Fibonacci Heaps

Porter Shawver

Outline

Section 1

Priority Queue Background

Priority Queues

For the entirety of this presentation, I will be using minimum priority queues, but maximum priority queues are implemented the same way.

Operations:

- Insert - add an element to the queue.
- FindMin - return the smallest element.
- DeleteMin - remove the smallest element.
- DecreaseKey - decrease an element.

Binary Heap

A binary heap is a binary tree that maintains two properties:

Binary Heap

A binary heap is a binary tree that maintains two properties:

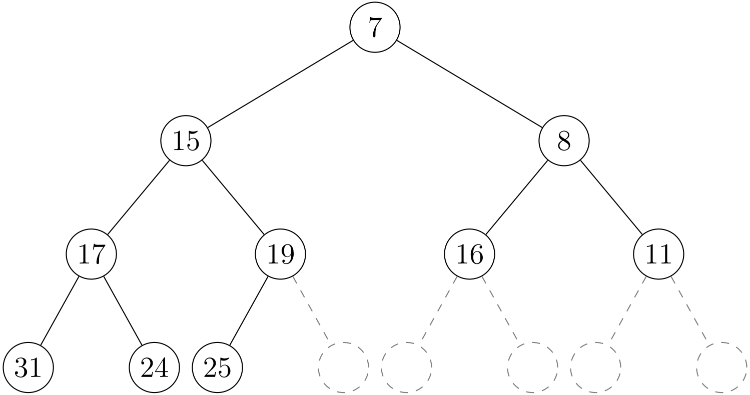
1. All children must have a value greater than their parent (“heap property”). Thus, the root should be the next value out of the queue.

Binary Heap

A binary heap is a binary tree that maintains two properties:

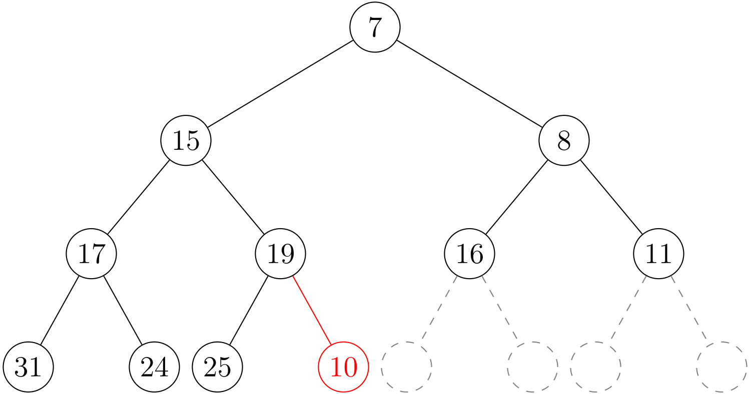
1. All children must have a value greater than their parent ("heap property"). Thus, the root should be the next value out of the queue.
2. The tree is complete (every level must be completely full, except the last). This allows us to use an array, and helps with worst case runtime.

Binary Heap Example



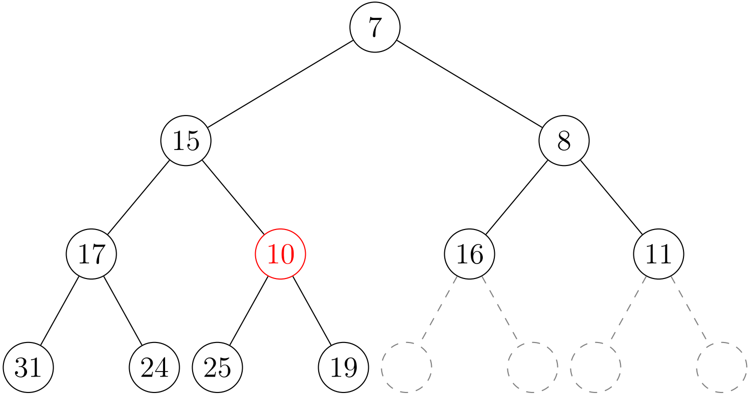
Binary Heap INSERT

1. Add value to first open space.



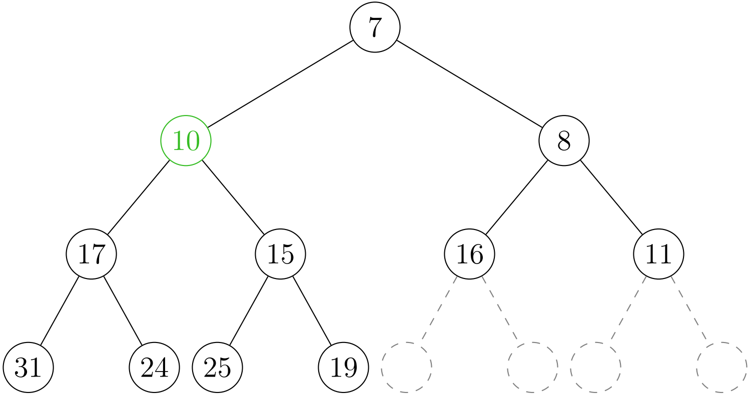
Binary Heap INSERT

2. Swap with parent until the parent is smaller to maintain heap property.



Binary Heap INSERT

2. Swap with parent until the parent is smaller to maintain heap property.



Binary Heap INSERT

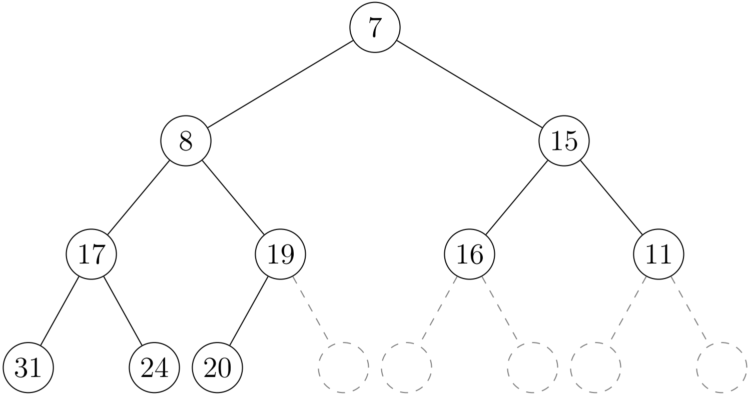
Worst case, the inserted value is the new smallest value, and must be swapped all the way to the top: $O(\log n)$.

Binary Heap FindMin

The root is always the min, so finding it is $O(1)$.

