Fibonacci Heaps

Christian Wulff-Nilsen

Seventh lecture

Algorithms and Data Structures

DIKU

February 27, 2023

Overview for today

- Introduction and comparison to binary heaps
- The overall structure of a Fibonacci heap
- Maintaining a Fibonacci heap and bounding the amortized update time for each type of operation

What is a min-heap?

- We consider a min-heap which is a data structure containing elements with real-numbered keys
- The following types of operations are supported:
  - $\texttt{Make-Heap}()$: returns a new empty heap
  - $\texttt{Insert}(H, x)$: inserts element x into heap H
  - $\texttt{Minimum}(H)$: returns pointer to element in H with min key
  - $\texttt{Extract-Min}(H)$: deletes min-key element from H and returns pointer to it
  - $\texttt{Union}(H_1, H_2)$: returns a new heap whose set of elements is the union of the sets of elements of heaps H_1 and H_2; H_1 and H_2 are destroyed in the process
  - $\texttt{Decrease-Key}(H, x, k)$: decreases the key of element x in heap H to the new value k
  - $\texttt{Delete}(H, x)$: deletes element x from heap H

Binary heaps compared to Fibonacci heaps

- Consider n operations applied to an initially empty heap
- Performance comparison for binary heaps and Fibonacci heaps:

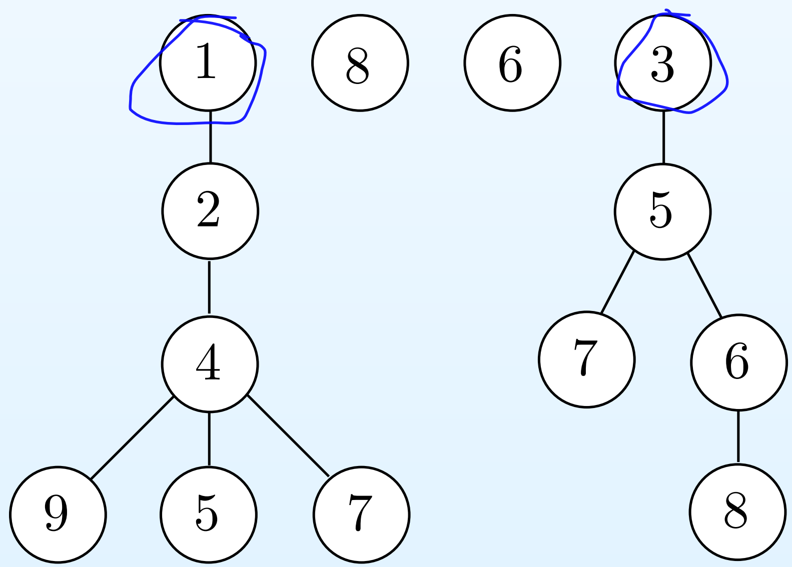
Operation type	Binary heap	Fibonacci heap
`Make-Heap`	$O(1)$	$O(1)$
`Insert`	$O(\lg n)$	$O(1)$
`Minimum`	$O(1)$	$O(1)$
`Extract-Min`	$O(\lg n)$	$O(\lg n)$
`Union`	$O(n)$	$O(1)$
`Decrease-Key`	$O(\lg n)$	$O(1)$
`Delete`	$O(\lg n)$	$O(\lg n)$

- The bounds for binary heaps are worst-case whereas the bounds for Fibonacci heaps are amortized
- The improvement for `Decrease-Key` is important in Dijkstra's and Prim's algorithms (presented later in the course).

Structure of a Fibonacci heap

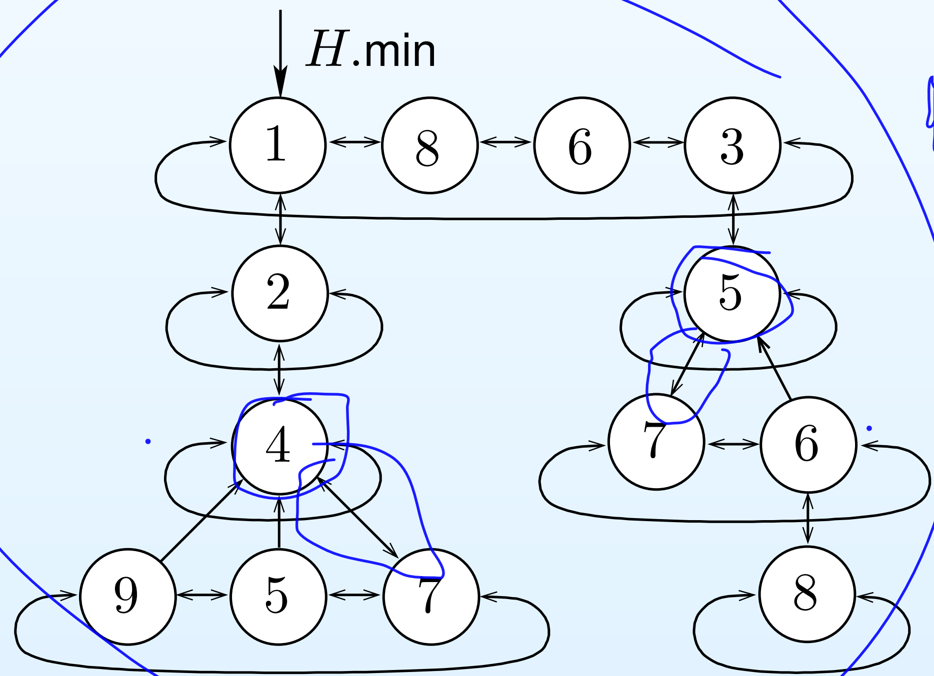
- A Fibonacci heap consists of a collection of rooted trees where each node x has a key $\mathrm{key}(x) \in \mathbb{R}$
- Each tree has the *min-heap property*: for each node x having a parent p, $\mathrm{key}(p) \le \mathrm{key}(x)$
- Thus, one of the root nodes contains a key of minimum value among all nodes of the heap

min-heap



Pointers in a Fibonacci heap H

- The root nodes are connected in a circular, doubly linked list
- Each node has a pointer to its parent (if any)
- Each node has a pointer to a single one of its children (if any)
- Sibling nodes are connected in a circular, doubly linked list
- Finally, there is a pointer H.min to a root node containing a key of minimum value among all nodes of H

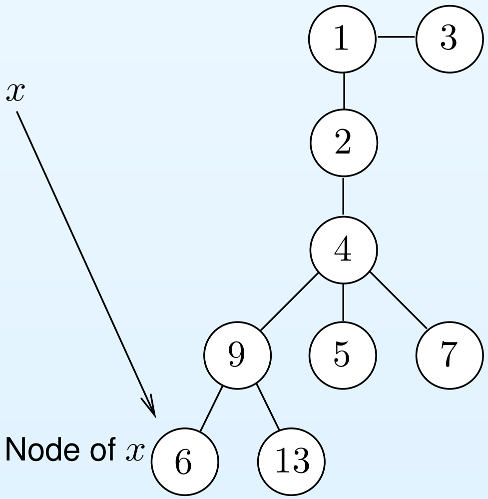


Node attributes

- Each node x of a Fibonacci heap has the following auxiliary data (in addition to $\mathrm{key}(x)$ and pointers to other nodes):
  - x.deg: the number of children of x
  - x.mark: a bit indicating whether x is marked (more on this later)

Obtaining the node of an element

- We will often refer to an inserted element x and its node in the Fibonacci heap H as if they were the same
- When a user executes, e.g., $\texttt{Delete}(H, x)$, we require that its node can be obtained in $O(1)$ worst-case time
- This can be ensured as follows:
  - When x is inserted, its node is stored in some memory location
  - This memory location does not change when H changes
  - A pointer is kept from x to its node's fixed memory location
  - It allows constant-time access to the node of x



The potential function

- Let H be a Fibonacci heap
- We use the potential method from the previous lecture to bound the amortized update time of H
- Let $t(H)$ denote the number of trees of H, i.e., the size of the root node list
- Let $m(H)$ denote the number of marked nodes of H
- Define the potential function $\Phi(H)$ by

$$\Phi(H) = t(H) + 2m(H)$$

- Here, we allow H to represent a collection of Fibonacci heaps (in case of multiple calls to `Make-Heap`)
- In that case, $\Phi(H)$ is the sum of potentials of each Fibonacci heap in this collection

The potential function is valid

- We defined $\Phi(H)$ by

$$\Phi(H) = t(H) + 2m(H)$$

- Why is this a valid potential function?
  - Let H_0 be the initial empty data structure and let H_i denote H just after the ith operation, $i > 0$
  - Φ is valid since $\Phi(H_0) = 0$ and $\Phi(H_i) \geq 0$ for all $i \geq 0$

Amortized cost

- Recall that the amortized cost of the ith operation is
  $\hat{c}_i = c_i + \Phi(H_i) - \Phi(H_{i-1})$ $+\Phi r$
- In the following, we consider the ith operation and let H denote the heap just prior to the operation and let H' denote the heap just after the operation $+\Phi r$
- Thus, $H = H_{i-1}$, $H' = H_i$, and $\hat{c}_i = c_i + \Phi(H') - \Phi(H)$

The `Make-Heap` operation

- Suppose the ith operation is `Make-Heap`
- This operation simply returns an empty heap which can be done in $c_i = O(1)$ time
- The potential is unchanged by this operation so

$$\hat{c}_i = c_i + \Phi(H') - \Phi(H) = c_i = O(1)$$

The `Insert` operation

- Suppose the ith operation is $\texttt{Insert}(H, x)$
- It works as follows:
  - x.mark $=$ false
  - If H is empty, make a root list containing just x and let H.min $= x$
  - Otherwise, obtain $r = H$.min and add x as a neighbor to r in the root list; update H.min to x if $\mathrm{key}(x) < \mathrm{key}(r)$
- $\texttt{Insert}(H, x)$ can be executed in $c_i = O(1)$ time
- The amortized cost of $\texttt{Insert}(H, x)$ is

