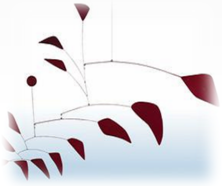
EECS 4101-5101
Advanced Data Structures



Shahin Kamali

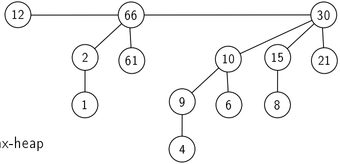
Topic 4b Fibonacci Heaps

York University

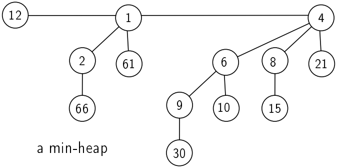
Picture is from the cover of the textbook CLRS.

Fibonacci Heaps

- Last time, we studied Binomial heaps for priority queues.
  - Insert, ExtractMax (or ExtractMin), Delete (with given pointer), and Merge all took $O(\log n)$ time.



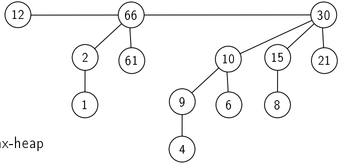
a max-heap



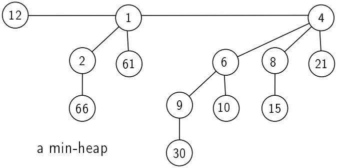
a min-heap

Fibonacci Heaps

- Last time, we studied Binomial heaps for priority queues.
  - Insert, ExtractMax (or ExtractMin), Delete (with given pointer), and Merge all took $O(\log n)$ time.



a max-heap

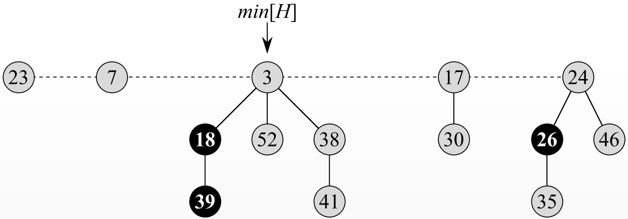


a min-heap

- Today, we study **Fibonacci Heaps** which are a more-relaxed and faster structure.
  - They support Insert and Merge in $O(1)$ and ExtractMax and Delete in $O(\log n)$ amortized time.
  - In our examples, we use min-heaps, but everything can symmetrically extend to max heaps.

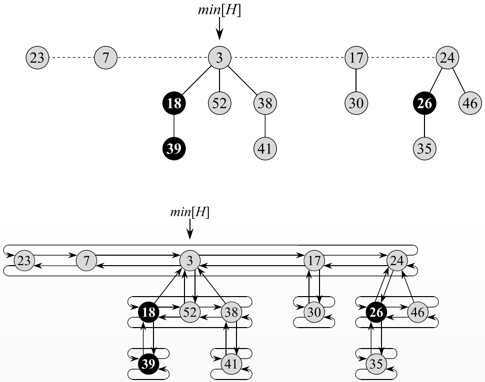
Fibonacci Heaps

- Fibonacci heaps are similar to Binomial heaps, in the sense that they are a collection of trees with the heap property
  - But the trees do not need to have any particular structure
  - The order of tree is defined by their degree, and there can be multiple trees of the same degree.
  - Nodes may be marked, indicating that they have had a child that is “lost" (moved).
  - We augment the tree with a “min" pointer.



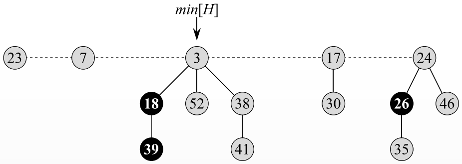
Fibonacci Heaps

- A complete implementation involves pointers to all children, parents, direct sibling, etc.
  - We omit these details in our example figures.



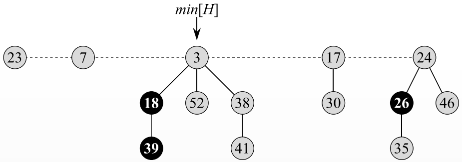
Potential Fibonacci Heaps

- Let $t(H)$ denote the number of trees and $m(H)$ denote the number of marked nodes in a Fibonacci heap H.
  We define the **potential** of H to be $\Phi(H) = t(H) + 2m(H)$.
  - E.g., er we have $t(H) = 5$, $m(H) = 3$, and $\phi(H) = 5 + 2 \cdot 3 = 11$.



Potential Fibonacci Heaps

- Let $t(H)$ denote the number of trees and $m(H)$ denote the number of marked nodes in a Fibonacci heap H.
  We define the **potential** of H to be $\Phi(H) = t(H) + 2m(H)$.
  - E.g., er we have $t(H) = 5$, $m(H) = 3$, and $\phi(H) = 5 + 2 \cdot 3 = 11$.
- Note that the potential is only used for the analysis (not the implementation).



Potential Fibonacci Heaps

- We define the amortized cost for operation t to be:

$$amortizedCost(t) = actualCost(t) + c(\cdot\Phi(t) - \Phi(t-1))$$

where c is a sufficiently large constant we define later.