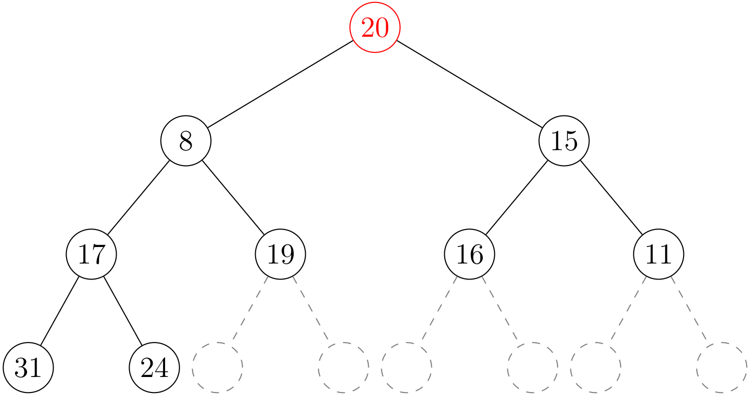
Binary Heap DeleteMin

1. Return the root value, and replace the root node with the last element in the array.



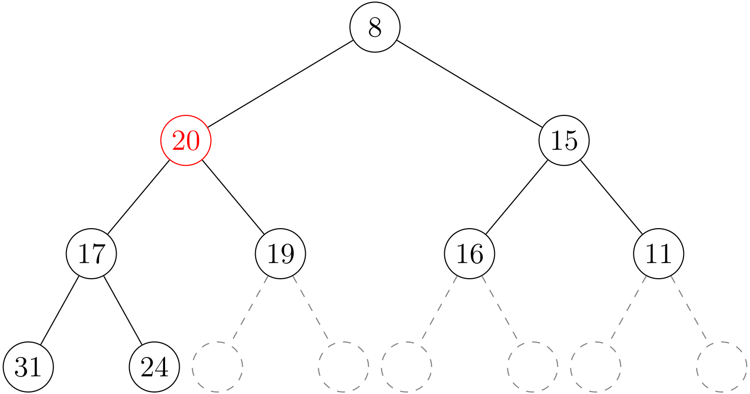
Binary Heap DeleteMin

1. Return the root value, and replace the root node with the last element in the array.



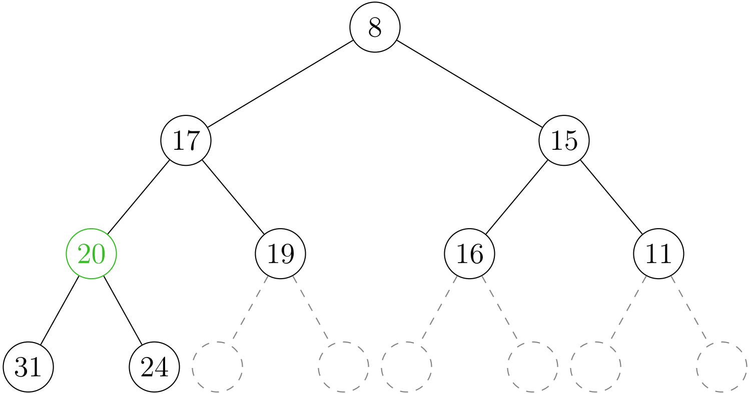
Binary Heap DeleteMin

2. Swap with smallest child until both children are larger.



Binary Heap DeleteMin

2. Swap with smallest child until both children are larger.



Binary Heap DeleteMin

Worst case, the replacement value will be swapped all the way to the bottom. Because this is a binary tree, we know this is $O(\log n)$.

Binary Heap DecreaseKey

This is the same as insert: decrease the priority of the node, then swap it with its parent until the parent is larger: $O(\log n)$.

Binary vs. Fibonacci Heap Runtimes

	Binary Heap	Fibonacci Heap
Insert	$O(\log n)$	$O(1)$
FindMin	$O(1)$	$O(1)$
DeleteMin	$O(\log n)$	$O(\log n)$
DecreaseKey	$O(\log n)$	$O(1)$

Questions?

Section 2

Binomial Heaps

Binomial Heaps

First step on the way to Fibonacci heaps: what if we use more than one tree?

Binomial Heaps

First step on the way to Fibonacci heaps: what if we use more than one tree?

Properties:

- Composed of a forest of trees.

Binomial Heaps

First step on the way to Fibonacci heaps: what if we use more than one tree?

Properties:

- Composed of a forest of trees.
- Binomial tree of order k is a node with k children **that are themselves binomial trees** of orders $k-1, k-2, \ldots, 2, 1, 0$.

Binomial Heaps

First step on the way to Fibonacci heaps: what if we use more than one tree?

Properties:

- Composed of a forest of trees.
- Binomial tree of order k is a node with k children **that are themselves binomial trees** of orders $k-1, k-2, \ldots, 2, 1, 0$.
- There can be at most one binomial tree for each order.

Binomial Heaps

First step on the way to Fibonacci heaps: what if we use more than one tree?

Properties:

- Composed of a forest of trees.
- Binomial tree of order k is a node with k children **that are themselves binomial trees** of orders $k-1, k-2, \ldots, 2, 1, 0$.
- There can be at most one binomial tree for each order.
- Level ℓ has $\binom{k}{\ell}$ nodes in a binomial tree of order k.

Binomial Heaps

First step on the way to Fibonacci heaps: what if we use more than one tree?

Properties:

- Composed of a forest of trees.
- Binomial tree of order k is a node with k children **that are themselves binomial trees** of orders $k-1, k-2, \ldots, 2, 1, 0$.
- There can be at most one binomial tree for each order.
- Level ℓ has $\binom{k}{\ell}$ nodes in a binomial tree of order k.
- Thus, a tree of order k has 2^k nodes.

Binomial Heaps

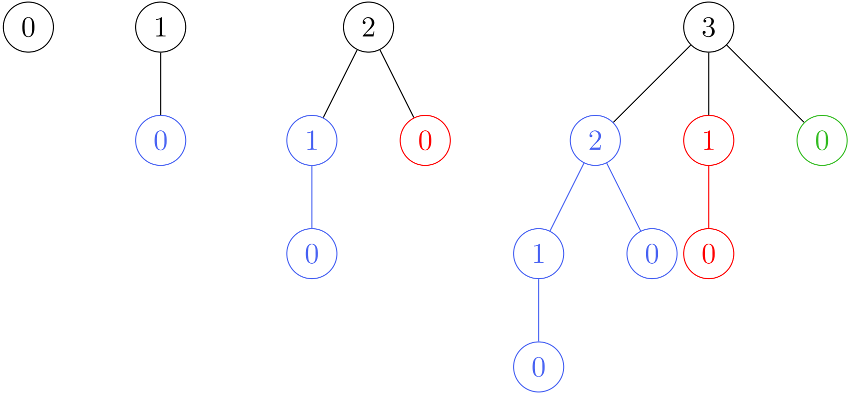
First step on the way to Fibonacci heaps: what if we use more than one tree?

Properties:

- Composed of a forest of trees.
- Binomial tree of order k is a node with k children **that are themselves binomial trees** of orders $k-1, k-2, \ldots, 2, 1, 0$.
- There can be at most one binomial tree for each order.
- Level ℓ has $\binom{k}{\ell}$ nodes in a binomial tree of order k.
- Thus, a tree of order k has 2^k nodes.
- Thus, the largest degree is bounded by $O(\log n)$.

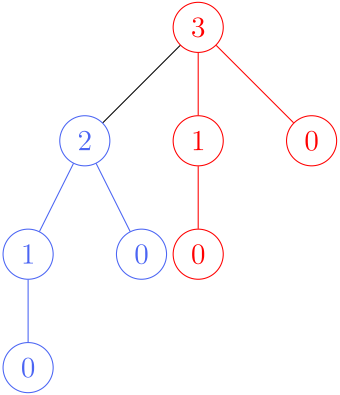
Binomial Heap Example

(All one heap, nodes labeled with their degree.)



Binomial Heap Notes

- Combining two binomial heaps of degree d gives a binomial heap of degree $d + 1$:



Binomial Heap Insert

Using amortized analysis, a binomial heap can insert in $O(1)$ by adding a single node as a new tree.

(More on amortized analysis in a moment.)

Section 3

Fibonacci Heaps

Fibonacci Heaps

Properties:

- Binomial heap that can have more than one tree of any degree.

