5.2 Fibonacci Heaps

Frank Stajano

Thomas Sauerwald
## Priority Queues Overview

<table>
<thead>
<tr>
<th>Operation</th>
<th>Linked list</th>
<th>Binary heap</th>
<th>Binomial heap</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MAKE-HEAP</strong></td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td><strong>INSERT</strong></td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td><strong>MINIMUM</strong></td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td><strong>EXTRACT-MIN</strong></td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td><strong>MERGE</strong></td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td><strong>DECREASE-KEY</strong></td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td><strong>DELETE</strong></td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>
## Priority Queues Overview

<table>
<thead>
<tr>
<th>Operation</th>
<th>Linked list</th>
<th>Binary heap</th>
<th>Binomial heap</th>
<th>Fibon. heap</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MAKE-HEAP</strong></td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td><strong>INSERT</strong></td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td><strong>MINIMUM</strong></td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td><strong>EXTRACT-MIN</strong></td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td><strong>MERGE</strong></td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td><strong>DECREASE-KEY</strong></td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td><strong>DELETE</strong></td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>
## Binomial Heap vs. Fibonacci Heap: Costs

<table>
<thead>
<tr>
<th>Operation</th>
<th>Binomial heap</th>
<th>Fibonacci heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAKE-HEAP</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>INSERT</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>MINIMUM</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>EXTRACT-MIN</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>MERGE</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>DECREASE-KEY</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>DELETE</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

All these cost bounds hold if $n$ is the size of the heap.
## Binomial Heap vs. Fibonacci Heap: Costs

<table>
<thead>
<tr>
<th>Operation</th>
<th>Binomial heap actual cost</th>
<th>Fibonacci heap amortized cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAKE-HEAP</td>
<td>$\mathcal{O}(1)$</td>
<td>$\mathcal{O}(1)$</td>
</tr>
<tr>
<td>INSERT</td>
<td>$\mathcal{O}(\log n)$</td>
<td>$\mathcal{O}(1)$</td>
</tr>
<tr>
<td>MINIMUM</td>
<td>$\mathcal{O}(\log n)$</td>
<td>$\mathcal{O}(1)$</td>
</tr>
<tr>
<td>EXTRACT-MIN</td>
<td>$\mathcal{O}(\log n)$</td>
<td>$\mathcal{O}(\log n)$</td>
</tr>
<tr>
<td>MERGE</td>
<td>$\mathcal{O}(\log n)$</td>
<td>$\mathcal{O}(1)$</td>
</tr>
<tr>
<td>DECREASE-KEY</td>
<td>$\mathcal{O}(\log n)$</td>
<td>$\mathcal{O}(1)$</td>
</tr>
<tr>
<td>DELETE</td>
<td>$\mathcal{O}(\log n)$</td>
<td>$\mathcal{O}(\log n)$</td>
</tr>
</tbody>
</table>

All these cost bounds hold if $n$ is the size of the heap.
## Binomial Heap vs. Fibonacci Heap: Costs

<table>
<thead>
<tr>
<th>Operation</th>
<th>Binomial heap actual cost</th>
<th>Fibonacci heap amortized cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAKE-HEAP</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>INSERT</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>MINIMUM</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>EXTRACT-MIN</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>MERGE</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>DECREASE-KEY</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>DELETE</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

All these cost bounds hold if $n$ is the size of the heap.
## Binomial Heap vs. Fibonacci Heap: Costs

<table>
<thead>
<tr>
<th>Operation</th>
<th>Binomial heap actual cost</th>
<th>Fibonacci heap amortized cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAKE-HEAP</td>
<td>$\mathcal{O}(1)$</td>
<td>$\mathcal{O}(1)$</td>
</tr>
<tr>
<td>INSERT</td>
<td>$\mathcal{O}(\log n)$</td>
<td>$\mathcal{O}(1)$</td>
</tr>
<tr>
<td>MINIMUM</td>
<td>$\mathcal{O}(\log n)$</td>
<td>$\mathcal{O}(1)$</td>
</tr>
<tr>
<td>EXTRACT-MIN</td>
<td>$\mathcal{O}(\log n)$</td>
<td>$\mathcal{O}(\log n)$</td>
</tr>
<tr>
<td>MERGE</td>
<td>$\mathcal{O}(\log n)$</td>
<td>$\mathcal{O}(1)$</td>
</tr>
<tr>
<td>DECREASE-KEY</td>
<td>$\mathcal{O}(\log n)$</td>
<td>$\mathcal{O}(1)$</td>
</tr>
<tr>
<td>DELETE</td>
<td>$\mathcal{O}(\log n)$</td>
<td>$\mathcal{O}(\log n)$</td>
</tr>
</tbody>
</table>

All these cost bounds hold if $n$ is the size of the heap.