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(H') - \Phi(H) \\ &= c_i + t(H') + 2m(H') - (t(H) + 2m(H)) \\ &= c_i + 1 = O(1)\end{aligned}$$

The `Minimum` operation

- Suppose the ith operation is $\texttt{Minimum}(H)$
- This operation simply returns H.min so $c_i = O(1)$
- Since the potential does not change, $\hat{c}_i = c_i = O(1)$

The `Extract-Min` operation

- Suppose the ith operation is $\texttt{Extract-Min}(H)$
- This operation works as follows:
  - Remove $r = H$.min from the root list
  - Add all children of r to the root list
  - Apply `Consolidate` to the updated heap (next slide)
  - Finally, return r

The `Consolidate` operation

- Let $D(n) \in \mathbb{N}$ denote an upper bound on the maximum number of children of a node in any n-node heap
- We later show that we can choose $D(n) = \Theta(\lg n)$
- `Consolidate` initializes an array of NIL pointers, $A[0, \ldots, D(n)]$, where n is the number of nodes of H
- It then uses A to iteratively pair up trees whose roots have the same degree
- Two trees are paired up by attaching one root r_1 as a child of the other root r_2 such that $\mathrm{key}(r_1) \geq \mathrm{key}(r_2)$ (ensuring the min-heap property)
- At termination, all roots have distinct degrees
- These roots are scanned to find the new H.min

Illustration of `Extract-Min`, including `Consolidate`

- First, `Extract-Min`(H) deletes H.min and adds its children as root nodes:

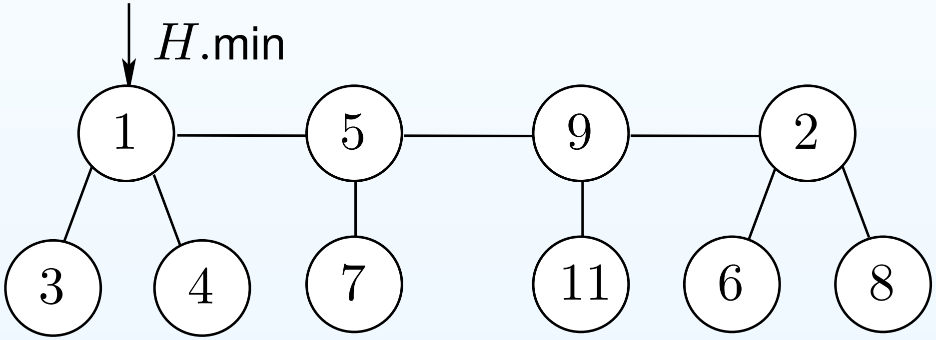


Illustration of `Extract-Min`, including `Consolidate`

- First, $\texttt{Extract-Min}(H)$ deletes H.min and adds its children as root nodes:

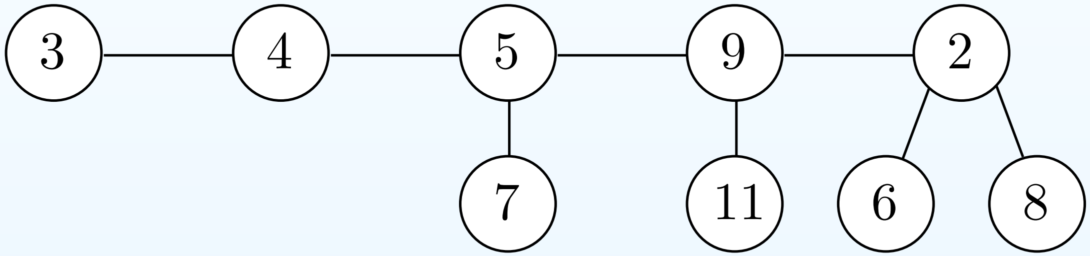


Illustration of `Extract-Min` including `Consolidate`

- Then `Consolidate`(H) is applied (the red node is the root currently being processed):

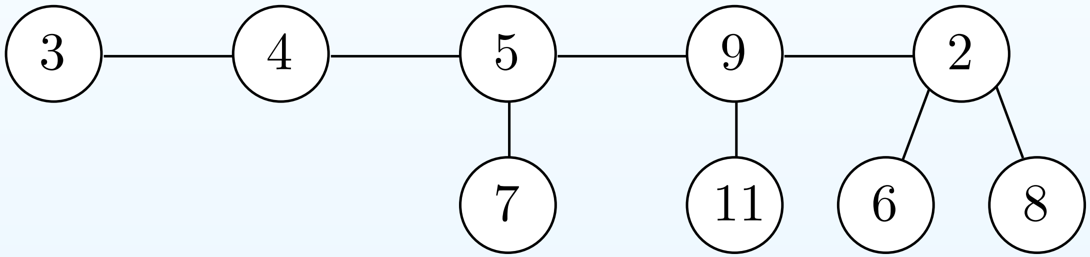
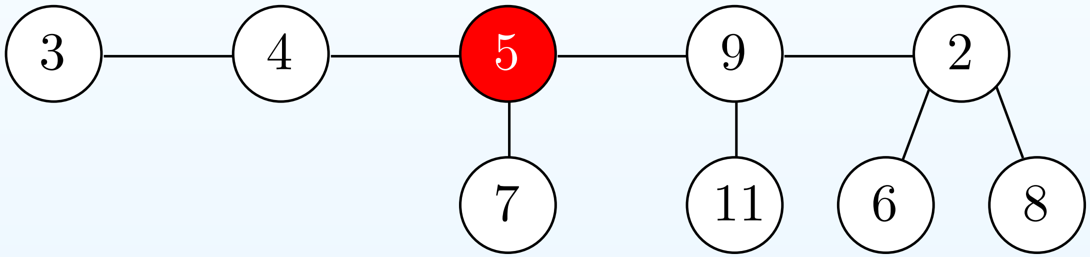


Illustration of `Extract-Min` including `Consolidate`

- Then `Consolidate`(H) is applied (the red node is the root currently being processed):

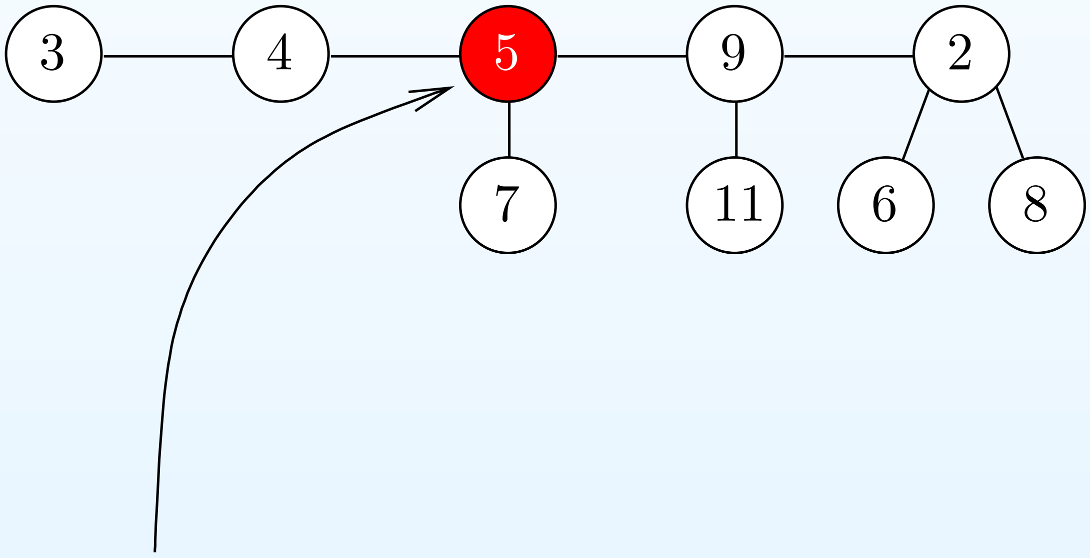


Current node has degree 1 so add pointer from $A[1]$

$$A[0, 1, 2, 3, \ldots, D(n)]$$

Illustration of `Extract-Min` including `Consolidate`

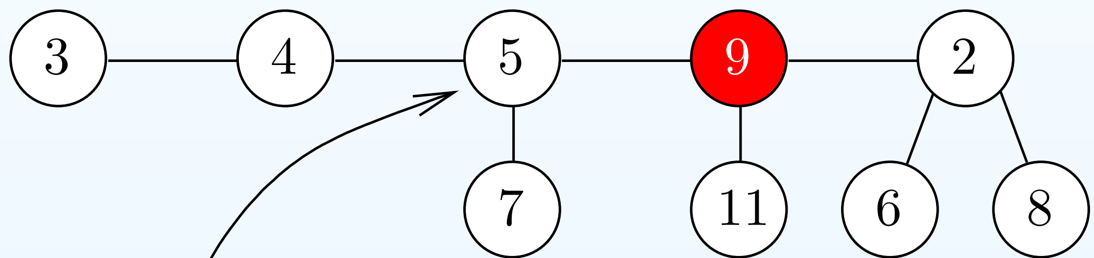
- Then `Consolidate`(H) is applied (the red node is the root currently being processed):



$$A[0, 1, 2, 3, \ldots, D(n)]$$

Illustration of `Extract-Min` including `Consolidate`

- Then `Consolidate`(H) is applied (the red node is the root currently being processed):