Fibonacci Heaps

Properties:

- Binomial heap that can have more than one tree of any degree.
- We can remove at most one child from each node. (We mark nodes that have had a child removed)

Fibonacci Heaps

Properties:

- Binomial heap that can have more than one tree of any degree.
- We can remove at most one child from each node. (We mark nodes that have had a child removed)

Very loose structure; could just be a list of all root nodes. We use amortized analysis to get the runtimes we want.

Formalizing Amortized Analysis

- We use amortized analysis when the work done by slow operations can be attributed evenly to a predictable number of fast operations.

Formalizing Amortized Analysis

- We use amortized analysis when the work done by slow operations can be attributed evenly to a predictable number of fast operations.
- With the *Potential Method*, we “bank” time in a potential function, ϕ, on the fast operations, then “cash out” on the slow operations.

Formalizing Amortized Analysis

- We use amortized analysis when the work done by slow operations can be attributed evenly to a predictable number of fast operations.
- With the *Potential Method*, we “bank” time in a potential function, ϕ, on the fast operations, then “cash out” on the slow operations.
- Formally, we define $\Delta\phi_o$ as the change in potential after operation o. Then the amortized time is calculated as

  $$T_{amortized}(o) = T_{actual}(o) + C \cdot \Delta\phi_o$$

  for a constant C that disappears in big O.

Formalizing Amortized Analysis

- We use amortized analysis when the work done by slow operations can be attributed evenly to a predictable number of fast operations.
- With the *Potential Method*, we “bank” time in a potential function, ϕ, on the fast operations, then “cash out” on the slow operations.
- Formally, we define $\Delta\phi_o$ as the change in potential after operation o. Then the amortized time is calculated as

  $$T_{amortized}(o) = T_{actual}(o) + C \cdot \Delta\phi_o$$

  for a constant C that disappears in big O.
- When we “cash out”, the amortized time is smaller than the actual time, and ϕ decreases (usually to 0).

Amortizing a Fibonacci Heap

- We define ϕ as

  $$\phi = t + 2m$$

  for a Fibonacci heap with t trees and m marked nodes.

Amortizing a Fibonacci Heap

- We define ϕ as

  $$\phi = t + 2m$$

  for a Fibonacci heap with t trees and m marked nodes.
- This looks really arbitrary, but we will soon see why it works.

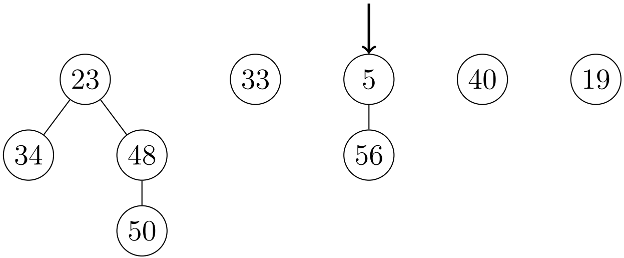
Fibonacci Heap Insert

- To insert, we simply add a new tree to the heap with the node we are adding as the root.

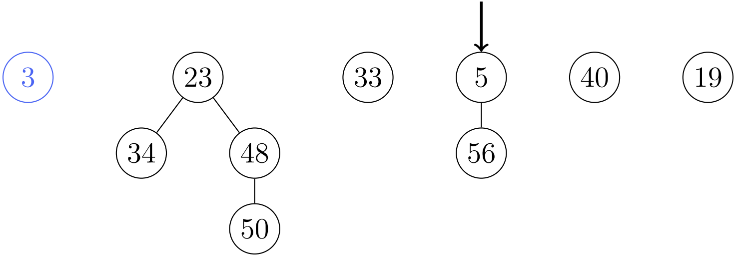
Fibonacci Heap Insert

- To insert, we simply add a new tree to the heap with the node we are adding as the root.
- We maintain a pointer to the minimum element, so if this is the new minimum, update the pointer.
- $O(1)$.

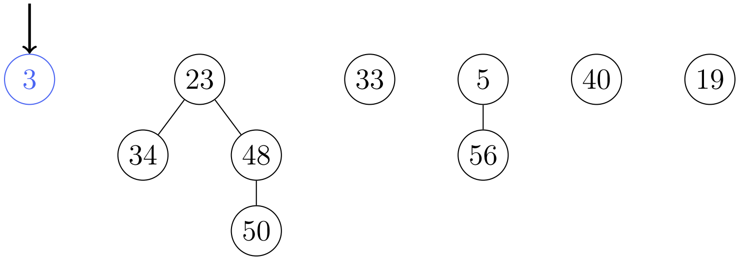
Fibonacci Heap Insert Example



Fibonacci Heap INSERT Example



Fibonacci Heap INSERT Example

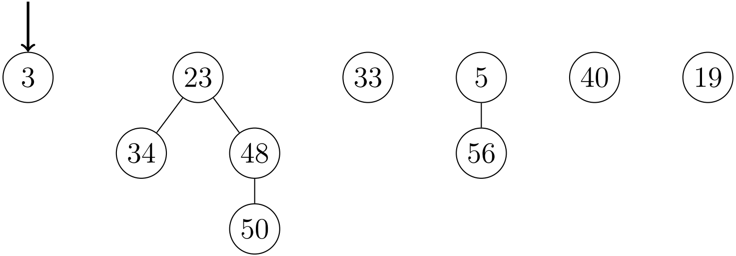


Fibonacci Heap FindMin

Because we maintain a pointer to the minimum element, accessing it is $O(1)$.

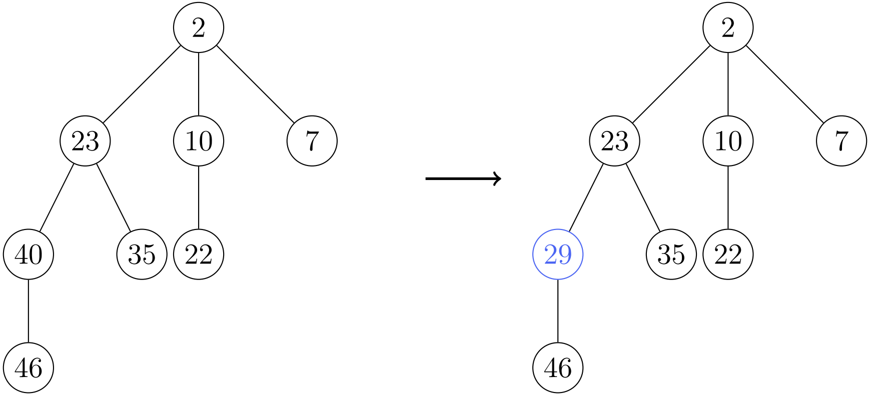
Fibonacci Heap FindMin

Because we maintain a pointer to the minimum element, accessing it is $O(1)$.