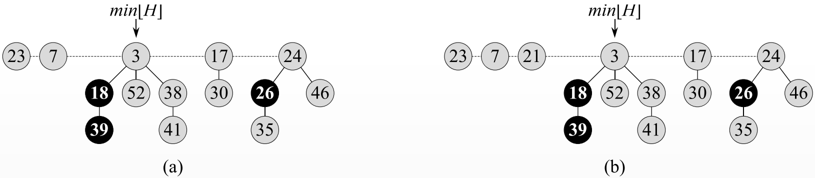
$$\begin{aligned} ActualCost &= actCost(1) + actCost(2) + \ldots + actCost(m-1) + actCost(m) \\ &= actCost(1) + c\Phi(1) - c\Phi(0) + actCost(2) + c\Phi(2) - c\Phi(1) + \ldots + actCost(m) + c\Phi(m) - c\Phi(m-1) + c\Phi(m) - c\Phi(0) \\ &= amortizedCost(1) + amortizedCost(2) + \ldots + amortizedCost(m-1) + amrotizedCost(m) + \underbrace{c(\Phi(m) - \Phi(0))}_{\textbf{a constant}}. \end{aligned}$$

- So, if we show the amortized cost for an operation is $O(f(x))$, the total cost for all m operations will be $O(mf(x))$.

Insertion Fibonacci Heaps

- To insert an element to H, just create a single node and add it as a tree (just before the $min(H)$), and update the min pointer if needed.

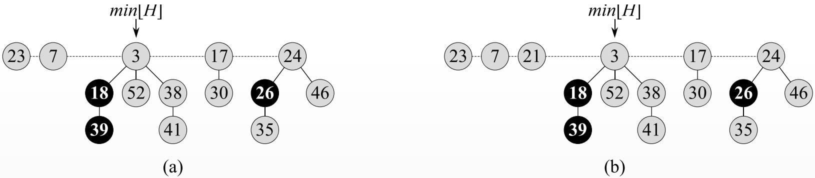
E.g., insert(21)



Insertion Fibonacci Heaps

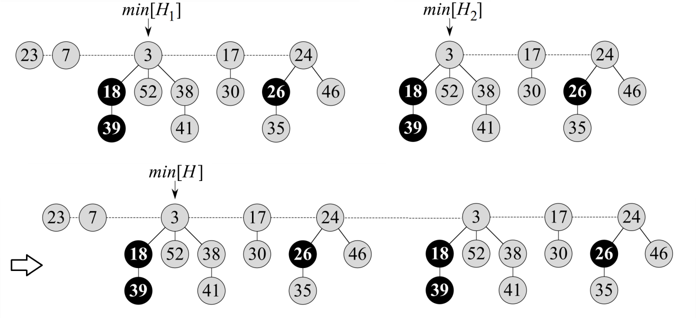
- To insert an element to H, just create a single node and add it as a tree (just before the $min(H)$), and update the min pointer if needed.
  - Actual cost is? $O(1)$
  - The number of trees $t(H)$ has increased by 1; the number of marked nodes stays unchanged $\rightarrow \Delta(\Phi) = 1$.
  - AmortizedCost = actualCost + $\Delta(\Phi) = O(1)$.

E.g., insert(21)



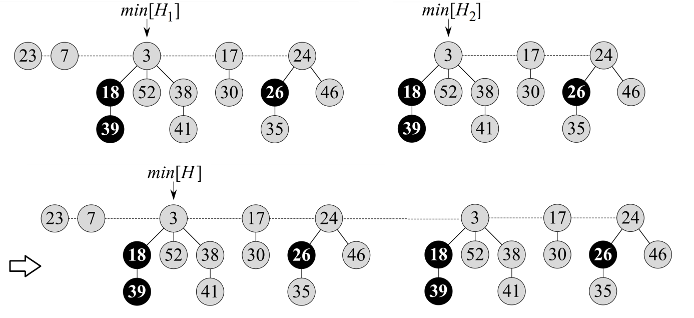
Merging Two Fibonacci Heaps

- To merge two Fibonacci heaps H_1 and H_2, We just need to update a few pointers to merge the set of trees (and also the min-pointer).



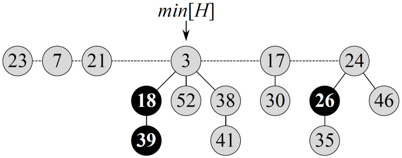
Merging Two Fibonacci Heaps

- To merge two Fibonacci heaps H_1 and H_2, We just need to update a few pointers to merge the set of trees (and also the min-pointer).
  - Actual cost is? $O(1)$
  - The number of trees $t(H)$ equals to $t(H_1) + t(H_2)$. The number of marked nodes and the potentail is not changed
  - AmortizedCost = actualCost + 0 = $O(1)$.



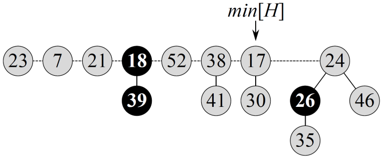
Extracting the Minimum Node

- To extract the minimum element from H, we first remove the minimum element, and add its children to the list of the trees in H.
- Go through all trees, and merge trees of the same degree (similarly to Binomial heap).
  - Maintain an array A of pointers to the trees, where $A[i]$ points to a tree of degree i.
  - Do a linear scan of the trees. When you visit a tree T with degree d, if $A[d]$ is null, let $A[d] = T$, and if $A[d]$ is not null, merge T with $A[d]$, update $A[d] = \textit{null}$ and let $A[d+1] = $ merged-tree (continue merging if $A[d+1]$ is not null).



