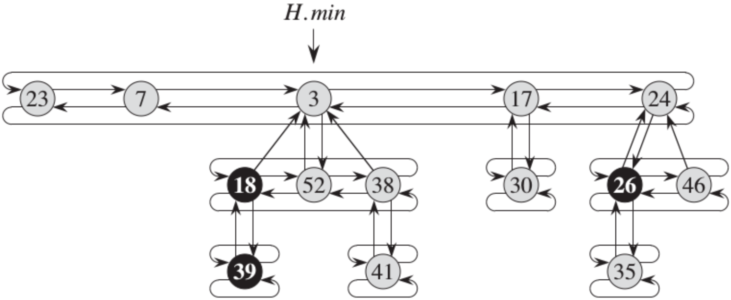
- A Fibonacci heap H containing 5 trees, with total 14 elements.



- $H.min$ 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 as **black coloured nodes above**)...
  - ...or **unmarked**/regular (shown as the grey coloured nodes above)
  - We will examine in later slides what this means.

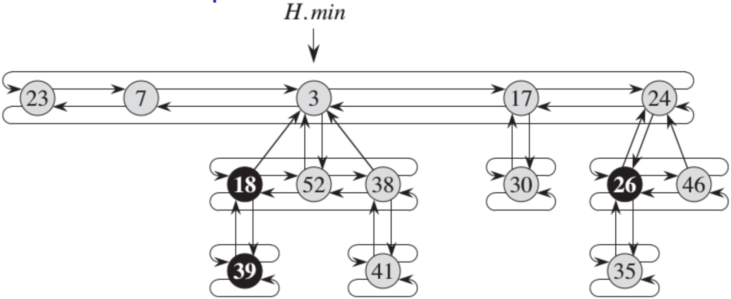
Fibonacci heaps are best represented using circular doubly linked lists

- Circular doubly linked list representation of the example in the previous slide.



- 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.

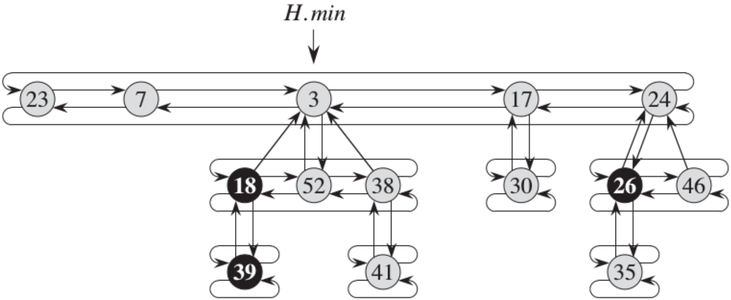
Fibonacci heaps – Attributes



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-priority** operation what this means.
  - Quickly, '**marked**' implies the node has lost a child; **unmarked**' implies it hasn't lost a child. Details when slides #51-54 are covered.
  - Eg: 18 is '**marked**'. 30 is '**unmarked**'.

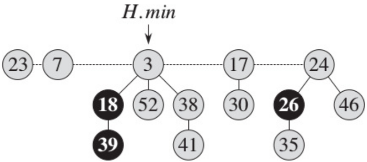
Fibonacci heaps – Attributes (continued)



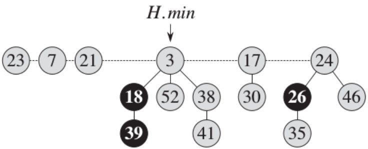
- Access to the Fibonacci heap H is via the pointer to the **`minimum`** (priority) node in the entire heap, denoted by $H.min$.
- Roots of all trees in the Fibonacci heap are connected by a **`root_list`**,
- ...where each tree's root can be accessed via $left$ and $right$ pointers, starting from $H.min$.

Fibonacci heaps – **insert** operation

insert(x) into a Fibonacci heap H. (Here $x =$ 21.)