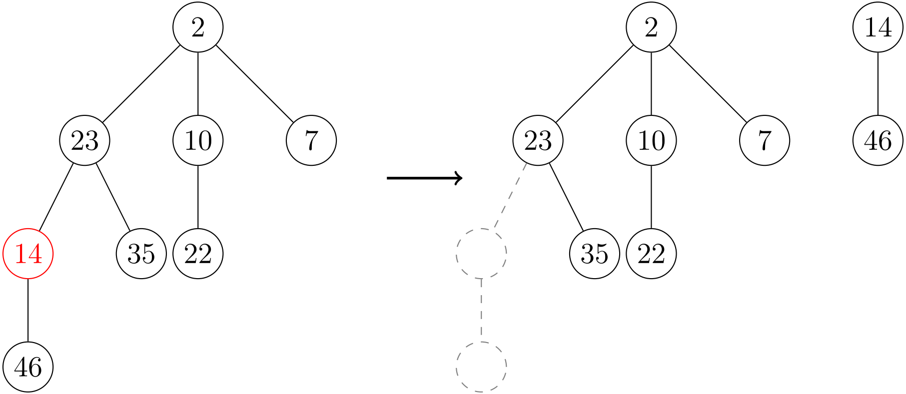
Fibonacci Heap DecreaseKey

1. Decrease the key. If the new value maintains the heap property, we are done.



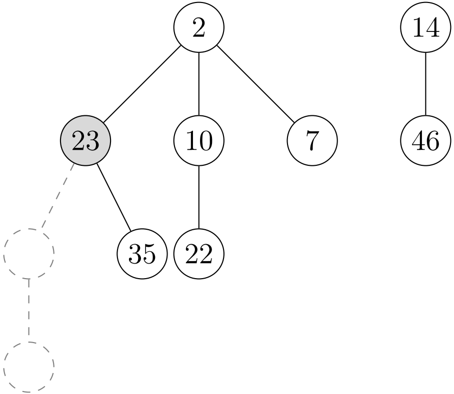
Fibonacci Heap DecreaseKey

2. Otherwise, cut the decreased key (and its subtree) out into a new tree.



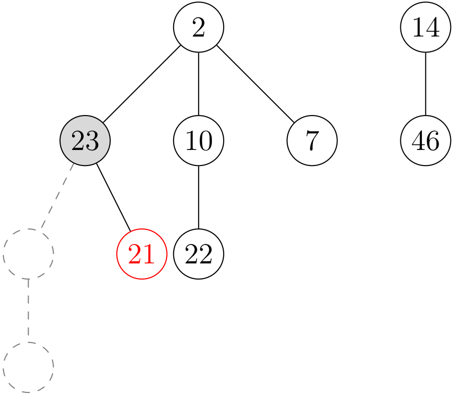
Fibonacci Heap DecreaseKey

3. If the parent was unmarked, mark it. Otherwise, cut it out, unmark it, and repeat for this node's parent.



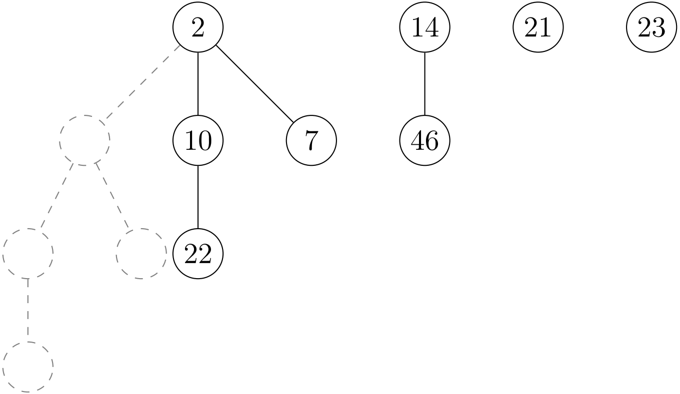
Fibonacci Heap DecreaseKey

Now, if we cut out 35, we would also cut out 23:



Fibonacci Heap DecreaseKey

Now, if we cut out 35, we would also cut out 23:



Fibonacci Heap DecreaseKey Analysis

- Suppose we made k cuts in total. Then we added k trees in $O(k)$.

Fibonacci Heap DecreaseKey Analysis

- Suppose we made k cuts in total. Then we added k trees in $O(k)$.
- Of all the nodes we cut out, only the first was unmarked, but all are unmarked now. Including the final extra node we marked, we have that marked nodes changed by $-(k-1)+1=-k+2$.

Fibonacci Heap DecreaseKey Analysis

- Suppose we made k cuts in total. Then we added k trees in $O(k)$.
- Of all the nodes we cut out, only the first was unmarked, but all are unmarked now. Including the final extra node we marked, we have that marked nodes changed by $-(k-1)+1=-k+2$.
- Remember $\phi = t + 2m$ for t trees and m marked nodes, and $T_{amortized}(o) = T_{actual}(o) + C \cdot \Delta\phi$.

Fibonacci Heap DecreaseKey Analysis

- Suppose we made k cuts in total. Then we added k trees in $O(k)$.
- Of all the nodes we cut out, only the first was unmarked, but all are unmarked now. Including the final extra node we marked, we have that marked nodes changed by $-(k-1)+1=-k+2$.
- Remember $\phi = t + 2m$ for t trees and m marked nodes, and $T_{amortized}(o) = T_{actual}(o) + C \cdot \Delta\phi$.
- Thus, $\Delta\phi = k + 2(-k+2) = -k+4$.

Fibonacci Heap DecreaseKey Analysis

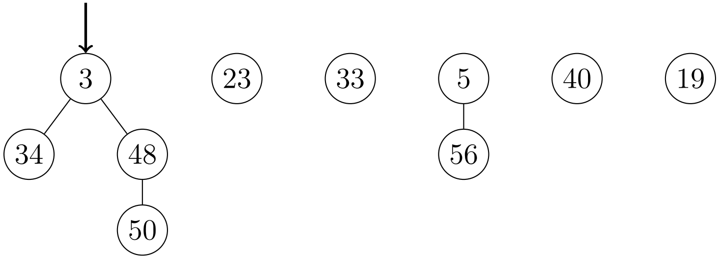
- Suppose we made k cuts in total. Then we added k trees in $O(k)$.
- Of all the nodes we cut out, only the first was unmarked, but all are unmarked now. Including the final extra node we marked, we have that marked nodes changed by $-(k-1)+1=-k+2$.
- Remember $\phi = t + 2m$ for t trees and m marked nodes, and $T_{amortized}(o) = T_{actual}(o) + C \cdot \Delta\phi$.
- Thus, $\Delta\phi = k + 2(-k+2) = -k+4$.
- So our amortized time is $O(1)$.

Fibonacci Heap DecreaseKey Analysis

- Suppose we made k cuts in total. Then we added k trees in $O(k)$.
- Of all the nodes we cut out, only the first was unmarked, but all are unmarked now. Including the final extra node we marked, we have that marked nodes changed by $-(k-1)+1=-k+2$.
- Remember $\phi = t + 2m$ for t trees and m marked nodes, and $T_{amortized}(o) = T_{actual}(o) + C \cdot \Delta\phi$.
- Thus, $\Delta\phi = k + 2(-k+2) = -k+4$.
- So our amortized time is $O(1)$.
- Notice our choice of ϕ was important here.

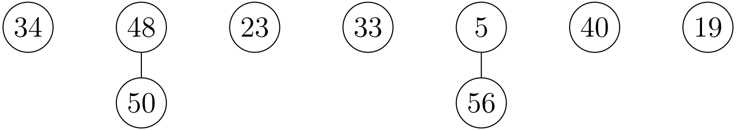
Fibonacci Heap DeleteMin Phase 1

Cut out the minimum node (which must be a root) and make its children roots of new trees.



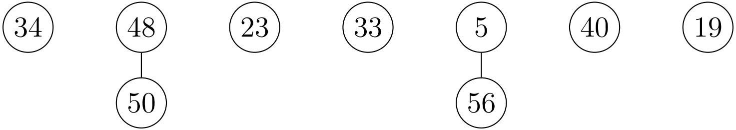
Fibonacci Heap DeleteMin Phase 1

Cut out the minimum node (which must be a root) and make its children roots of new trees.