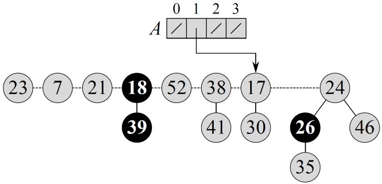
Extracting the Minimum Node

- To extract the minimum element from H, we first remove the minimum element, and add its children to the list of the trees in H.
- Go through all trees, and merge trees of the same degree (similarly to Binomial heap).
  - Maintain an array A of pointers to the trees, where $A[i]$ points to a tree of degree i.
  - Do a linear scan of the trees. When you visit a tree T with degree d, if $A[d]$ is null, let $A[d] = T$, and if $A[d]$ is not null, merge T with $A[d]$, update $A[d] = null$ and let $A[d+1] =$ merged-tree (continue merging if $A[d+1]$ is not null).



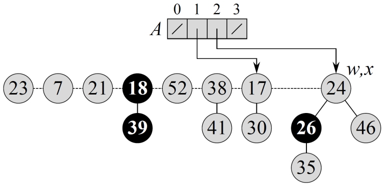
Extracting the Minimum Node

- To extract the minimum element from H, we first remove the minimum element, and add its children to the list of the trees in H.
- Go through all trees, and merge trees of the same degree (similarly to Binomial heap).
  - Maintain an array A of pointers to the trees, where $A[i]$ points to a tree of degree i.
  - Do a linear scan of the trees. When you visit a tree T with degree d, if $A[d]$ is null, let $A[d] = T$, and if $A[d]$ is not null, merge T with $A[d]$, update $A[d] = \textit{null}$ and let $A[d+1] =$ merged-tree (continue merging if $A[d+1]$ is not null).



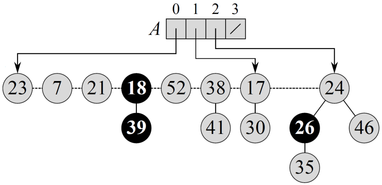
Extracting the Minimum Node

- To extract the minimum element from H, we first remove the minimum element, and add its children to the list of the trees in H.
- Go through all trees, and merge trees of the same degree (similarly to Binomial heap).
  - Maintain an array A of pointers to the trees, where $A[i]$ points to a tree of degree i.
  - Do a linear scan of the trees. When you visit a tree T with degree d, if $A[d]$ is null, let $A[d] = T$, and if $A[d]$ is not null, merge T with $A[d]$, update $A[d] = null$ and let $A[d+1] =$ merged-tree (continue merging if $A[d+1]$ is not null).



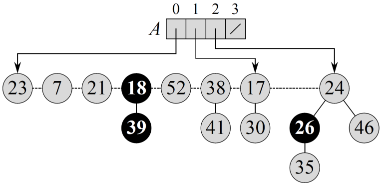
Extracting the Minimum Node

- To extract the minimum element from H, we first remove the minimum element, and add its children to the list of the trees in H.
- Go through all trees, and merge trees of the same degree (similarly to Binomial heap).
  - Maintain an array A of pointers to the trees, where $A[i]$ points to a tree of degree i.
  - Do a linear scan of the trees. When you visit a tree T with degree d, if $A[d]$ is null, let $A[d] = T$, and if $A[d]$ is not null, merge T with $A[d]$, update $A[d] = \textit{null}$ and let $A[d+1] =$ merged-tree (continue merging if $A[d+1]$ is not null).



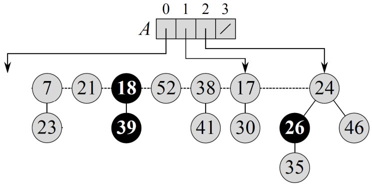
Extracting the Minimum Node

- To extract the minimum element from H, we first remove the minimum element, and add its children to the list of the trees in H.
- Go through all trees, and merge trees of the same degree (similarly to Binomial heap).
  - Maintain an array A of pointers to the trees, where $A[i]$ points to a tree of degree i.
  - Do a linear scan of the trees. When you visit a tree T with degree d, if $A[d]$ is null, let $A[d] = T$, and if $A[d]$ is not null, merge T with $A[d]$, update $A[d] = \textit{null}$ and let $A[d+1] =$ merged-tree (continue merging if $A[d+1]$ is not null).



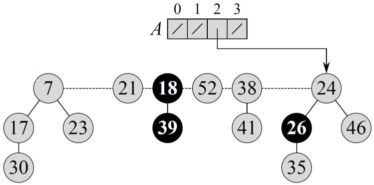
Extracting the Minimum Node

- To extract the minimum element from H, we first remove the minimum element, and add its children to the list of the trees in H.
- Go through all trees, and merge trees of the same degree (similarly to Binomial heap).
  - Maintain an array A of pointers to the trees, where $A[i]$ points to a tree of degree i.
  - Do a linear scan of the trees. When you visit a tree T with degree d, if $A[d]$ is null, let $A[d] = T$, and if $A[d]$ is not null, merge T with $A[d]$, update $A[d] = null$ and let $A[d+1] =$ merged-tree (continue merging if $A[d+1]$ is not null).



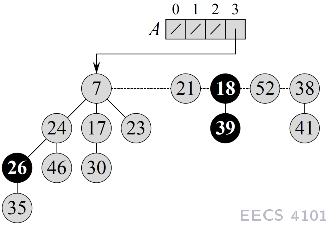
Extracting the Minimum Node

- To extract the minimum element from H, we first remove the minimum element, and add its children to the list of the trees in H.
- Go through all trees, and merge trees of the same degree (similarly to Binomial heap).
  - Maintain an array A of pointers to the trees, where $A[i]$ points to a tree of degree i.
  - Do a linear scan of the trees. When you visit a tree T with degree d, if $A[d]$ is null, let $A[d] = T$, and if $A[d]$ is not null, merge T with $A[d]$, update $A[d] = \mathit{null}$ and let $A[d+1] =$ merged-tree (continue merging if $A[d+1]$ is not null).