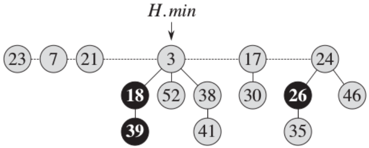
↓ **insert** 21



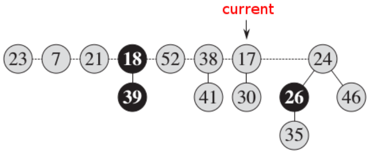
- Access H via the pointer $H.min$.
- **insert** x into the **root_list**, making it the left sibling of $H.min$ element/root.
- if $x < H.min$ (comparing the respective priorities/keys), update $H.min$ to point to the new x root element.
- This is $O(1)$-time operation.

Fibonacci heaps – **extract-min** operation

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



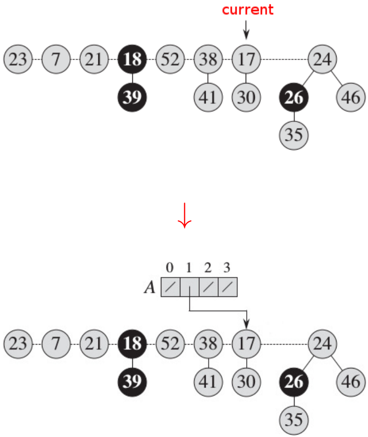
↓ **extract-min**



- Identify minimum element via the pointer $H.min$.
- Extract minimum element (= 3 in this running example),...
- ...promote/add all children (subtrees) to the root list, and
- ...set the current pointer to the *right* sibling of $H.min$.
- IMPORTANT: Now run **consolidate** (or merge) operation.
  - **consolidate** operation ensures that no two roots have the same degree.

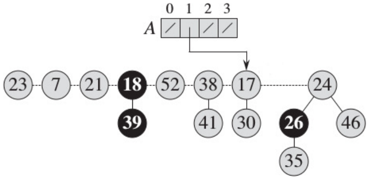
Fibonacci heaps – **consolidate** operation

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

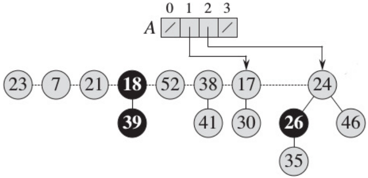


- Starting from current which is pointing to 17...
- Maintain an auxiliary array A to keep track of the root nodes indexed by their $degrees$ (i.e., number of children). Initially, A is empty.
- Since the root node at current=17 has $degree = 1$...
- ...and $A[1]$ slot is empty, so...
- ...get $A[1]$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

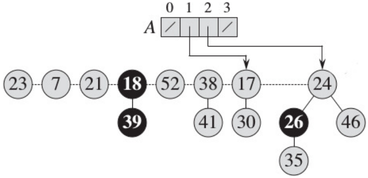


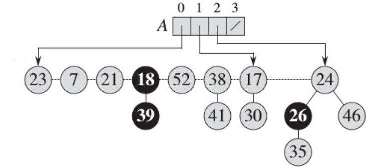
↓



- 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.

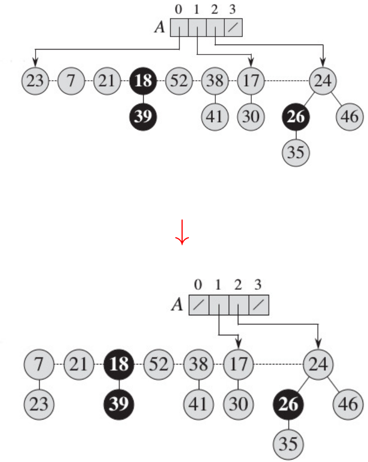
Fibonacci heaps – `consolidate` operation ...continued





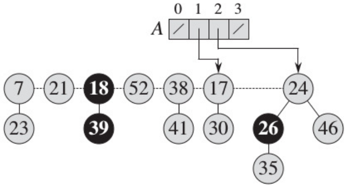
- Now, current=23 has *degree*=0.
- $A[0]$ is empty, so...
- ...get $A[0]$ to point to the root node at current =23.
- Next, move current to the right sibling, i.e. current=7.