Fibonacci Heap DeleteMin Phase 1

Cut out the minimum node (which must be a root) and make its children roots of new trees.



With d children, this operation is $O(d)$. We will see later that $O(d) = O(\log n)$ for a fibonacci heap with n nodes.

Fibonacci Heap DeleteMin Phase 2

- Now we need to update the minimum pointer.

Fibonacci Heap DeleteMin Phase 2

- Now we need to update the minimum pointer.
- Until now, we haven't enforced much structure, so in general, there might now be n roots, and finding the minimum is $O(n)$.

Fibonacci Heap DeleteMin Phase 2

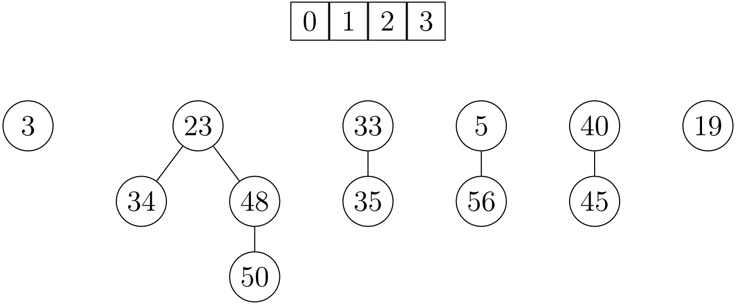
- Now we need to update the minimum pointer.
- Until now, we haven't enforced much structure, so in general, there might now be n roots, and finding the minimum is $O(n)$.
- However, if we "clean up," future calls to DeleteMin could be faster, and could give us a $\Delta\phi$ that helps in amortized analysis.

Fibonacci Heap DeleteMin Phase 2

- Now we need to update the minimum pointer.
- Until now, we haven't enforced much structure, so in general, there might now be n roots, and finding the minimum is $O(n)$.
- However, if we "clean up," future calls to DeleteMin could be faster, and could give us a $\Delta\phi$ that helps in amortized analysis.
- Like the last slide, I will assume the largest degree of any root is $O(\log n)$, and prove this later.