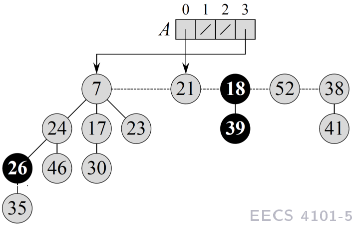
Extracting the Minimum Node

- To extract the minimum element from H, we first remove the minimum element, and add its children to the list of the trees in H.
- Go through all trees, and merge trees of the same degree (similarly to Binomial heap).
  - Maintain an array A of pointers to the trees, where $A[i]$ points to a tree of degree i.
  - Do a linear scan of the trees. When you visit a tree T with degree d, if $A[d]$ is null, let $A[d] = T$, and if $A[d]$ is not null, merge T with $A[d]$, update $A[d] = \textit{null}$ and let $A[d+1] =$ merged-tree (continue merging if $A[d+1]$ is not null).



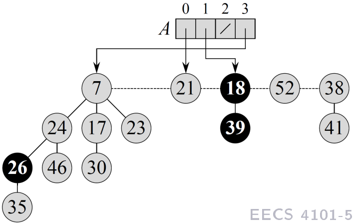
Extracting the Minimum Node

- To extract the minimum element from H, we first remove the minimum element, and add its children to the list of the trees in H.
- Go through all trees, and merge trees of the same degree (similarly to Binomial heap).
  - Maintain an array A of pointers to the trees, where $A[i]$ points to a tree of degree i.
  - Do a linear scan of the trees. When you visit a tree T with degree d, if $A[d]$ is null, let $A[d] = T$, and if $A[d]$ is not null, merge T with $A[d]$, update $A[d] = null$ and let $A[d+1] =$ merged-tree (continue merging if $A[d+1]$ is not null).



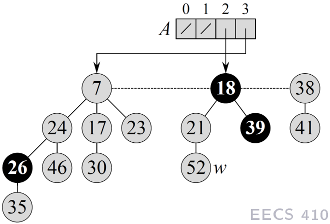
Extracting the Minimum Node

- To extract the minimum element from H, we first remove the minimum element, and add its children to the list of the trees in H.
- Go through all trees, and merge trees of the same degree (similarly to Binomial heap).
  - Maintain an array A of pointers to the trees, where $A[i]$ points to a tree of degree i.
  - Do a linear scan of the trees. When you visit a tree T with degree d, if $A[d]$ is null, let $A[d] = T$, and if $A[d]$ is not null, merge T with $A[d]$, update $A[d] = \textit{null}$ and let $A[d+1] =$ merged-tree (continue merging if $A[d+1]$ is not null).



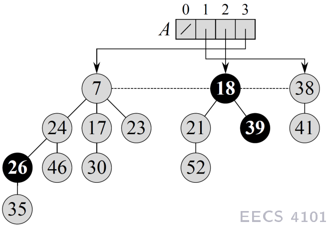
Extracting the Minimum Node

- To extract the minimum element from H, we first remove the minimum element, and add its children to the list of the trees in H.
- Go through all trees, and merge trees of the same degree (similarly to Binomial heap).
    - Maintain an array A of pointers to the trees, where $A[i]$ points to a tree of degree i.
    - Do a linear scan of the trees. When you visit a tree T with degree d, if $A[d]$ is null, let $A[d] = T$, and if $A[d]$ is not null, merge T with $A[d]$, update $A[d] = \textit{null}$ and let $A[d+1] =$ merged-tree (continue merging if $A[d+1]$ is not null).



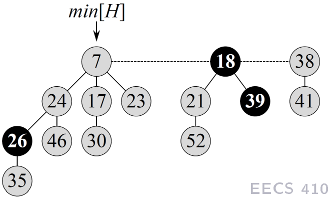
Extracting the Minimum Node

- To extract the minimum element from H, we first remove the minimum element, and add its children to the list of the trees in H.
- Go through all trees, and merge trees of the same degree (similarly to Binomial heap).
  - Maintain an array A of pointers to the trees, where $A[i]$ points to a tree of degree i.
  - Do a linear scan of the trees. When you visit a tree T with degree d, if $A[d]$ is null, let $A[d] = T$, and if $A[d]$ is not null, merge T with $A[d]$, update $A[d] = \mathit{null}$ and let $A[d+1] =$ merged-tree (continue merging if $A[d+1]$ is not null).



Extracting the Minimum Node

- To extract the minimum element from H, we first remove the minimum element, and add its children to the list of the trees in H.
- Go through all trees, and merge trees of the same degree (similarly to Binomial heap).
  - Maintain an array A of pointers to the trees, where $A[i]$ points to a tree of degree i.
  - Do a linear scan of the trees. When you visit a tree T with degree d, if $A[d]$ is null, let $A[d] = T$, and if $A[d]$ is not null, merge T with $A[d]$, update $A[d] = null$ and let $A[d+1] =$ merged-tree (continue merging if $A[d+1]$ is not null).