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

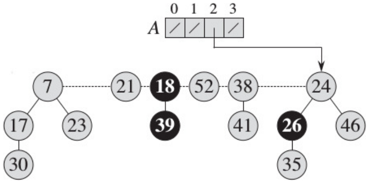


- Now, current=7 has *degree*=0.
- But $A[0]$ is already **occupied** with a pointer to 23.
- Therefore, resolve this clash by **merging** (consolidating) trees with roots 7 and 23, and set $A[0]$ to empty.
- To maintain the (min-)heap property, root node 23 becomes the **child** of root node 7.
- current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

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

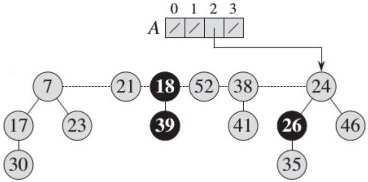


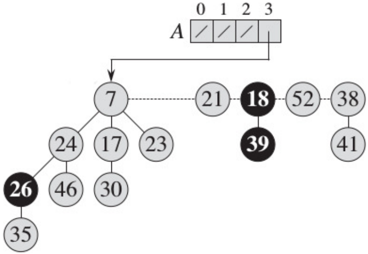
↓



- 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.

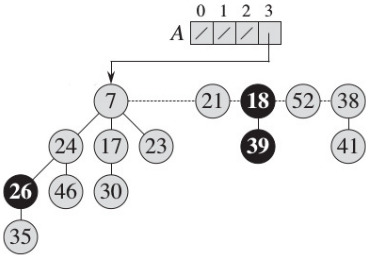
Fibonacci heaps – **consolidate** operation ...continued

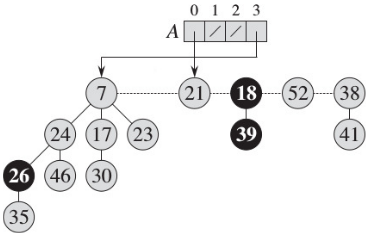




- Repeat: current= 7 has $degree$=2.
- But $A[2]$ is already **occupied** with a pointer to 24.
- Therefore, resolve this clash by **merging** (consolidating) trees with roots 7 and 24, and set $A[2]$ to empty.
- To maintain the (min-)heap property, root node 24 becomes the **child** of root node 7.
- 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

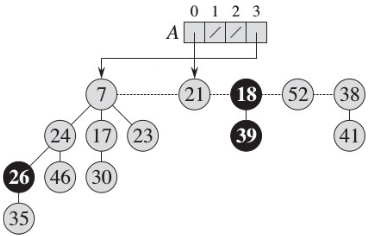
Fibonacci heaps – consolidate operation ...continued

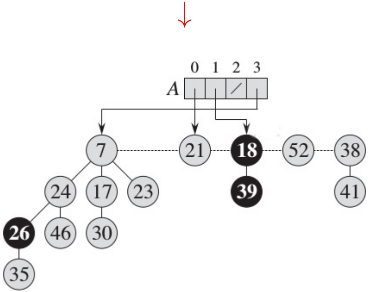




- Now, current=21 has $degree$=0.
- $A[0]$ is empty, so...
- ...get $A[0]$ to point to the root node at current=21.
- Next, move current to the right sibling, i.e current=18.

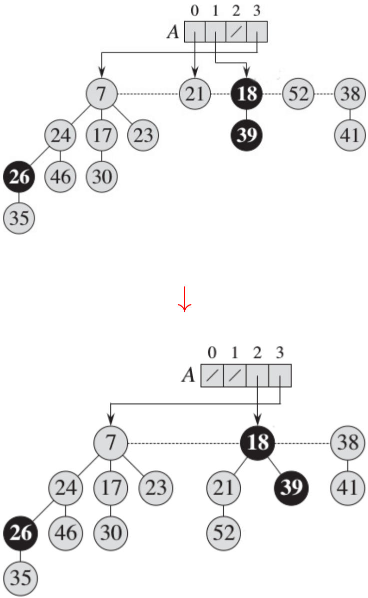
Fibonacci heaps – **consolidate** operation ...continued