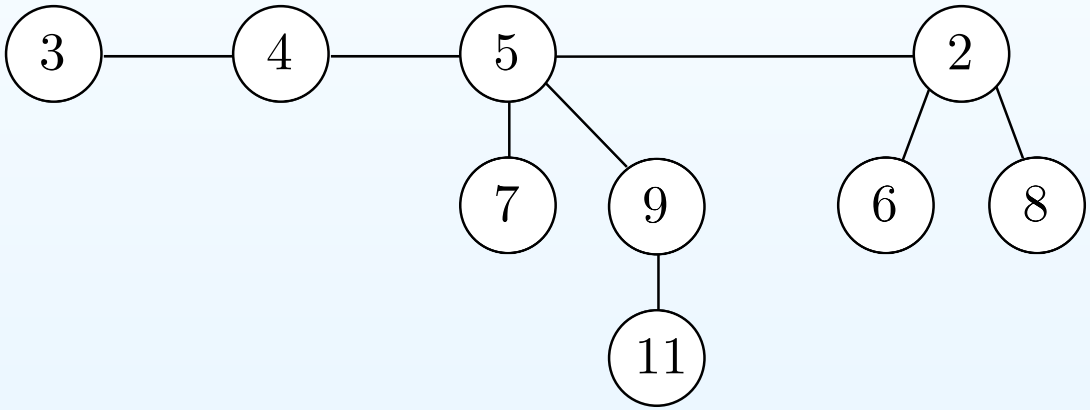


$A[1]$ is already used so pair up roots

$$A[0, 1, 2, 3, \ldots, D(n)]$$

Illustration of `Extract-Min` including `Consolidate`

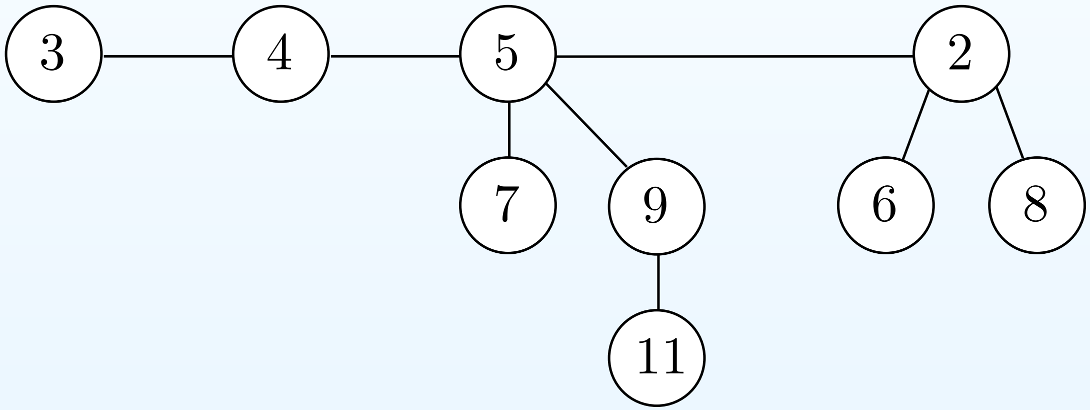
- Then `Consolidate`(H) is applied (the red node is the root currently being processed):



$$A[0, 1, 2, 3, \ldots, D(n)]$$

Illustration of `Extract-Min` including `Consolidate`

- Then `Consolidate`(H) is applied (the red node is the root currently being processed):

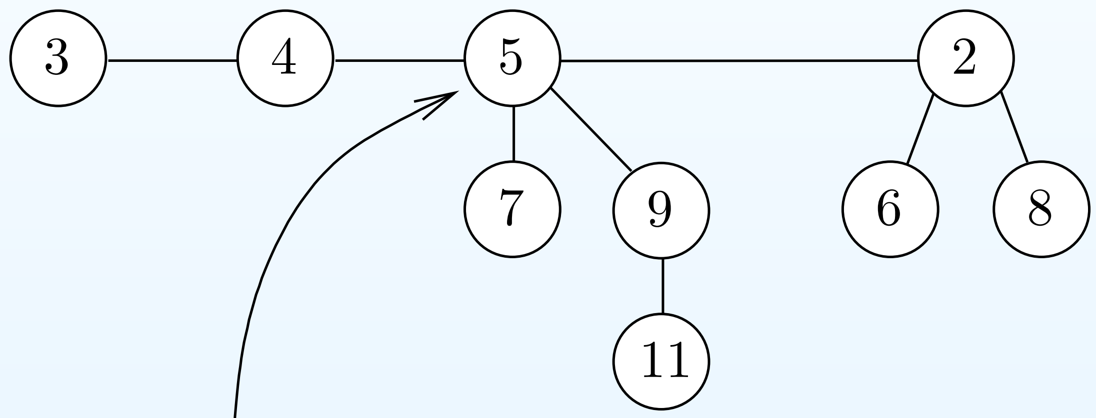


New root has degree 2 so update A

$$A[0, 1, 2, 3, \ldots, D(n)]$$

Illustration of `Extract-Min` including `Consolidate`

- Then `Consolidate`(H) is applied (the red node is the root currently being processed):



$$A[0, 1, 2, 3, \ldots, D(n)]$$

Illustration of `Extract-Min` including `Consolidate`

- Then `Consolidate`(H) is applied (the red node is the root currently being processed):

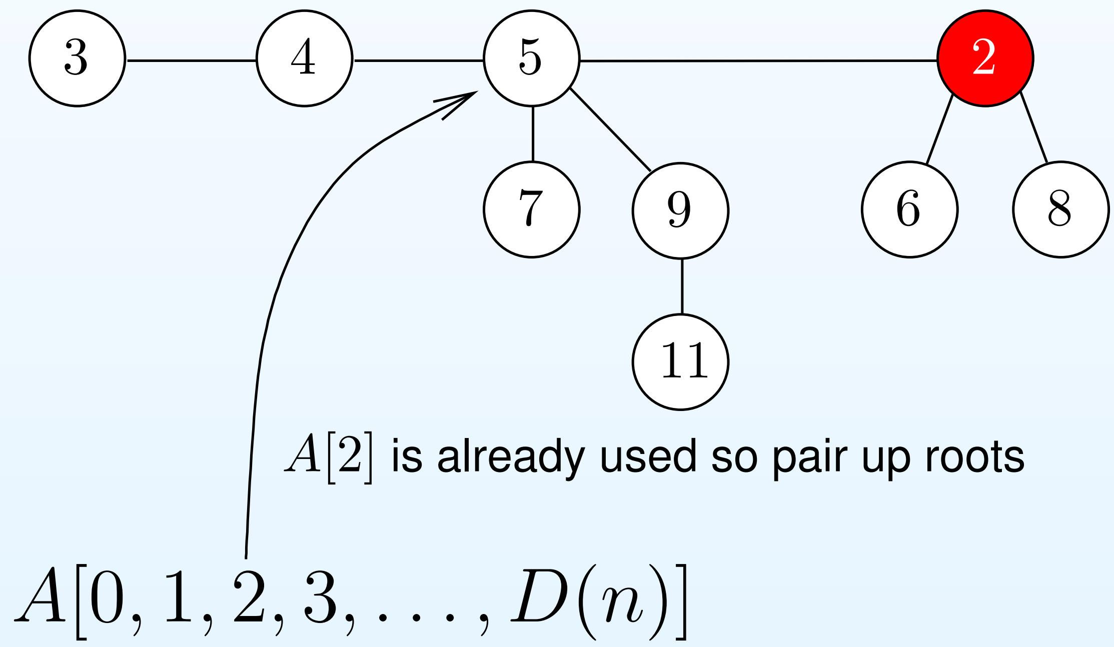
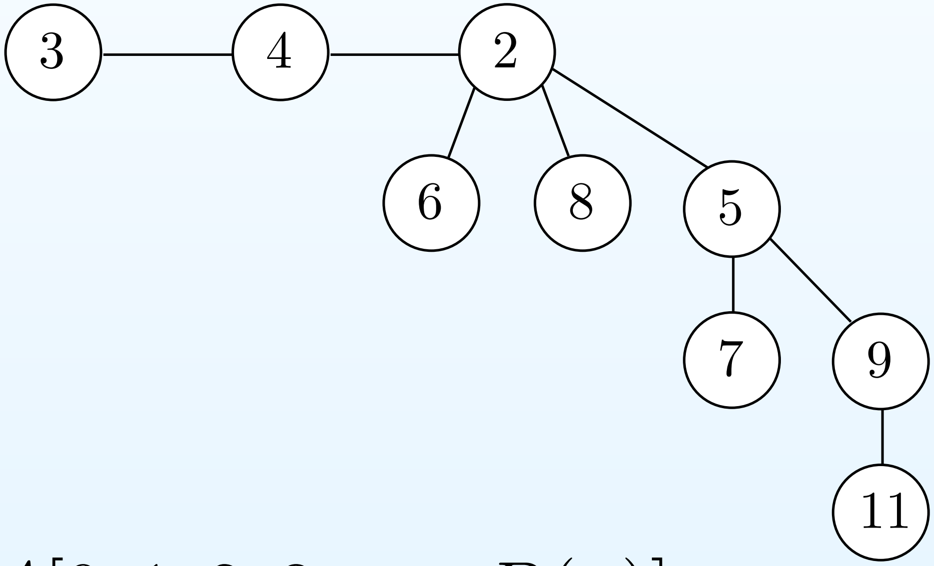