Fibonacci Sequence Background

- Recall that $F_1 = 1, F_2 = 1, F_3 = 2, \ldots, F_i = F_{i-1} + F_{i-2}$.

Fibonacci Sequence Background

- Recall that $F_1 = 1, F_2 = 1, F_3 = 2, \ldots, F_i = F_{i-1} + F_{i-2}$.
- We can use induction to show $1 + F_1 + F_2 + \ldots + F_i = F_{i+2}$.
  - Base: $1 + F_1 = 1 + 1 = 2 = F_3$.
  - Induction step: $(1 + F_1 + F_2 + \ldots + F_{i-1}) + F_i = F_{i+1} + F_i = F_{i+2}$.

Fibonacci Sequence Background

- Recall that $F_1 = 1, F_2 = 1, F_3 = 2, \ldots, F_i = F_{i-1} + F_{i-2}$.
- We can use induction to show $1 + F_1 + F_2 + \ldots + F_i = F_{i+2}$.
  - Base: $1 + F_1 = 1 + 1 = 2 = F_3$.
  - Induction step: $(1 + F_1 + F_2 + \ldots + F_{i-1}) + F_i = F_{i+1} + F_i = F_{i+2}$.
- Asymptotically, we have $F_n = \Theta(\Phi^n)$, where $\Phi = \frac{1+\sqrt{5}}{2} \approx 1.618$ is the golden ratio.

Extracting the Minimum Node

- Let $N(d)$ = min. number of nodes in a single tree T with degree d at the root. What is $N(d)$?
- We use induction to show $N(d) \geq F_{d+2}$.
  - In the base, when $d = 0$, we have $N(d) = 1 = F_2$.

Extracting the Minimum Node

- Let $N(d)$ = min. number of nodes in a single tree T with degree d at the root. What is $N(d)$?
- We use induction to show $N(d) \geq F_{d+2}$.
  - In the base, when $d = 0$, we have $N(d) = 1 = F_2$.
  - Let T be a tree with minimum number of nodes $N(d)$. Sort subtrees of the root of T by their degree as $T_1, \ldots, T_d$; let that c_i denote the degree of T_i. We have $c_1 \geq 0$.
  - For $i \geq 2$, the tree T_i at some point is merged by the tree formed by the root at $T_1, \ldots, T_{i-1}$; at the time of the merger, the degree of T_i had been $i - 1$ (why?); It is possible that T_i lost a child after and its degree became $i - 2$.

Extracting the Minimum Node

- Let $N(d)$ = min. number of nodes in a single tree T with degree d at the root. What is $N(d)$?
- We use induction to show $N(d) \geq F_{d+2}$.
  - In the base, when $d = 0$, we have $N(d) = 1 = F_2$.
  - Let T be a tree with minimum number of nodes $N(d)$. Sort subtrees of the root of T by their degree as $T_1, \ldots, T_d$; let that c_i denote the degree of T_i. We have $c_1 \geq 0$.
  - For $i \geq 2$, the tree T_i at some point is merged by the tree formed by the root at $T_1, \ldots, T_{i-1}$; at the time of the merger, the degree of T_i had been $i - 1$ (why?); It is possible that T_i lost a child after and its degree became $i - 2$.
  - So, by inductive hypothesis, for $i \geq 2$, we have $Size(T_i) \geq F_{(i-2)+2} = F_i$.

Extracting the Minimum Node

- Let $N(d)$ = min. number of nodes in a single tree T with degree d at the root. What is $N(d)$?
- We use induction to show $N(d) \geq F_{d+2}$.
  - In the base, when $d = 0$, we have $N(d) = 1 = F_2$.
  - Let T be a tree with minimum number of nodes $N(d)$. Sort subtrees of the root of T by their degree as $T_1, \ldots, T_d$; let that c_i denote the degree of T_i. We have $c_1 \geq 0$.
  - For $i \geq 2$, the tree T_i at some point is merged by the tree formed by the root at $T_1, \ldots, T_{i-1}$; at the time of the merger, the degree of T_i had been $i - 1$ (why?); It is possible that T_i lost a child after and its degree became $i - 2$.
  - So, by inductive hypothesis, for $i \geq 2$, we have $Size(T_i) \geq F_{(i-2)+2} = F_i$.
  - The size of T is $N(d) = 1 + Size(T_1) + F_2 + F_3 + \ldots F_d = 1 + \sum_{i=0}^{d} F_i = F_{d+2}$.

Extracting the Minimum Node

- Let $N(d)$ = min. number of nodes in a single tree T with degree d at the root. What is $N(d)$?
- We use induction to show $N(d) \geq F_{d+2}$.
  - In the base, when $d = 0$, we have $N(d) = 1 = F_2$.
  - Let T be a tree with minimum number of nodes $N(d)$. Sort subtrees of the root of T by their degree as $T_1, \ldots, T_d$; let that c_i denote the degree of T_i. We have $c_1 \geq 0$.
  - For $i \geq 2$, the tree T_i at some point is merged by the tree formed by the root at $T_1, \ldots, T_{i-1}$; at the time of the merger, the degree of T_i had been $i - 1$ (why?); It is possible that T_i lost a child after and its degree became $i - 2$.
  - So, by inductive hypothesis, for $i \geq 2$, we have $Size(T_i) \geq F_{(i-2)+2} = F_i$.
  - The size of T is $N(d) = 1 + Size(T_1) + F_2 + F_3 + \ldots F_d = 1 + \sum_{i=0}^{d} F_i = F_{d+2}$.
- **Fact 1: The number of nodes in a tree of degree d after merging (extracting min) is at least $N(d) = F_{d+2}$.**