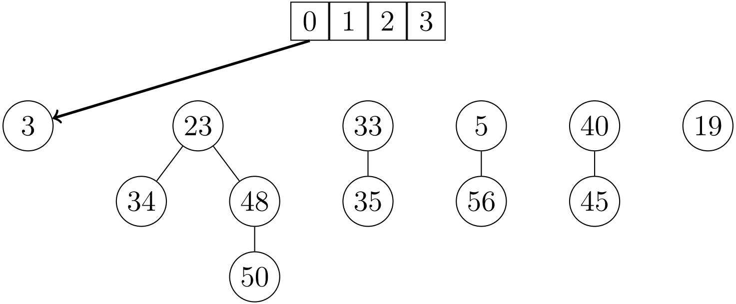
Fibonacci Heap DeleteMin Phase 2

1. Create an array of length $O(\log n)$ to track trees of every possible degree.



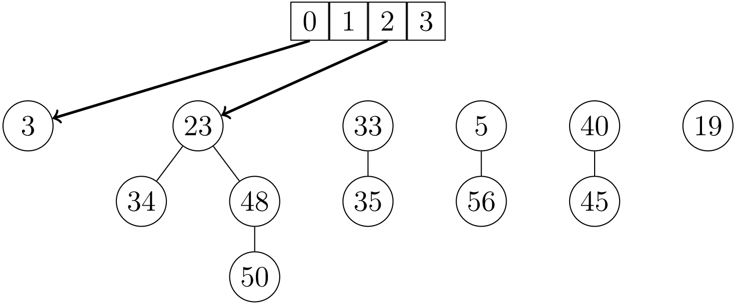
Fibonacci Heap DELETEMIN Phase 2

2. Put each tree into an index of the array. If there is already a tree there, merge them to create a tree of degree +1.



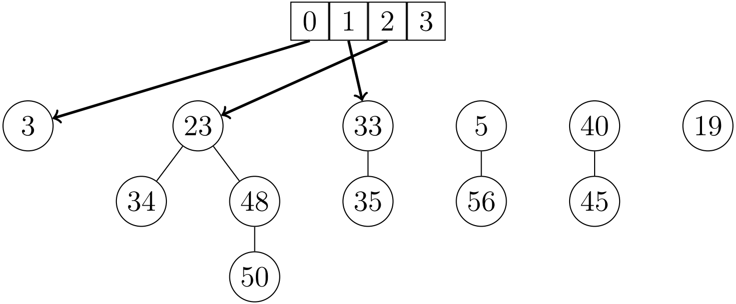
Fibonacci Heap DeleteMin Phase 2

2. Put each tree into an index of the array. If there is already a tree there, merge them to create a tree of degree +1.



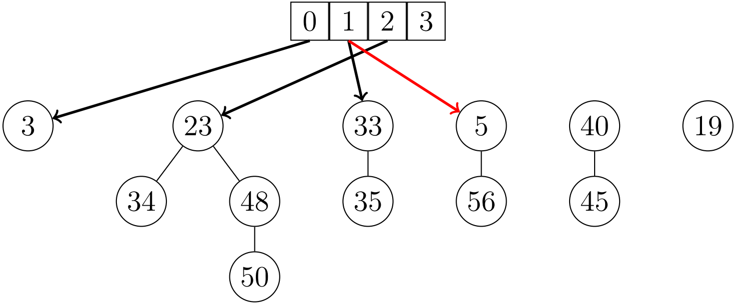
Fibonacci Heap DELETEMIN Phase 2

2. Put each tree into an index of the array. If there is already a tree there, merge them to create a tree of degree +1.



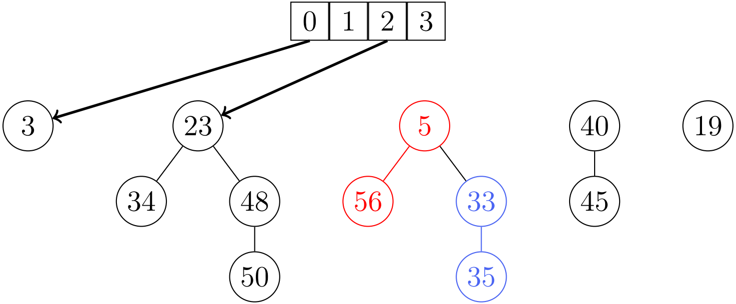
Fibonacci Heap DeleteMin Phase 2

2. Put each tree into an index of the array. If there is already a tree there, merge them to create a tree of degree +1.



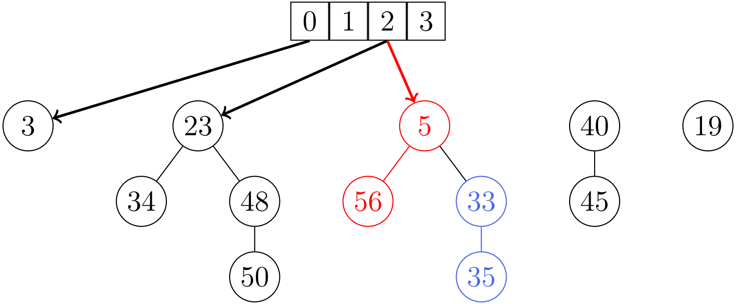
Fibonacci Heap DELETEMIN Phase 2

2. Put each tree into an index of the array. If there is already a tree there, merge them to create a tree of degree +1.



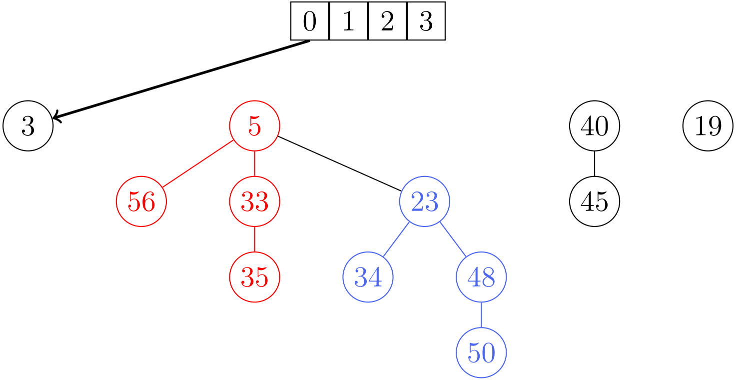
Fibonacci Heap DELETEMIN Phase 2

2. Put each tree into an index of the array. If there is already a tree there, merge them to create a tree of degree +1.



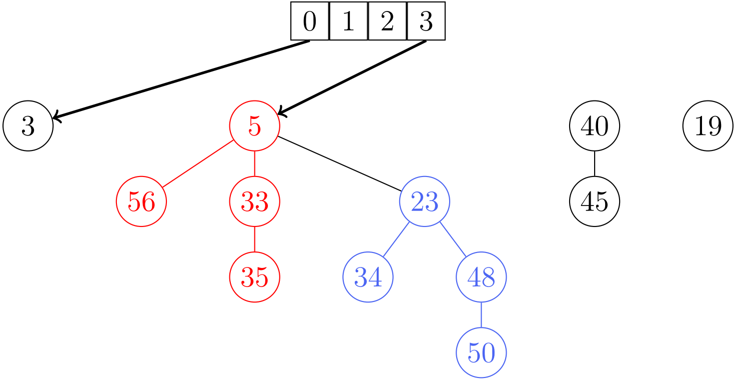
Fibonacci Heap DeleteMin Phase 2

2. Put each tree into an index of the array. If there is already a tree there, merge them to create a tree of degree +1.



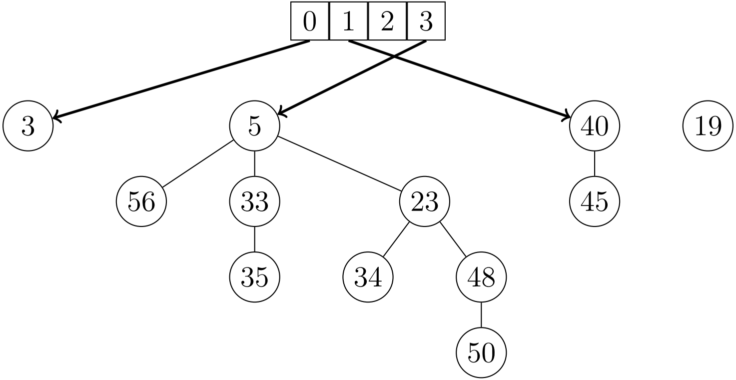
Fibonacci Heap DeleteMin Phase 2

2. Put each tree into an index of the array. If there is already a tree there, merge them to create a tree of degree +1.



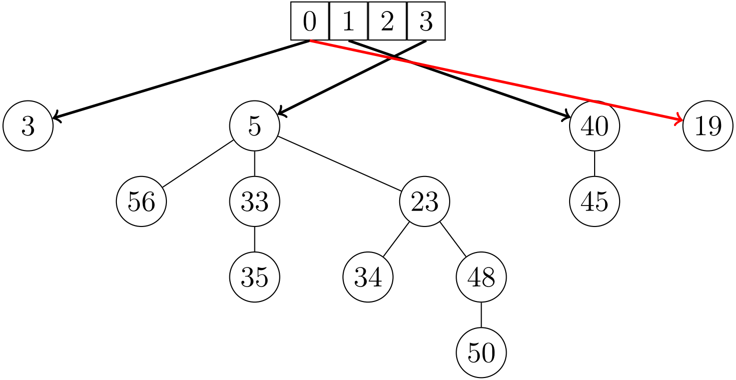
Fibonacci Heap DeleteMin Phase 2

2. Put each tree into an index of the array. If there is already a tree there, merge them to create a tree of degree +1.



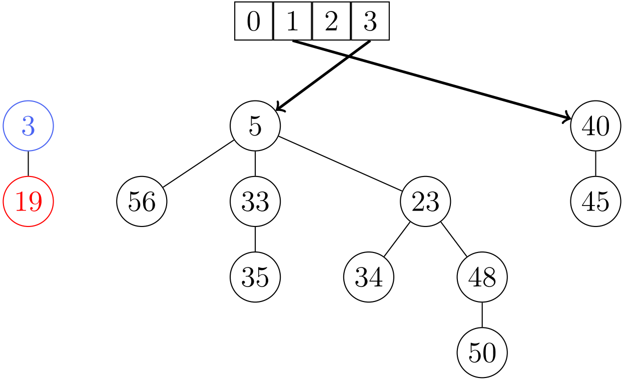
Fibonacci Heap DELETEMIN Phase 2

2. Put each tree into an index of the array. If there is already a tree there, merge them to create a tree of degree +1.



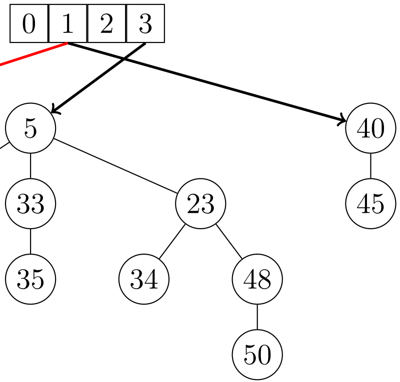
Fibonacci Heap DELETEMIN Phase 2

2. Put each tree into an index of the array. If there is already a tree there, merge them to create a tree of degree +1.



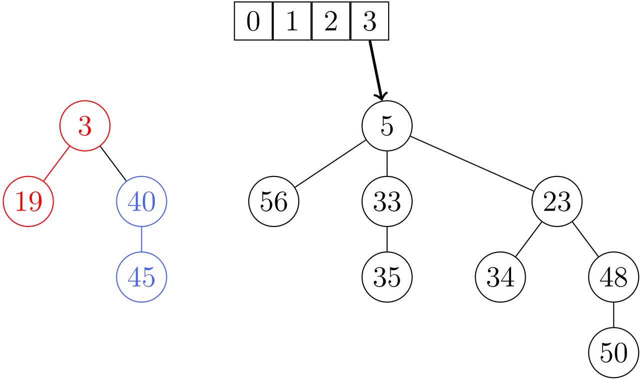
Fibonacci Heap DeleteMin Phase 2

2. Put each tree into an index of the array. If there is already a tree there, merge them to create a tree of degree +1.



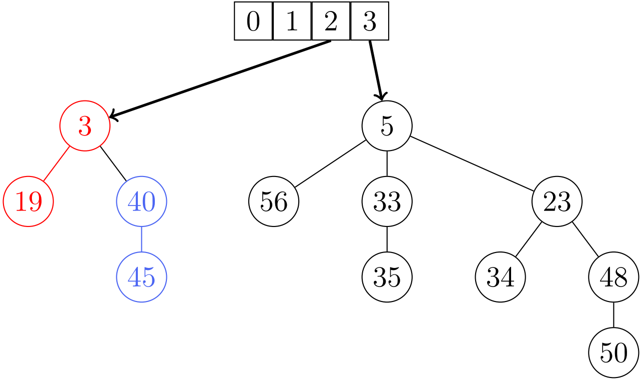
Fibonacci Heap DELETEMIN Phase 2

2. Put each tree into an index of the array. If there is already a tree there, merge them to create a tree of degree +1.



Fibonacci Heap DeleteMin Phase 2

2. Put each tree into an index of the array. If there is already a tree there, merge them to create a tree of degree +1.



Fibonacci Heap DeleteMin Phase 2 Analysis

- We need to allocate the array and visit each of m starting roots for $O(\log n + m)$ actual runtime.