Illustration of `Extract-Min` including `Consolidate`

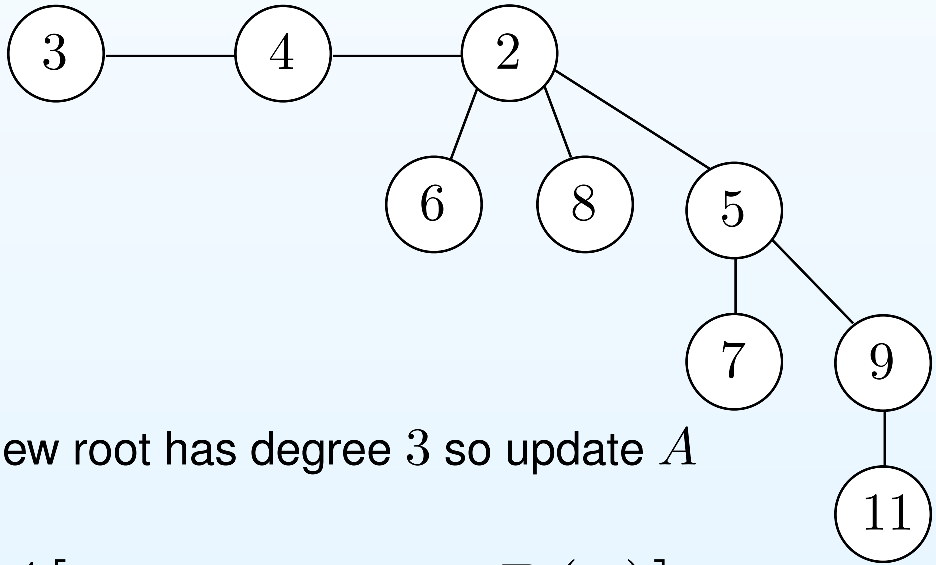
- Then $\texttt{Consolidate}(H)$ is applied (the red node is the root currently being processed):



$$A[0, 1, 2, 3, \ldots, D(n)]$$

Illustration of `Extract-Min` including `Consolidate`

- Then `Consolidate`(H) is applied (the red node is the root currently being processed):

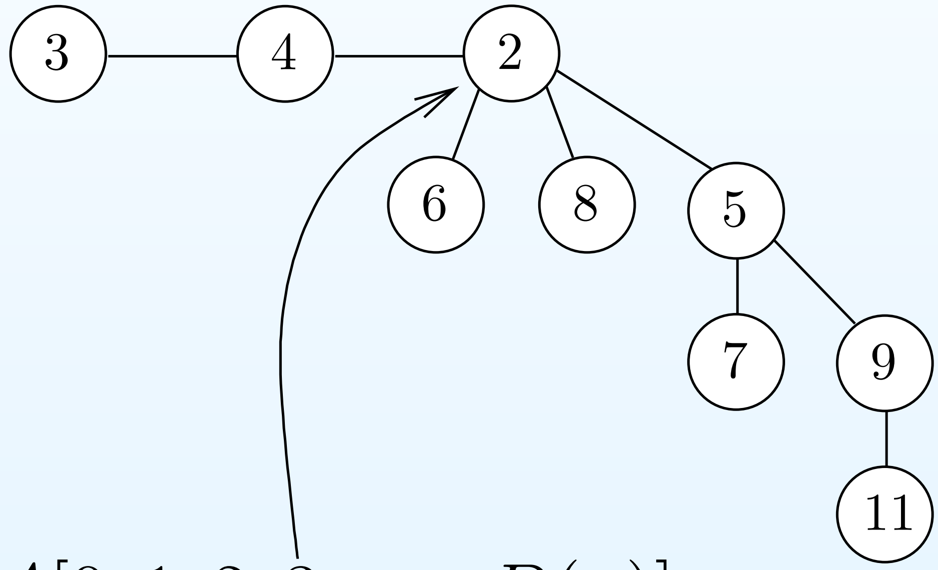


New root has degree 3 so update A

$$A[0, 1, 2, 3, \ldots, D(n)]$$

Illustration of `Extract-Min` including `Consolidate`

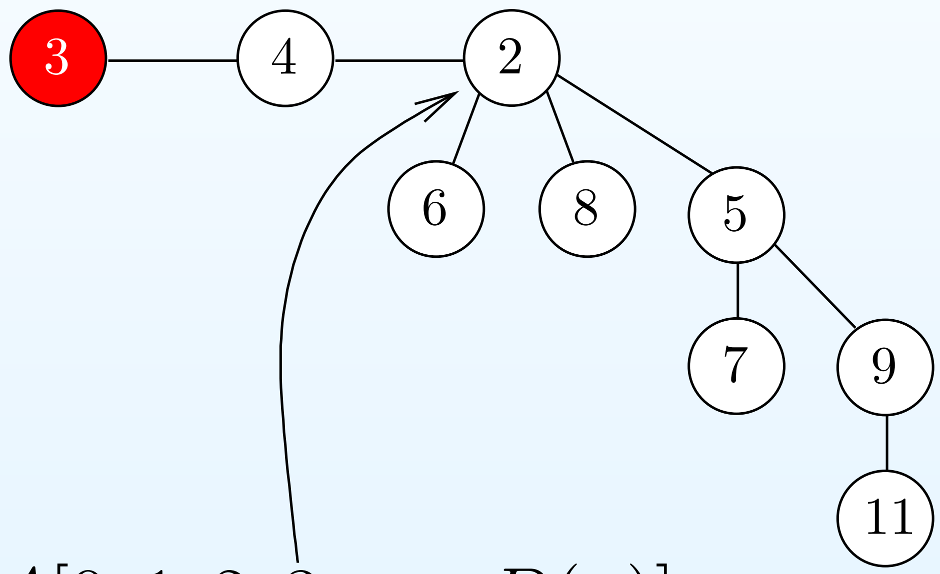
- Then `Consolidate`(H) is applied (the red node is the root currently being processed):



$$A[0, 1, 2, 3, \ldots, D(n)]$$

Illustration of `Extract-Min` including `Consolidate`

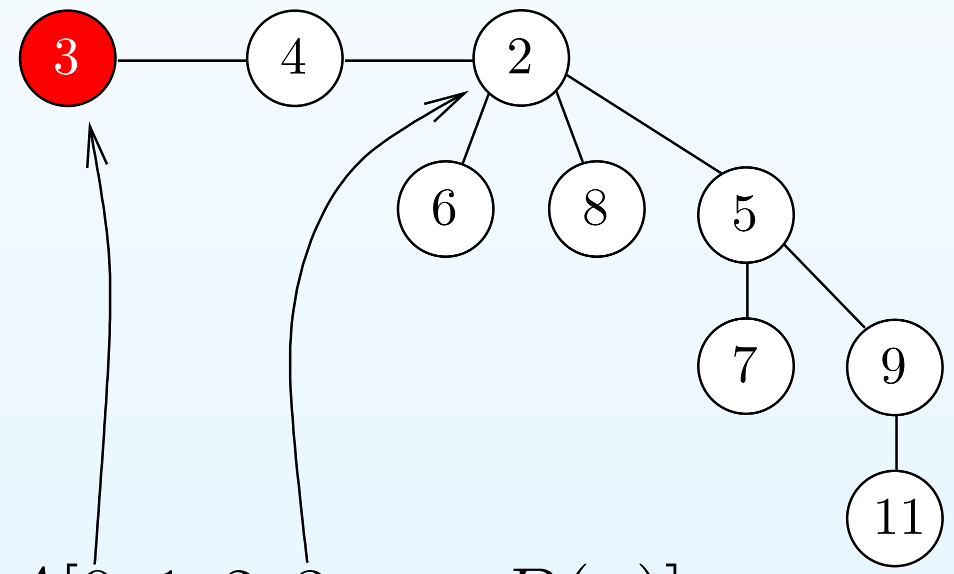
- Then `Consolidate`(H) is applied (the red node is the root currently being processed):



$$A[0, 1, 2, 3, \ldots, D(n)]$$

Illustration of `Extract-Min` including `Consolidate`

- Then `Consolidate`(H) is applied (the red node is the root currently being processed):



$$A[0, 1, 2, 3, \ldots, D(n)]$$

Illustration of `Extract-Min` including `Consolidate`

- Then `Consolidate`(H) is applied (the red node is the root currently being processed):

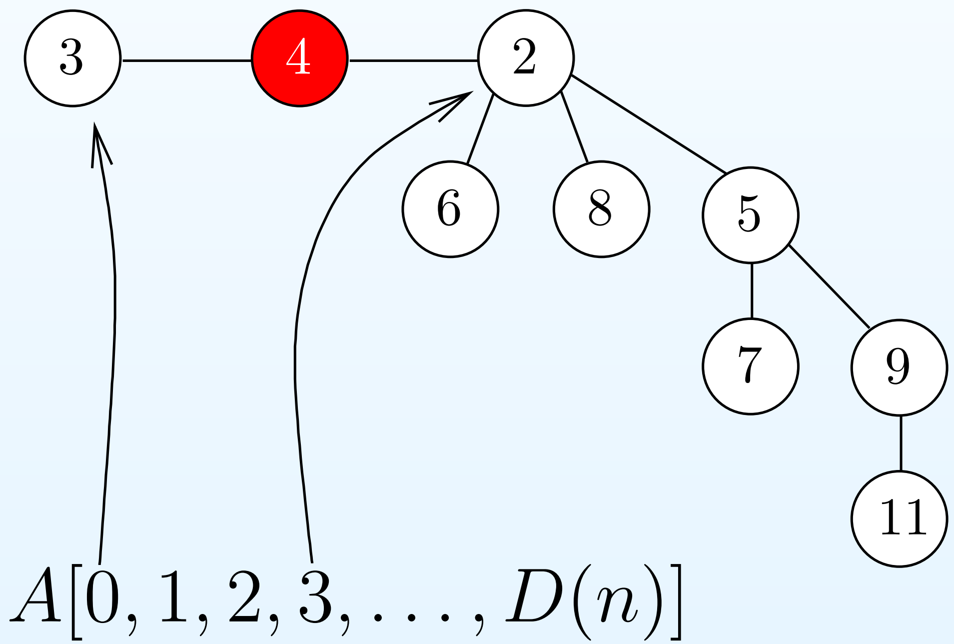
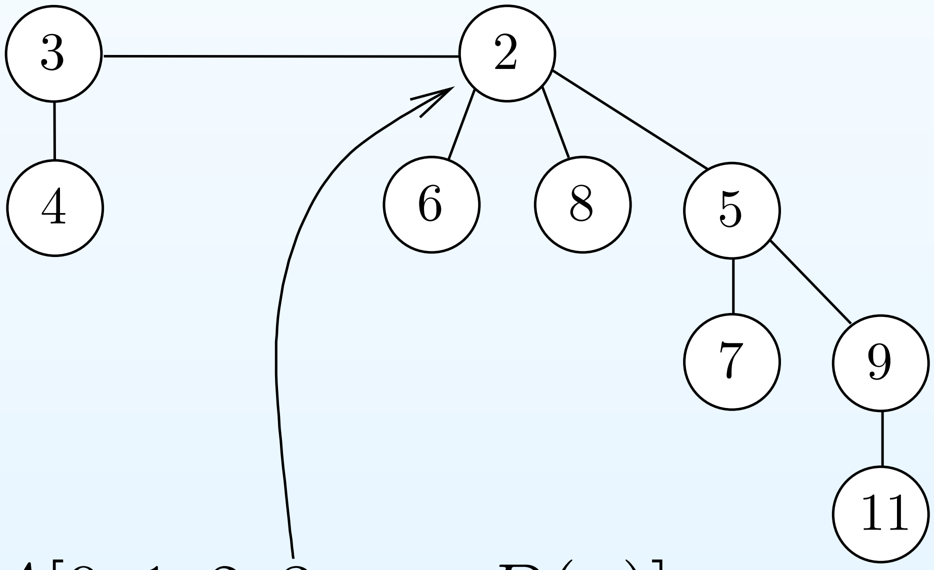