Extracting the Minimum Node

- Let $D(m)$ = max. degree of any node in a single tree T with m nodes right after extractMin. What is $D(m)$?
- Let $N(d)$ = min. number of nodes in a single tree T with degree d at the root. We just used induction to show $N(d) \geq F_{d+2}$.
- Therefore, we have $N(d) \geq F_{d+2} \in \Theta(\Phi^{d+2})$ or $\log(N(d)) \in \Omega(d+2)$, or $d \in O(\log N(d))$. Equivalently $D(m) \in O(\log n)$.
- **Fact 2: The degree of any tree in a Fibonacci heap at any time is at most $D(m) = O(\log n)$.**
  - After the merger, the degrees never increase until the next merge.

Extracting the Minimum Node

- Let $P(n)$ = maximum number of trees after the merger.
- We show that $P(n) \in O(\log n)$.
  - No trees have the same degree (why?) and each tree with degree d has at least $N(d) = F_{d+2}$ nodes (Fact 1).
  - The total number of nodes is thus $n \geq F_1 + \ldots + F_{P(n)} = F_{P(n)+2} - 1$, that is $P(n) \in O(\log n)$

Extracting the Minimum Node

- Let $P(n)$ = maximum number of trees after the merger.
- We show that $P(n) \in O(\log n)$.
  - No trees have the same degree (why?) and each tree with degree d has at least $N(d) = F_{d+2}$ nodes (Fact 1).
  - The total number of nodes is thus $n \geq F_1 + \ldots + F_{P(n)} = F_{P(n)+2} - 1$, that is $P(n) \in O(\log n)$
- **Fact 3: The number of trees in a Fibonacci tree right after merging (extracting min) is at most $P(n) = O(\log n)$.**

Extracting the Minimum Node

- **Fact 1: The number of nodes in a tree of degree d after merging (extracting min) is at least $N(d) = F_{d+2}$.**
- **Fact 2: The degree of any tree in a Fibonacci heap at any time is at most $D(m) = O(\log n)$.**
- **Fact 3: The number of trees in a Fibonacci tree right after merging (extracting min) is at most $P(n) = O(\log n)$.**

Extracting the Minimum node

- For the time complexity, we consider these notations:
  - Let $D(n)$ = max degree of any node in a single tree with n nodes.
  - Let $t(H)$ = number of trees in the heap H before the merger.
  - $m(H)$ = number of marked nodes in the heap H
  - Potential function $\Phi(H) = t(H) + 2m(H)$

Extracting the Minimum node

- For the time complexity, we consider these notations:
  - Let $D(n)$ = max degree of any node in a single tree with n nodes.
  - Let $t(H)$ = number of trees in the heap H before the merger.
  - $m(H)$ = number of marked nodes in the heap H
  - Potential function $\Phi(H) = t(H) + 2m(H)$
- Actual cost is $O(\log n + t(H))$
  - $O(D(n)) = O(\log n)$ work adding min's children into root list and $t(H)$ for the linear scan (each merger takes constant time).
- The number of trees is $t(H)$ before extractMin and at most $O(\log n)$ after the merger (Fact 3). The number of marked nodes does not change. So, we can write $\Delta(\Phi) \leq O(\log n) + 1 - t(H)$.

Extracting the Minimum node

- For the time complexity, we consider these notations:
  - Let $D(n)$ = max degree of any node in a single tree with n nodes.
  - Let $t(H)$ = number of trees in the heap H before the merger.
  - $m(H)$ = number of marked nodes in the heap H
  - Potential function $\Phi(H) = t(H) + 2m(H)$
- Actual cost is $O(\log n + t(H))$
  - $O(D(n)) = O(\log n)$ work adding min's children into root list and $t(H)$ for the linear scan (each merger takes constant time).
- The number of trees is $t(H)$ before extractMin and at most $O(\log n)$ after the merger (Fact 3). The number of marked nodes does not change. So, we can write $\Delta(\Phi) \leq O(\log n) + 1 - t(H)$.
- The amortized cost will be $actualCost + c \cdot \Delta(\Phi) = O(\log n + t(H)) + c(\log n + 1 - t(H)) = O(\log n)$, assuming c is selected be large enough.

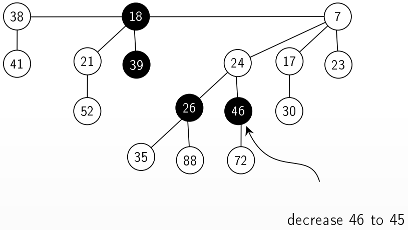
Extracting the Minimum node

Theorem

The amortized running time of extractMax in a Fibonacci heap with n keys is $O(\log n)$.

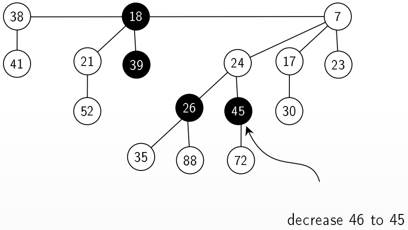
Decreasing Key

- Given a pointer to an element x, decrease key of x to k
- **Case 0: min-heap property not violated**
  - Decrease key of x to k.
  - The actual cost is $O(1)$, the potential is not changed $\rightarrow$ the amortized cost is $O(1)$.