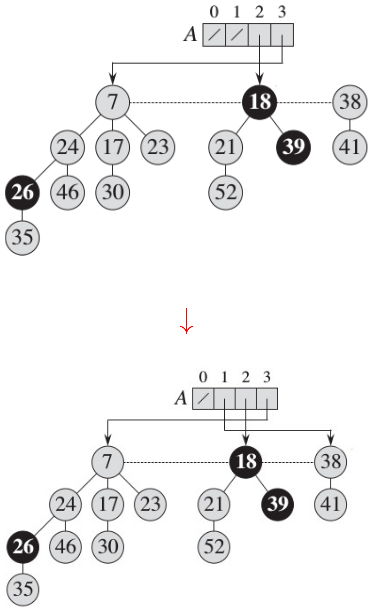
- Now, current=18 has *degree*=1.
- $A[1]$ is empty, so...
- ...get $A[1]$ to point to the root node at current=18.
- Next, move current to the right sibling, i.e current=52.

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



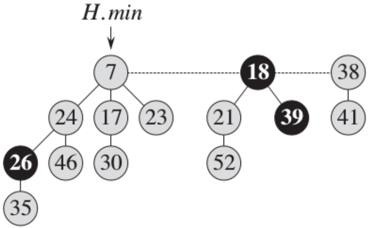
- Now, current=52 has $degree$=0, but $A[0]$ is occupied.
- So, in operations similar to those on slides #41-43...
- ...we get to the state shown in the figure on the left (below).
- current now points to root 38.

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.
- This has now completed one full cycle on the doubly linked list. STOP!

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. We will intuit why this is so, at the end.

Fibonacci heaps – **decrease-priority** operation

We want to decrease priority 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 #50)
  - case 2: When it does!
    - We will handle this over subcases, Case 2a (slide #51) and Case 2b (slide #52-54).

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

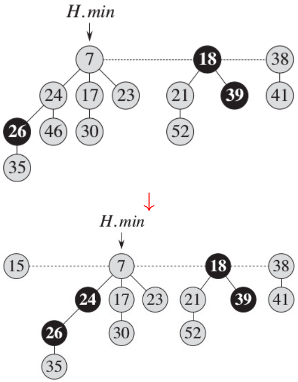
Fibonacci heaps – **decrease-priority**: Case 1

When **decrease-priority** does not violate the heap property. Simply decrease the priority 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-priority** on x assumes a pointer to x as part of its input.

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

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

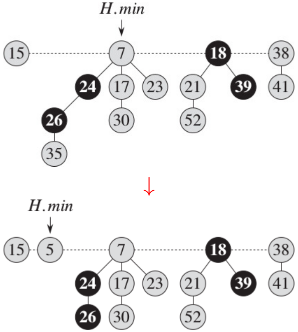


- Decreasing 46 to 15 violates the heap property...
- ...because its parent 24 $>$ 15 (previously 46).
- To address this violation:
  - Cut the subtree rooted at 15 (prev. 46)...
  - ...and promote it into the root list. (Update $H.min$, if necessary.)
  - If necessary, update the mark of 15 to 'unmarked' 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-priority** on x assumes a pointer to x as part of its input.

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

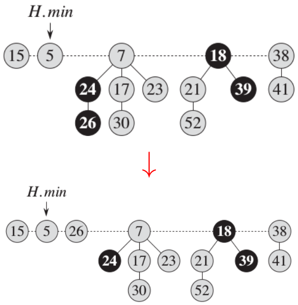
Another case arises, when the original parent of the subtree, promoted to the root level, is already ‘**marked**’ – consider the example where we want to **decrease-priority** of node (with priority=) 35 to 5.



- 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...

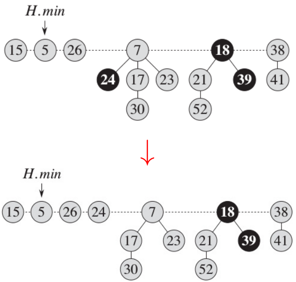
§Note: as with binary heaps, Fibonacci heaps are inefficient to **search** for any node x; For this reason, **decrease-priority** on x assumes a pointer to x as part of its input.

Fibonacci heaps – **decrease-priority**: 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-priority**: 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) of a child that is cut and promoted.

Run-time complexity of **decrease-priority** is $O(1)$ amortized.
Unfortunately, we are short of time to prove this. 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-priority** 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-priority	$O(1)$ (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=-