Illustration of `Extract-Min` including `Consolidate`

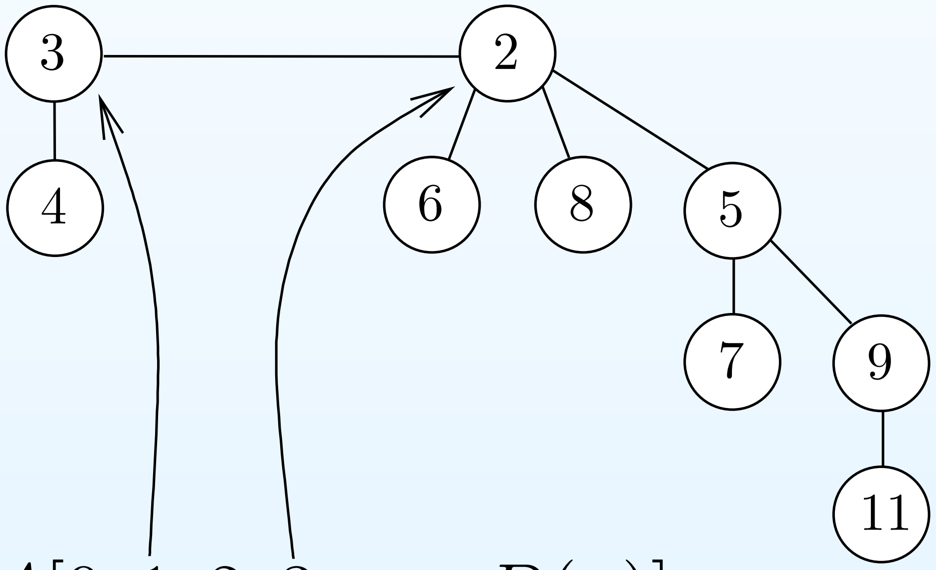
- Then `Consolidate`(H) is applied (the red node is the root currently being processed):



$$A[0, 1, 2, 3, \ldots, D(n)]$$

Illustration of `Extract-Min` including `Consolidate`

- Then `Consolidate`(H) is applied (the red node is the root currently being processed):



$$A[0, 1, 2, 3, \ldots, D(n)]$$

Worst-case running time for `Extract-Min`

- Extracting H.min and moving its children to the root list takes time $O(1 + D(n))$
- Just before `Consolidate`, there are $O(t(H) + D(n))$ roots
- At each step of `Consolidate`, we either traverse forward in the root list or reduce the number of roots by 1
- Hence, `Consolidate` takes worst-case time $O(t(H) + D(n))$
- Finding the new H.min can be done within this time bound as well
- We conclude that the worst-case running time of `Extract-Min` is $c_i = O(t(H) + D(n) + 1) = O(t(H) + D(n))$
- This may be as large as $\Theta(n)$
- We now show that the amortized time is only $O(\lg n)$
- Intuition:
  - If the worst-case running time is long then the number of trees is reduced by a lot in `Consolidate`
  - This gives a large reduction in the potential which pays for the long worst-case running time

Amortized running time for `Extract-Min`

- We have shown that $c_i = O(t(H) + D(n))$
- We may choose to measure time in any unit we like (microseconds, seconds, etc.)
- Thus, we may assume that $c_i \le t(H) + D(n)$
- The potential before the operation is $\Phi(H) = t(H) + 2m(H)$
- The potential afterwards is $\Phi(H') = t(H') + 2m(H')$
- We have $m(H') \le m(H)$
- Also, $t(H') \le D(n) + 1$ since all roots have distinct degrees in $\{0, \ldots, D(n)\}$ after `Consolidate`
- From the above, the amortized cost of `Extract-Min` is

$$\hat{c}_i = c_i + \Phi(H') - \Phi(H)$$

$$\le \overbrace{t(H) + D(n)}^{\ge c_i} + \overbrace{\underbrace{D(n) + 1}_{\ge t(H')} + \underbrace{2m(H)}_{\ge 2m(H')}}^{\ge \Phi(H')} - \overbrace{(t(H) + 2m(H))}^{=\Phi(H)}$$

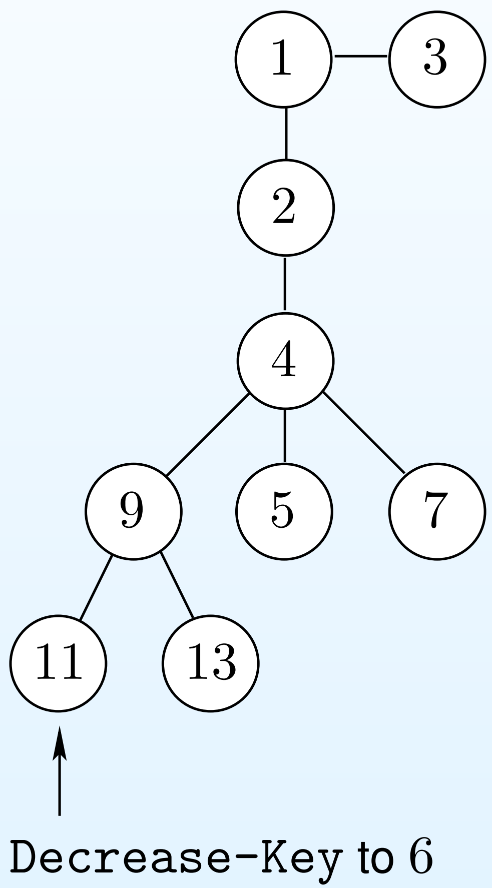
$$= 2D(n) + 1 = \Theta(\lg n) \quad O(\lg n)$$

The `Decrease-Key` operation

- Suppose the ith operation is $\texttt{Decrease-Key}(H, x, k)$
- $\mathrm{key}(x)$ is reduced to k (we assume that k is no greater than the old key of x)
- This change may violate the min-heap property, i.e., it may happen that $\mathrm{key}(x) < \mathrm{key}(p)$ after the update where p is the parent of x
- If this is the case, `Decrease-Key` needs to do additional work (described later)

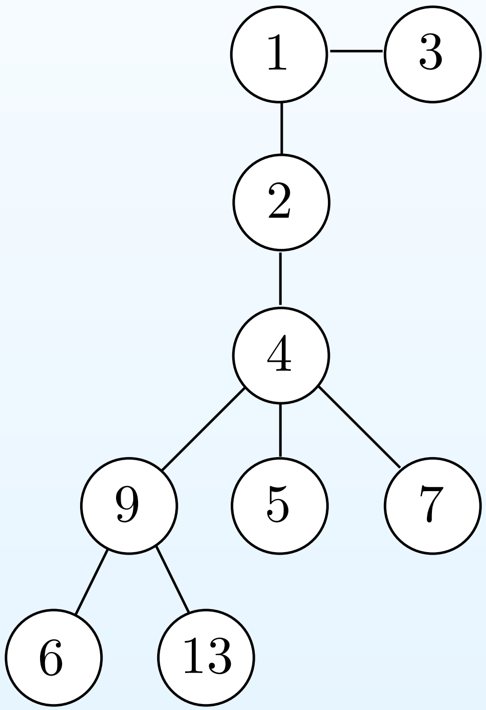
Decreasing a key may violate the min-heap property

- Example:



Decreasing a key may violate the min-heap property

- Example:



Rules for marking/unmarking nodes

- Recall that each node u of a Fibonacci heap has a field u.mark
- When u has just been added to the heap, u.mark is set to false
- If u gets a parent (in `Consolidate`), u.mark is set to false
- When u loses its first child, u.mark is set to true
- When u loses its second child, u.mark is set to false and u becomes a new root of the Fibonacci heap
- Silly, dark, but fairly useful mnemonic:
  - When a node loses a child, it becomes sad and is thus marked by the situation
  - When it loses its second child, it can't take it any more and starts a new life as a happy (unmarked) root
- A root node is not marked by losing a child

Cascading cut

- Let us return to `Decrease-Key` just after $\mathrm{key}(x)$ is reduced to a value less than $\mathrm{key}(p)$ (if $\mathrm{key}(x) \geq \mathrm{key}(p)$, no further updates are done)
- Then x is cut from p and added as a new root and x.mark $=$ false
- If p.mark was false, it is now updated to true and the process stops
- If p.mark was already true prior to cutting x, p now loses its second child so it too becomes a new root and p.mark is set to false
- These updates continue recursively with the parent of p
- The recursion stops if reaching a root
- We call this process a *cascading cut* since multiple nodes may be cut and made into new roots

Illustration of `Decrease-Key` with cascading cuts

- Example with two `Decrease-Key` operations (red nodes are marked):