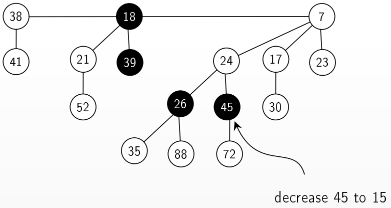
Decreasing Key

- Given a pointer to an element x, decrease key of x to k
- **Case 0: min-heap property not violated**
  - Decrease key of x to k.
  - The actual cost is $O(1)$, the potential is not changed $\rightarrow$ the amortized cost is $O(1)$.



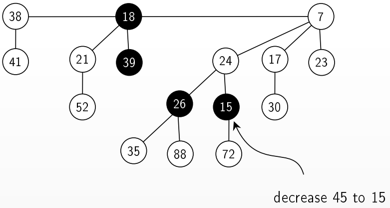
Decreasing Key

- Given a pointer to an element x, decrease key of x to k
- **Case 1: parent of x is unmarked**
  - Decrease key of x to k.
  - Cut off link between x and its parent, unmark x if marked, and mark parent of x.
  - Add tree rooted at x to root list, updating heap min pointer if needed.
  - The actual cost is $O(1)$, $t(h)$ is incremented and $m(h)$ is increased by at most 1 $\rightarrow \Delta(\Phi) \leq 3 \rightarrow$ amortized cost is $O(1)$.



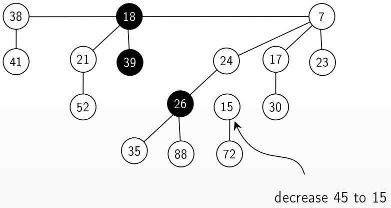
Decreasing Key

- Given a pointer to an element x, decrease key of x to k
- **Case 1: parent of x is unmarked**
  - Decrease key of x to k.
  - Cut off link between x and its parent, unmark x if marked, and mark parent of x.
  - Add tree rooted at x to root list, updating heap min pointer if needed.
  - The actual cost is $O(1)$, $t(h)$ is incremented and $m(h)$ is increased by at most 1 $\rightarrow \Delta(\Phi) \leq 3 \rightarrow$ amortized cost is $O(1)$.



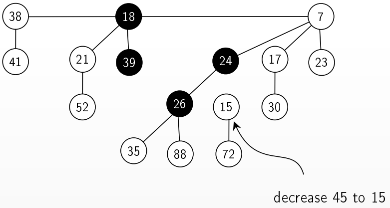
Decreasing Key

- Given a pointer to an element x, decrease key of x to k
- **Case 1: parent of x is unmarked**
    - Decrease key of x to k.
    - Cut off link between x and its parent, unmark x if marked, and mark parent of x.
    - Add tree rooted at x to root list, updating heap min pointer if needed.
    - The actual cost is $O(1)$, $t(h)$ is incremented and $m(h)$ is increased by at most 1 $\rightarrow \Delta(\Phi) \leq 3 \rightarrow$ amortized cost is $O(1)$.



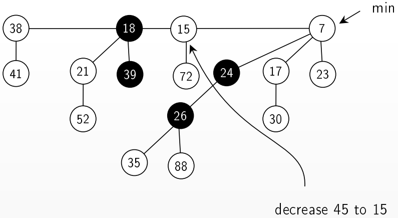
Decreasing Key

- Given a pointer to an element x, decrease key of x to k
- **Case 1: parent of x is unmarked**
  - Decrease key of x to k.
  - Cut off link between x and its parent, unmark x if marked, and mark parent of x.
  - Add tree rooted at x to root list, updating heap min pointer if needed.
  - The actual cost is $O(1)$, $t(h)$ is incremented and $m(h)$ is increased by at most 1 $\rightarrow \Delta(\Phi) \leq 3 \rightarrow$ amortized cost is $O(1)$.



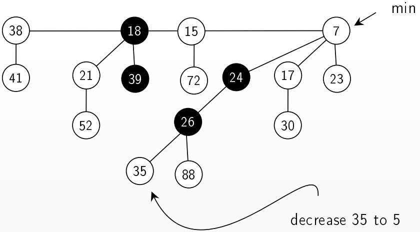
Decreasing Key

- Given a pointer to an element x, decrease key of x to k
- **Case 1: parent of x is unmarked**
  - Decrease key of x to k.
  - Cut off link between x and its parent, unmark x if marked, and mark parent of x.
  - Add tree rooted at x to root list, updating heap min pointer if needed.
  - The actual cost is $O(1)$, $t(h)$ is incremented and $m(h)$ is increased by at most 1 $\rightarrow \Delta(\Phi) \leq 3 \rightarrow$ amortized cost is $O(1)$.



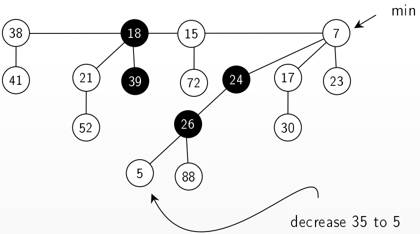
Decreasing Key

- Given a pointer to an element x, decrease key of x to k
- **Case 2: parent of x is marked**
  - Decrease key of x to k.
  - Cut off link between x and its parent $p[x]$, unmark x if marked, and add it to the list of the trees.
  - cut off link between $p[x]$ and $p[p[x]]$, add $p[x]$ to the list, unmark $p[x]$
    - If $p[p[x]]$ unmarked, then mark it and stop
    - If $p[p[x]]$ marked, cut off $p[p[x]]$, unmark, and repeat until unmarked node found or root reached