Fibonacci Heap DeleteMin Phase 2 Analysis

- We need to allocate the array and visit each of m starting roots for $O(\log n + m)$ actual runtime.
- $\Delta\phi = \Delta(\#trees) = O(\log n) - m$. (final - initial)

Fibonacci Heap DeleteMin Phase 2 Analysis

- We need to allocate the array and visit each of m starting roots for $O(\log n + m)$ actual runtime.
- $\Delta\phi = \Delta(\#trees) = O(\log n) - m$. (final - initial)
- Thus, the amortized runtime for this phase is

$$O(\log n + m) + C \cdot \big(O(\log n) - m\big)$$

Fibonacci Heap DeleteMin Phase 2 Analysis

- We need to allocate the array and visit each of m starting roots for $O(\log n + m)$ actual runtime.
- $\Delta\phi = \Delta(\#trees) = O(\log n) - m$. (final - initial)
- Thus, the amortized runtime for this phase is

$$\begin{aligned} &O(\log n + m) + C \cdot \big(O(\log n) - m\big) \\ &= O(\log\ n) + O(m) + C \cdot O(\log n) - C \cdot m \end{aligned}$$

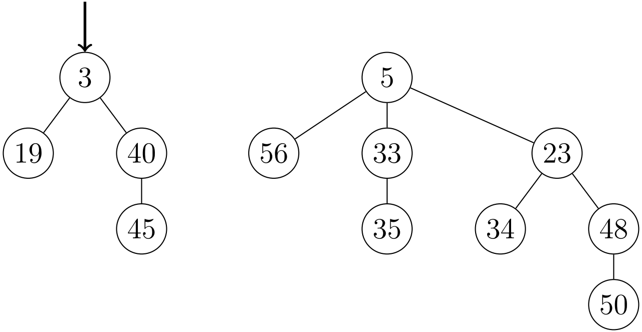
Fibonacci Heap DELETEMIN Phase 2 Analysis

- We need to allocate the array and visit each of m starting roots for $O(\log n + m)$ actual runtime.
- $\Delta\phi = \Delta(\#trees) = O(\log n) - m$. (final - initial)
- Thus, the amortized runtime for this phase is

$$\begin{aligned}
&O(\log n + m) + C \cdot \big(O(\log n) - m\big) \\
&= O(\log\ n) + O(m) + C \cdot O(\log n) - C \cdot m \\
&= O(\log n) \qquad\qquad \text{(with large enough } C\text{)}
\end{aligned}$$

Fibonacci Heap DELETEMIN Phase 3

Find the minimum of the $O(\log n)$ roots that remain in $O(\log n)$ time.



Fibonacci Heap DeleteMin Recap

Of all 3 phases, the slowest runtime was $O(\log\ n)$, so the overall runtime is $O(\log n)$.

Questions?

Fibonacci Heap Degree Bound Proof

- Earlier, we assumed that the largest degree of any root is $O(\log n)$. How do we know? This is where Fibonacci comes in.

Fibonacci Heap Degree Bound Proof

- Earlier, we assumed that the largest degree of any root is $O(\log n)$. How do we know? This is where Fibonacci comes in.
- We focus on one tree with degree d and n nodes total. We want

$$d \leq O(\log n)$$

Fibonacci Heap Degree Bound Proof

- Earlier, we assumed that the largest degree of any root is $O(\log n)$. How do we know? This is where Fibonacci comes in.
- We focus on one tree with degree d and n nodes total. We want

$$\begin{aligned} d &\leq O(\log n) \\ d &\leq \log_a n && \text{(for some } a > 1\text{)} \end{aligned}$$

Fibonacci Heap Degree Bound Proof

- Earlier, we assumed that the largest degree of any root is $O(\log n)$. How do we know? This is where Fibonacci comes in.
- We focus on one tree with degree d and n nodes total. We want

$$\begin{aligned}
d &\leq O(\log n) \\
d &\leq \log_a n && \text{(for some } a > 1\text{)} \\
d \cdot \log a &\leq \log n && \text{(if } a \leq 1\text{, the } \leq \text{ would flip)}
\end{aligned}$$

Fibonacci Heap Degree Bound Proof

- Earlier, we assumed that the largest degree of any root is $O(\log n)$. How do we know? This is where Fibonacci comes in.
- We focus on one tree with degree d and n nodes total. We want

$$
\begin{aligned}
d &\leq O(\log n) \\
d &\leq \log_a n && \text{(for some } a > 1\text{)} \\
d \cdot \log a &\leq \log n && \text{(if } a \leq 1\text{, the } \leq \text{ would flip)} \\
\log a^d &\leq \log n
\end{aligned}
$$

Fibonacci Heap Degree Bound Proof

- Earlier, we assumed that the largest degree of any root is $O(\log n)$. How do we know? This is where Fibonacci comes in.
- We focus on one tree with degree d and n nodes total. We want

$$\begin{aligned} d &\le O(\log n) \\ d &\le \log_a n && \text{(for some } a > 1\text{)} \\ d \cdot \log a &\le \log n && \text{(if } a \le 1\text{, the } \le \text{ would flip)} \\ \log a^d &\le \log n \\ a^d &\le n \end{aligned}$$

Fibonacci Heap Degree Bound Proof

- Earlier, we assumed that the largest degree of any root is $O(\log n)$. How do we know? This is where Fibonacci comes in.
- We focus on one tree with degree d and n nodes total. We want

$$
\begin{aligned}
d &\leq O(\log n) \\
d &\leq \log_a n && \text{(for some } a > 1\text{)} \\
d \cdot \log a &\leq \log n && \text{(if } a \leq 1 \text{, the } \leq \text{ would flip)} \\
\log a^d &\leq \log n \\
a^d &\leq n
\end{aligned}
$$

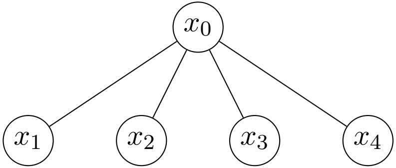
- Thus, we need to show that n grows exponentially with respect to d.

Fibonacci Heap Degree Bound Proof

- To do this, let's look for a lower bound on n for a given d.