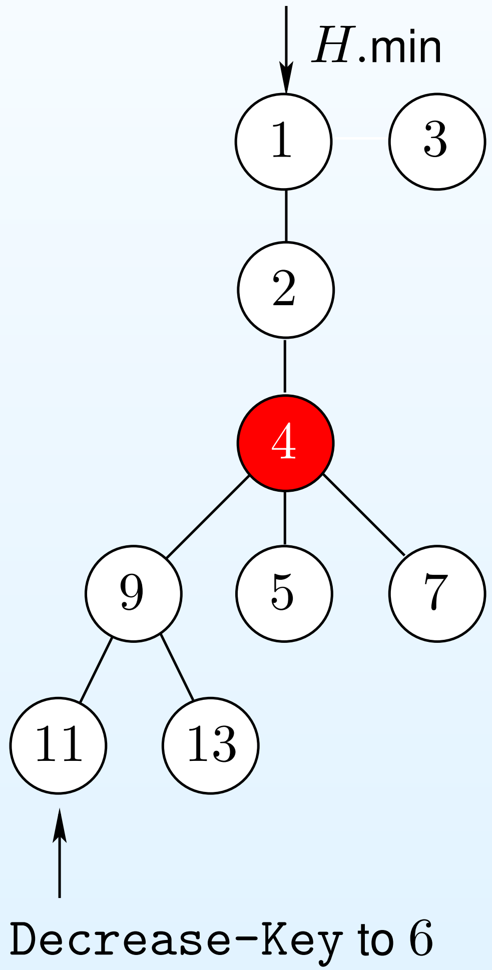


Illustration of `Decrease-Key` with cascading cuts

- Example with two `Decrease-Key` operations (red nodes are marked):

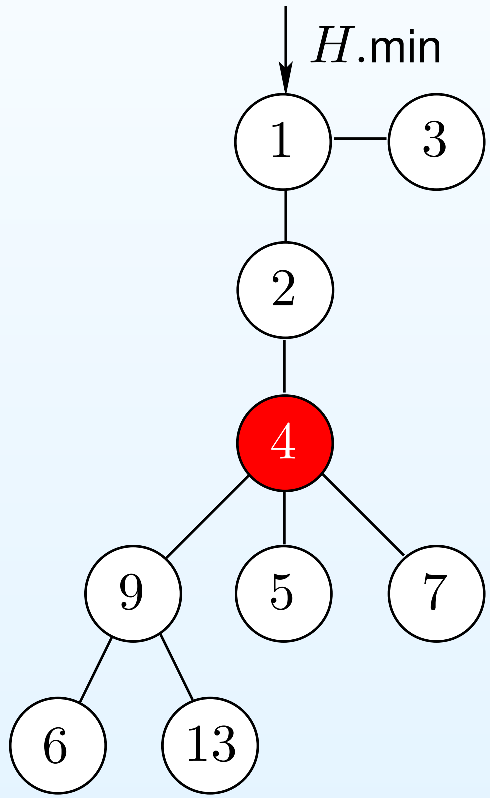


Illustration of `Decrease-Key` with cascading cuts

- Example with two `Decrease-Key` operations (red nodes are marked):

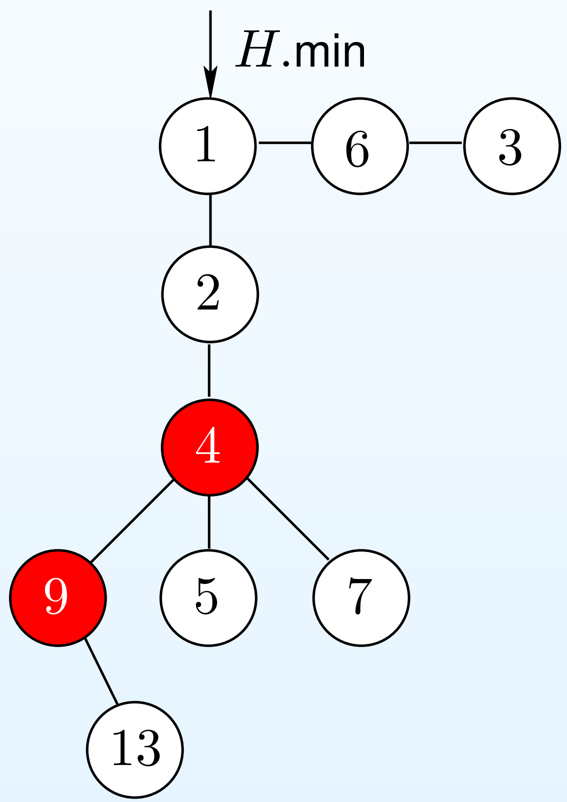


Illustration of `Decrease-Key` with cascading cuts

- Example with two `Decrease-Key` operations (red nodes are marked):

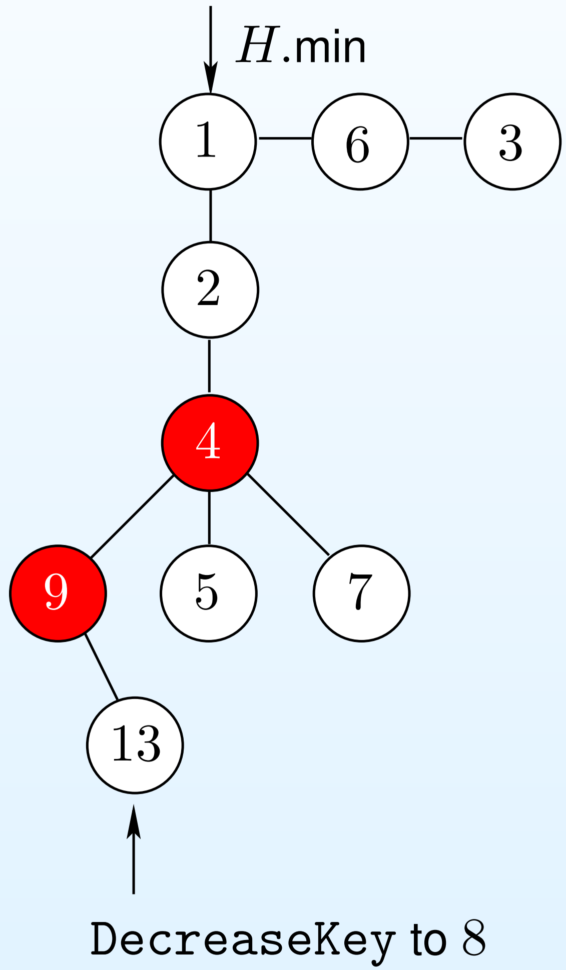


Illustration of `Decrease-Key` with cascading cuts

- Example with two `Decrease-Key` operations (red nodes are marked):

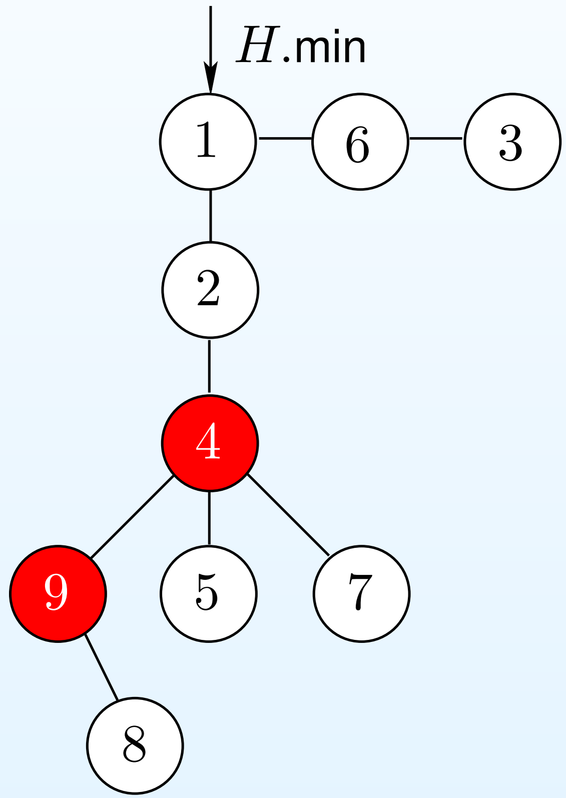


Illustration of `Decrease-Key` with cascading cuts

- Example with two `Decrease-Key` operations (red nodes are marked):

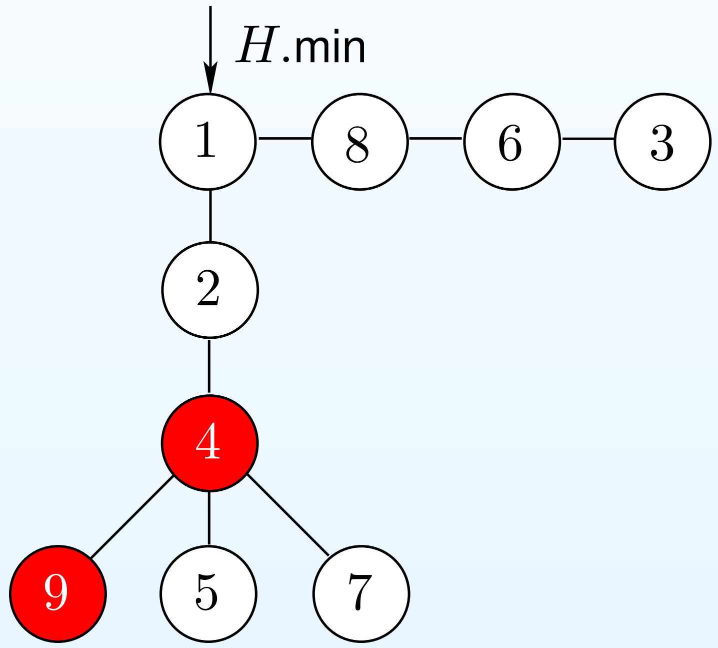


Illustration of `Decrease-Key` with cascading cuts

- Example with two `Decrease-Key` operations (red nodes are marked):