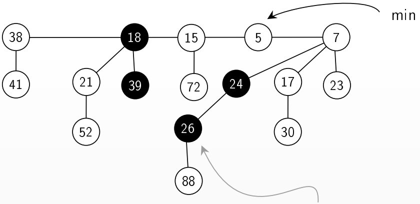
Decreasing Key

- Given a pointer to an element x, decrease key of x to k
- **Case 2: parent of x is marked**
  - Decrease key of x to k.
  - Cut off link between x and its parent $p[x]$, unmark x if marked, and add it to the list of the trees.
  - cut off link between $p[x]$ and $p[p[x]]$, add $p[x]$ to the list, unmark $p[x]$
    - If $p[p[x]]$ unmarked, then mark it and stop
    - If $p[p[x]]$ marked, cut off $p[p[x]]$, unmark, and repeat until unmarked node found or root reached



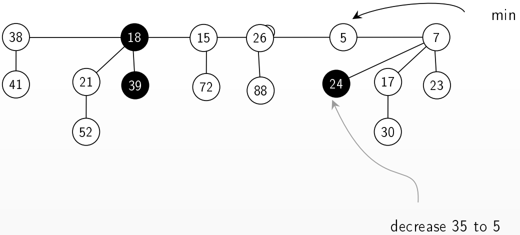
Decreasing Key

- Given a pointer to an element x, decrease key of x to k
- **Case 2: parent of x is marked**
  - Decrease key of x to k.
  - Cut off link between x and its parent $p[x]$, unmark x if marked, and add it to the list of the trees.
  - cut off link between $p[x]$ and $p[p[x]]$, add $p[x]$ to the list, unmark $p[x]$
    - If $p[p[x]]$ unmarked, then mark it and stop
    - If $p[p[x]]$ marked, cut off $p[p[x]]$, unmark, and repeat until unmarked node found or root reached



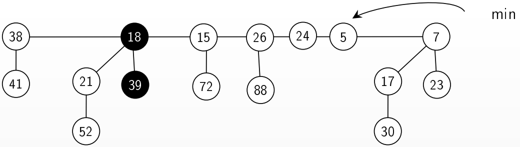
Decreasing Key

- Given a pointer to an element x, decrease key of x to k
- **Case 2: parent of x is marked**
  - Decrease key of x to k.
  - Cut off link between x and its parent $p[x]$, unmark x if marked, and add it to the list of the trees.
  - cut off link between $p[x]$ and $p[p[x]]$, add $p[x]$ to the list, unmark $p[x]$
    - If $p[p[x]]$ unmarked, then mark it and stop
    - If $p[p[x]]$ marked, cut off $p[p[x]]$, unmark, and repeat until unmarked node found or root reached



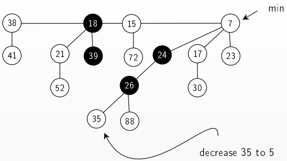
Decreasing Key

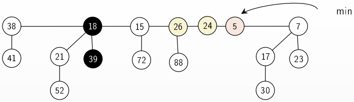
- Given a pointer to an element x, decrease key of x to k
- **Case 2: parent of x is marked**
  - Decrease key of x to k.
  - Cut off link between x and its parent $p[x]$, unmark x if marked, and add it to the list of the trees.
  - cut off link between $p[x]$ and $p[p[x]]$, add $p[x]$ to the list, unmark $p[x]$
    - If $p[p[x]]$ unmarked, then mark it and stop
    - If $p[p[x]]$ marked, cut off $p[p[x]]$, unmark, and repeat until unmarked node found or root reached



Decreasing Key

- Given a pointer to an element x, decrease key of x to k
- **Case 2: parent of x is marked**
  - Suppose p new trees are added here $\rightarrow t(h)$ is increased by p.
  - Roots of all these trees, except possibly the first one, have been marked before the operation and are unmarked after $\rightarrow m(h)$ is decremented by at least $p-1$.
  - Potential changes from $t(h)+2m(h)$ to at most $(t(h)+p)+2(m(h)-(p-1)) = t(h)+2m(h)-p+2$. That is $\Delta(\Phi) \leq 2-p$.
  - The actual cost is $O(p)$ (why?), and the amortized cost will be $O(p)+c(2-p) = O(1)$ for sufficiently large c.





Extracting the Minimum node

Theorem

The amortized running time of extractMin in a Fibonacci heap with n keys is $O(\log n)$.

Theorem

The amortized running time of decreaseKey in a Fibonacci heap with n keys is $O(1)$.

Extracting the Minimum node

Theorem

The amortized running time of extractMin in a Fibonacci heap with n keys is $O(\log n)$.

Theorem

The amortized running time of decreaseKey in a Fibonacci heap with n keys is $O(1)$.

- To delete an item (with a pointer to it), simply decrease its key to $-\infty$ and call extractMin. This runs in $O(\log n)$ amortized time.

Theorem

The amortized running time of delete in a Fibonacci heap with n keys is $O(\log n)$.

Data Structures for Priority Queues

- A summary of data structures for priority queues.

Operation	Binary heap (worst-case)	Binomial heap (worst-case)	Fibonacci heap (amortized)
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$	$\Theta(1))$
INSERT	$\Theta(\log n)$	$O(\log n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$O(\log n)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$
MERGE/UNION	$\Theta(n)$	$O(\log n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
DELETE	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$