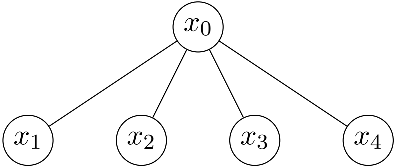
Fibonacci Heap Degree Bound Proof

- To do this, let's look for a lower bound on n for a given d.
- Consider a tree of degree 4:



Fibonacci Heap Degree Bound Proof

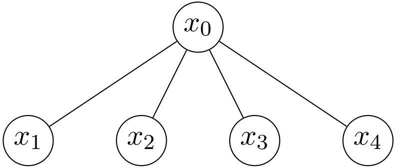
- To do this, let's look for a lower bound on n for a given d.
- Consider a tree of degree 4:



- When x_3 was added, it must have been degree 2.

Fibonacci Heap Degree Bound Proof

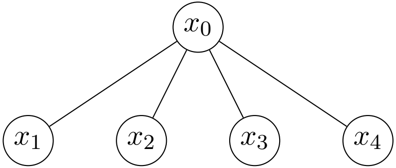
- To do this, let's look for a lower bound on n for a given d.
- Consider a tree of degree 4:



- When x_3 was added, it must have been degree 2.
- Because each node can have at most one child removed, x_3 must now have degree at least 1. Similarly, x_4 must have degree at least 2.

Fibonacci Heap Degree Bound Proof

- To do this, let's look for a lower bound on n for a given d.
- Consider a tree of degree 4:



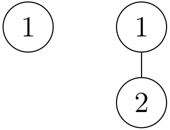
- When x_3 was added, it must have been degree 2.
- Because each node can have at most one child removed, x_3 must now have degree at least 1. Similarly, x_4 must have degree at least 2.
- In general, the i^{th} child must have degree at least $i - 2$.

Fibonacci Heap Degree Bound Proof

- We can use this to build up the minimum trees for each degree.

Fibonacci Heap Degree Bound Proof

- We can use this to build up the minimum trees for each degree.
- Minimum for degrees 0 and 1 (base cases) are trivial:

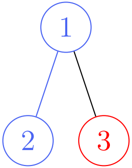


Fibonacci Heap Degree Bound Proof

- Then for tree of degree d we can add tree of degree $d - 2$ as a child of tree $d - 1$.

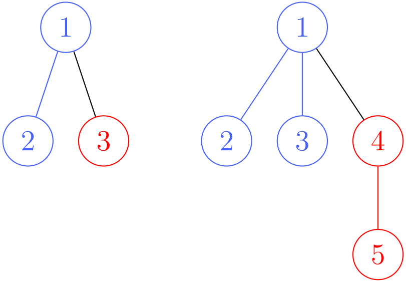
Fibonacci Heap Degree Bound Proof

- Then for tree of degree d we can add tree of degree $d - 2$ as a child of tree $d - 1$.
- Degrees 2–4 are as follows:



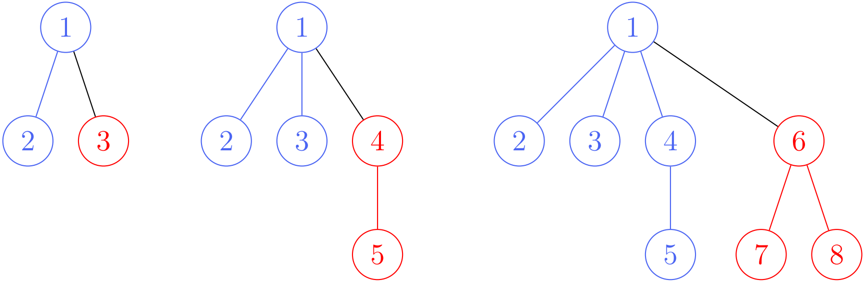
Fibonacci Heap Degree Bound Proof

- Then for tree of degree d we can add tree of degree $d - 2$ as a child of tree $d - 1$.
- Degrees 2–4 are as follows:



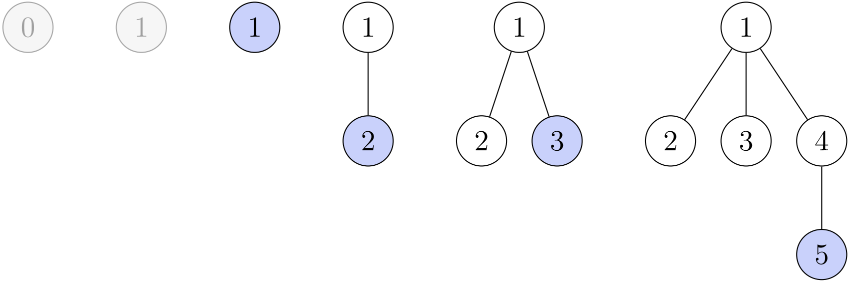
Fibonacci Heap Degree Bound Proof

- Then for tree of degree d we can add tree of degree $d - 2$ as a child of tree $d - 1$.
- Degrees 2–4 are as follows:



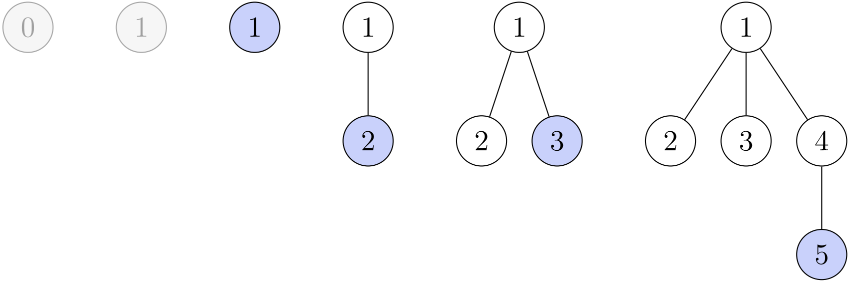
Fibonacci Heap Degree Bound Proof

- Fibonacci!



Fibonacci Heap Degree Bound Proof

- Fibonacci!



- We know that the growth of Fibonacci approaches $\Phi \approx 1.618$, so $n \geq \Phi^d$. which satisfies our goal.

Recap

	Binary Heap	Binomial Heap	Fibonacci Heap
Insert	$O(\log n)$	$O(1)$	$O(1)$
FindMin	$O(1)$	$O(1)$	$O(1)$
DeleteMin	$O(\log n)$	$O(\log n)$	$O(\log n)$
DecreaseKey	$O(\log n)$	$O(\log n)$	$O(1)$

In practice...

- Fibonacci heaps are usually implemented with pointers from parent to child, child to parent, and circularly between siblings.

In practice...

- Fibonacci heaps are usually implemented with pointers from parent to child, child to parent, and circularly between siblings.
- This leads to large big-O constants and heavy reliance on pointers that gives much worse cache performance than binary heaps.

In practice...

- Fibonacci heaps are usually implemented with pointers from parent to child, child to parent, and circularly between siblings.
- This leads to large big-O constants and heavy reliance on pointers that gives much worse cache performance than binary heaps.
- So, Fibonacci heaps are not often used, because they are too slow in practice.

In practice...

- Fibonacci heaps are usually implemented with pointers from parent to child, child to parent, and circularly between siblings.
- This leads to large big-O constants and heavy reliance on pointers that gives much worse cache performance than binary heaps.
- So, Fibonacci heaps are not often used, because they are too slow in practice.
- Fibonacci heaps can still be faster for really large amounts of data.

Questions?

Computers are useless. They only give you the answers.

— Pablo Picasso (1979)

Bibliography I

Wikimedia Foundation. (2024, January 17). *Fibonacci heap*. Wikipedia. https://en.wikipedia.org/wiki/Fibonacci_heap