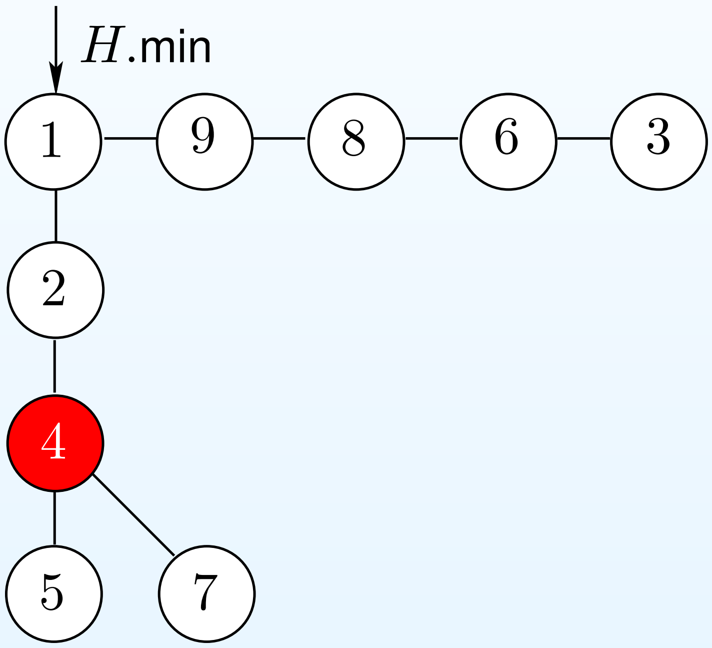


Illustration of `Decrease-Key` with cascading cuts

- Example with two `Decrease-Key` operations (red nodes are marked):

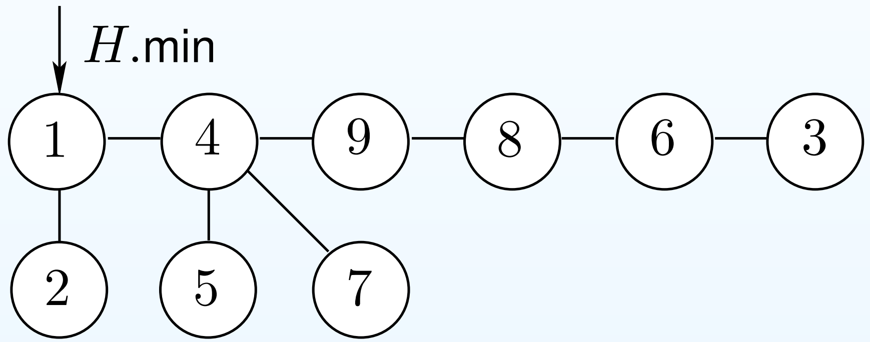
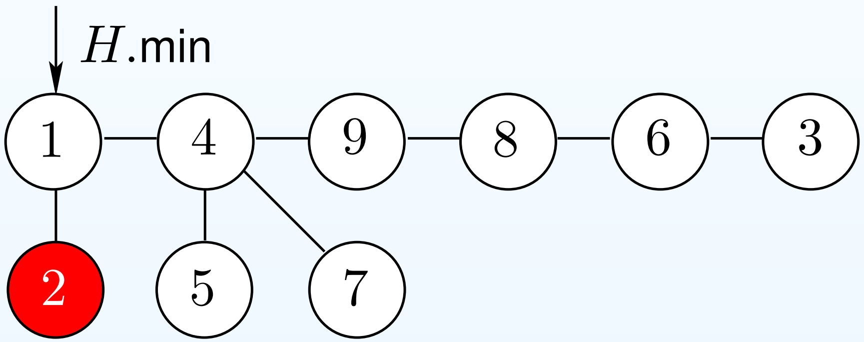


Illustration of `Decrease-Key` with cascading cuts

- Example with two `Decrease-Key` operations (red nodes are marked):



Amortized running time for `Decrease-Key`

- Let c be the number of new roots created in `Decrease-Key`(H, x, k)
- We consider the interesting case $c \geq 1$
- Then the worst-case cost of the operation is $c_i = O(c)$
- Choosing a suitable unit of time (like we did for `Extract-Min`), gives $c_i \leq c$
- Potential before the update: $\Phi(H) = t(H) + 2m(H)$
- Potential after the update: $\Phi(H') = t(H') + 2m(H')$
- Since c new roots are created, $t(H') = t(H) + c$
- At least $c - 1$ nodes change from marked to unmarked
- At most one node changes from unmarked to marked
- Thus, $m(H') \leq m(H) + 1 - (c - 1) = m(H) + 2 - c$
- Then

$$\begin{aligned}\Phi(H') = t(H') + 2m(H') &\leq \overbrace{t(H) + c}^{=t(H')} + \overbrace{2(m(H) + 2 - c)}^{\geq 2m(H')} \\ &= t(H) + 2m(H) + 4 - c\end{aligned}$$

Amortized running time for `Decrease-Key`

- We have shown:
  - $c_i \leq c$
  - $\Phi(H) = t(H) + 2m(H)$
  - $\Phi(H') \leq t(H) + 2m(H) + 4 - c$
- The amortized cost of `Decrease-Key` is therefore

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(H') - \Phi(H) \\ &\leq \overbrace{c}^{\geq c_i} + \overbrace{t(H) + 2m(H) + 4 - c}^{\geq \Phi(H')} - \overbrace{(t(H) + 2m(H))}^{=\Phi(H)} \\ &= 4 = O(1)\end{aligned}$$

The `Union` operation

- Suppose the ith operation is $\texttt{Union}(H_1, H_2)$
- The union H of the two is obtained by cutting open the two circular, doubly linked lists for H_1 and H_2 into one
- Then H.min is set to the node with minimum key among H_1.min and H_2.min
- This can be done in worst-case time $c_i = O(1)$
- Potential before update: $\Phi(H_1) + \Phi(H_2)$
- Potential after update: $\Phi(H)$
- We have $t(H) = t(H_1) + t(H_2)$ and $m(H) = m(H_1) + m(H_2)$
- Thus, $\Phi(H) = \Phi(H_1) + \Phi(H_2)$
- It follows that the potential difference is 0 so $\hat{c}_i = c_i = O(1)$

The `Delete` operation

- Suppose the ith operation is $\texttt{Delete}(H, x)$
- This operation can be implemented by first decreasing the key of x to $-\infty$ and then extracting the minimum element from H
- This takes amortized time $O(1) + O(\lg n) = O(\lg n)$

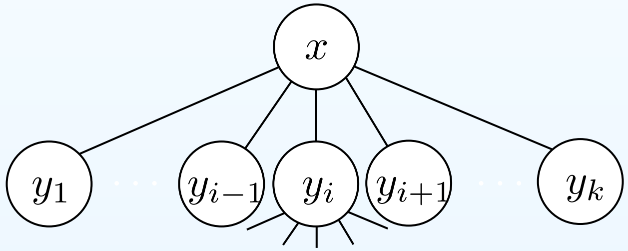
STOP

Bounding $D(n)$

- We have obtained the desired amortized time bounds for all types of operations
- We made the claim that we have an upper bound $D(n) = \Theta(\lg n)$ on the maximum degree of a node in any n-node Fibonacci heap
- We now prove this claim (the following slides are only cursory)

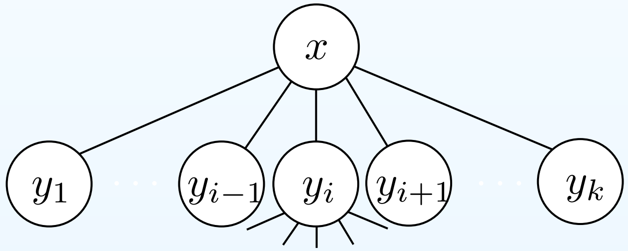
Lemma 1

- Let x be a node of a Fibonacci heap H and let $k = x$.deg
- Let $y_1, \ldots, y_k$ be the children of x ordered by birth (the time they were added as a child to x)
- Then y_1.deg ≥ 0 and y_i.deg $\geq i - 2$ for $i = 2, \ldots, k$



Lemma 1

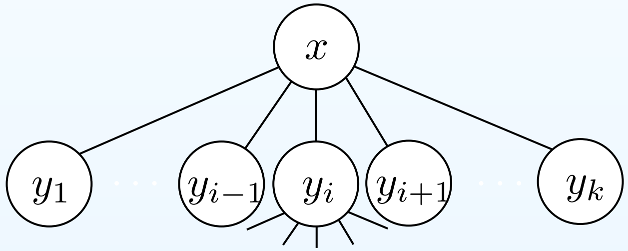
- Let x be a node of a Fibonacci heap H and let $k = x$.deg
- Let $y_1, \ldots, y_k$ be the children of x ordered by birth (the time they were added as a child to x)
- Then y_1.deg ≥ 0 and y_i.deg $\geq i - 2$ for $i = 2, \ldots, k$



- Proof (for $i \in \{2, \ldots, k\}$):

Lemma 1

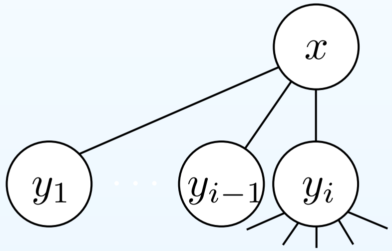
- Let x be a node of a Fibonacci heap H and let $k = x.\mathrm{deg}$
- Let $y_1, \ldots, y_k$ be the children of x ordered by birth (the time they were added as a child to x)
- Then $y_1.\mathrm{deg} \geq 0$ and $y_i.\mathrm{deg} \geq i - 2$ for $i = 2, \ldots, k$



- Proof (for $i \in \{2, \ldots, k\}$):
  - When y_i was added as a child to x, the two nodes had the same degree (`Consolidate`)

Lemma 1

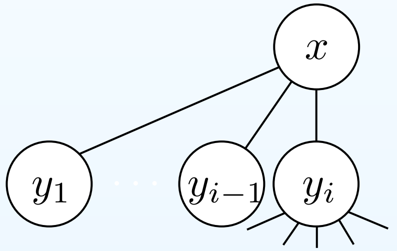
- Let x be a node of a Fibonacci heap H and let $k = x$.deg
- Let $y_1, \ldots, y_k$ be the children of x ordered by birth (the time they were added as a child to x)
- Then y_1.deg ≥ 0 and y_i.deg $\geq i-2$ for $i = 2, \ldots, k$



- Proof (for $i \in \{2, \ldots, k\}$):
  - When y_i was added as a child to x, the two nodes had the same degree (`Consolidate`)

Lemma 1

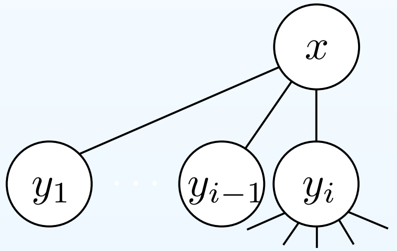
- Let x be a node of a Fibonacci heap H and let $k = x$.deg
- Let $y_1, \ldots, y_k$ be the children of x ordered by birth (the time they were added as a child to x)
- Then y_1.deg ≥ 0 and y_i.deg $\geq i - 2$ for $i = 2, \ldots, k$



- Proof (for $i \in \{2, \ldots, k\}$):
    - When y_i was added as a child to x, the two nodes had the same degree (`Consolidate`)
    - Since x had at least $i - 1$ children at that point, namely $y_1, \ldots, y_{i-1}$, node y_i had degree at least $i - 1$ at that point too

Lemma 1

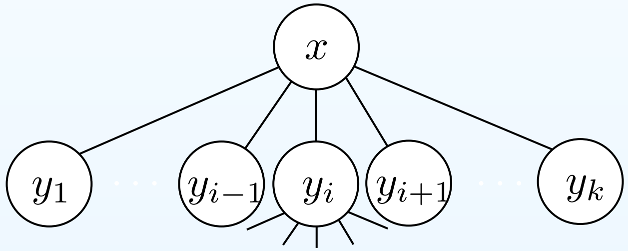
- Let x be a node of a Fibonacci heap H and let $k = x$.deg
- Let $y_1, \ldots, y_k$ be the children of x ordered by birth (the time they were added as a child to x)
- Then y_1.deg ≥ 0 and y_i.deg $\geq i - 2$ for $i = 2, \ldots, k$



- Proof (for $i \in \{2, \ldots, k\}$):
  - When y_i was added as a child to x, the two nodes had the same degree (`Consolidate`)
  - Since x had at least $i - 1$ children at that point, namely $y_1, \ldots, y_{i-1}$, node y_i had degree at least $i - 1$ at that point too
  - Since then, y_i could not have lost more than one child, as otherwise, it would have become a root (`Decrease-Key`)

Lemma 1

- Let x be a node of a Fibonacci heap H and let $k = x$.deg
- Let $y_1, \ldots, y_k$ be the children of x ordered by birth (the time they were added as a child to x)
- Then y_1.deg ≥ 0 and y_i.deg $\geq i - 2$ for $i = 2, \ldots, k$



- Proof (for $i \in \{2, \ldots, k\}$):
  - When y_i was added as a child to x, the two nodes had the same degree (`Consolidate`)
  - Since x had at least $i - 1$ children at that point, namely $y_1, \ldots, y_{i-1}$, node y_i had degree at least $i - 1$ at that point too
  - Since then, y_i could not have lost more than one child, as otherwise, it would have become a root (`Decrease-Key`)

Two additional lemmas (proofs omitted)

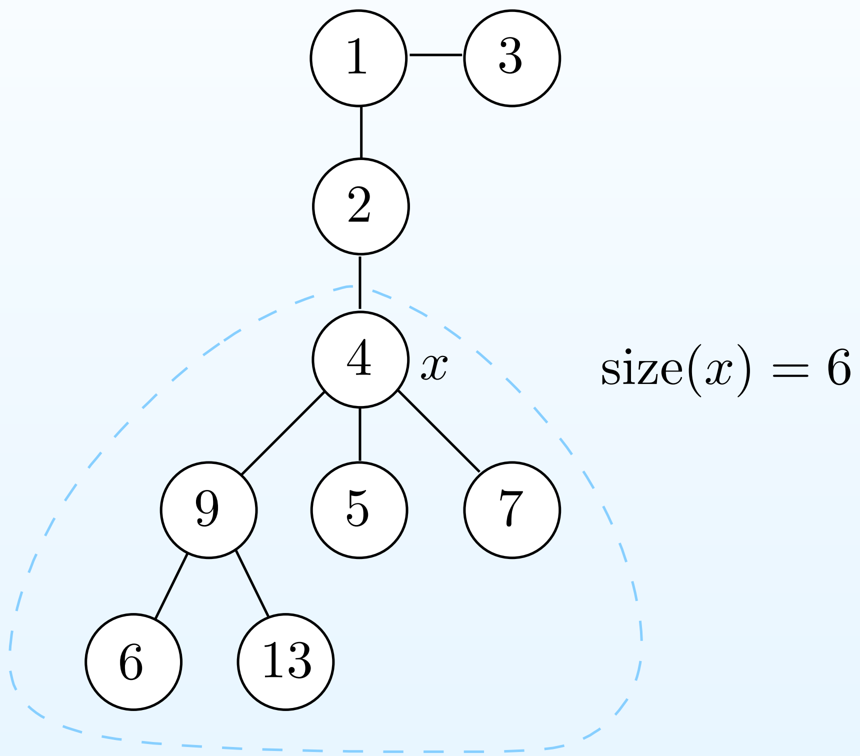
- Lemma 2: $\forall k \in \mathbb{N}_0$, $F_{k+2} = 1 + \sum_{i=0}^{k} F_i$
- Lemma 3: $\forall k \in \mathbb{N}_0$, $F_{k+2} \geq \phi^k$ where

$$\phi = (1 + \sqrt{5})/2 = 1,61803\ldots$$

is the golden ratio

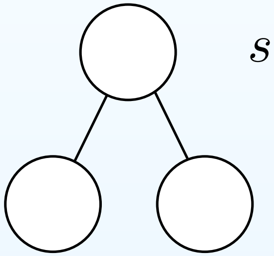
Relating tree size to node degree

- Define the *size* $\mathrm{size}(x)$ of a node x to be the number of nodes in the tree rooted at x



Relating tree size to node degree

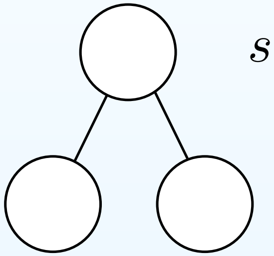
- Define the *size* $\mathrm{size}(x)$ of a node x to be the number of nodes in the tree rooted at x
- Let s_k be the minimum size of a node of degree k in any Fibonacci heap



$s_2 = 3$

Relating tree size to node degree

- Define the *size* $\mathrm{size}(x)$ of a node x to be the number of nodes in the tree rooted at x
- Let s_k be the minimum size of a node of degree k in any Fibonacci heap

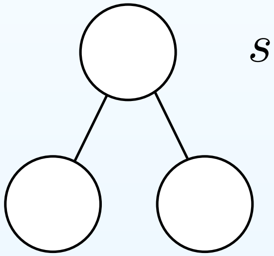


$$s_2 = 3 = F_4$$

- We claim that $s_k \geq F_{k+2}$

Relating tree size to node degree

- Define the *size* $\mathrm{size}(x)$ of a node x to be the number of nodes in the tree rooted at x
- Let s_k be the minimum size of a node of degree k in any Fibonacci heap



$s_2 = 3 = F_4$

- We claim that $s_k \geq F_{k+2}$
- If we can show this, it follows from Lemma 3 that for any degree k-node x of an n-node Fibonacci heap,

$$n \geq \mathrm{size}(x) \geq s_k \geq F_{k+2} \geq \phi^k$$

- Take logs on both sides:
  $\lg n \geq k \lg(\phi) \Rightarrow k \leq \frac{\lg n}{\lg(\phi)} = O(\lg n)$
- Hence, the maximum degree is $O(\lg n)$, as desired

Showing $s_k \geq F_{k+2}$

- It remains to show $s_k \geq F_{k+2}$
- We prove this by induction on $k \geq 0$
- Since $s_0 = 1 = F_2$ and $s_1 = 2 = F_3$, the claim holds for $k = 0, 1$
- Now, assume $k > 1$ and that the claim holds for smaller values
- Let x be a degree k-node with $\mathrm{size}(x) = s_k$
- Order the children of x by birth: $y_1, \ldots, y_k$
- By Lemma 1 and the observation that $s_{k_1} \geq s_{k_2}$ for all $k_1 \geq k_2$,

$$s_k = \mathrm{size}(x) = 1 + \sum_{i=1}^{k} \mathrm{size}(y_i) \geq 2 + \sum_{i=2}^{k} s_{y_i.\text{deg}} \geq 2 + \sum_{i=2}^{k} s_{i-2}$$

- The induction hypothesis and Lemma 2 then show the induction step:

$$s_k \geq 2 + \sum_{i=2}^{k} s_{i-2} \geq 2 + \sum_{i=2}^{k} F_i = 1 + \sum_{i=0}^{k} F_i = F_{k+2}$$

Plan for the lecture on March 6 (Pawel's lecture)

- Binary Search Trees
- Balanced Binary Search Trees: red-black trees
- Note: there is no lecture on Wednesday, March 1, due to Åbent Hus at KU
- Good luck with the rest of the course and see you for the question hour!