### Binomial Heap:

$k/2$ DECREASE-KEY + $k/2$ INSERT
## Binomial Heap vs. Fibonacci Heap: Costs

<table>
<thead>
<tr>
<th>Operation</th>
<th>Binomial heap actual cost</th>
<th>Fibonacci heap amortized cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAKE-HEAP</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>INSERT</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>MINIMUM</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>EXTRACT-MIN</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>MERGE</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>DECREASE-KEY</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>DELETE</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

All these cost bounds hold if \( n \) is the size of the heap.

### Binomial Heap: \( k/2 \text{ DECREASE-KEY} + k/2 \text{ INSERT} \)
- \( c_1 = c_2 = \cdots = c_k = O(\log n) \)
## Binomial Heap vs. Fibonacci Heap: Costs

<table>
<thead>
<tr>
<th>Operation</th>
<th>Binomial heap actual cost</th>
<th>Fibonacci heap amortized cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAKE-HEAP</td>
<td>$\mathcal{O}(1)$</td>
<td>$\mathcal{O}(1)$</td>
</tr>
<tr>
<td>INSERT</td>
<td>$\mathcal{O}(\log n)$</td>
<td>$\mathcal{O}(1)$</td>
</tr>
<tr>
<td>MINIMUM</td>
<td>$\mathcal{O}(\log n)$</td>
<td>$\mathcal{O}(1)$</td>
</tr>
<tr>
<td>EXTRACT-MIN</td>
<td>$\mathcal{O}(\log n)$</td>
<td>$\mathcal{O}(\log n)$</td>
</tr>
<tr>
<td>MERGE</td>
<td>$\mathcal{O}(\log n)$</td>
<td>$\mathcal{O}(1)$</td>
</tr>
<tr>
<td>DECREASE-KEY</td>
<td>$\mathcal{O}(\log n)$</td>
<td>$\mathcal{O}(1)$</td>
</tr>
<tr>
<td>DELETE</td>
<td>$\mathcal{O}(\log n)$</td>
<td>$\mathcal{O}(\log n)$</td>
</tr>
</tbody>
</table>

All these cost bounds hold if $n$ is the size of the heap.

### Binomial Heap: $k/2$ DECREASE-KEY + $k/2$ INSERT

- $c_1 = c_2 = \cdots = c_k = \mathcal{O}(\log n)$
- $\sum_{i=1}^{k} c_i = \mathcal{O}(k \log n)$
## Binomial Heap vs. Fibonacci Heap: Costs

<table>
<thead>
<tr>
<th>Operation</th>
<th>Binomial heap actual cost</th>
<th>Fibonacci heap amortized cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAKE-HEAP</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>INSERT</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>MINIMUM</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>EXTRACT-MIN</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>MERGE</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>DECREASE-KEY</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>DELETE</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

Binomial Heap: $k/2$ DECREASE-KEY + $k/2$ INSERT

- $c_1 = c_2 = \cdots = c_k = O(\log n)$

$\Rightarrow \sum_{i=1}^{k} c_i = O(k \log n)$

Fibonacci Heap: $k/2$ DECREASE-KEY + $k/2$ INSERT
## Binomial Heap vs. Fibonacci Heap: Costs

<table>
<thead>
<tr>
<th>Operation</th>
<th>Binomial heap actual cost</th>
<th>Fibonacci heap amortized cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAKE-HEAP</td>
<td>$\mathcal{O}(1)$</td>
<td>$\mathcal{O}(1)$</td>
</tr>
<tr>
<td>INSERT</td>
<td>$\mathcal{O}(\log n)$</td>
<td>$\mathcal{O}(1)$</td>
</tr>
<tr>
<td>MINIMUM</td>
<td>$\mathcal{O}(\log n)$</td>
<td>$\mathcal{O}(1)$</td>
</tr>
<tr>
<td>EXTRACT-MIN</td>
<td>$\mathcal{O}(\log n)$</td>
<td>$\mathcal{O}(\log n)$</td>
</tr>
<tr>
<td>MERGE</td>
<td>$\mathcal{O}(\log n)$</td>
<td>$\mathcal{O}(1)$</td>
</tr>
<tr>
<td>DECREASE-KEY</td>
<td>$\mathcal{O}(\log n)$</td>
<td>$\mathcal{O}(1)$</td>
</tr>
<tr>
<td>DELETE</td>
<td>$\mathcal{O}(\log n)$</td>
<td>$\mathcal{O}(\log n)$</td>
</tr>
</tbody>
</table>

All these cost bounds hold if $n$ is the size of the heap.

**Binomial Heap:** $k/2$ DECREASE-KEY + $k/2$ INSERT
- $c_1 = c_2 = \cdots = c_k = \mathcal{O}(\log n)$
- $\sum_{i=1}^{k} c_i = \mathcal{O}(k \log n)$

**Fibonacci Heap:** $k/2$ DECREASE-KEY + $k/2$ INSERT
- $\tilde{c}_1 = \tilde{c}_2 = \cdots = \tilde{c}_k = \mathcal{O}(1)$
## Binomial Heap vs. Fibonacci Heap: Costs

<table>
<thead>
<tr>
<th>Operation</th>
<th>Binomial heap actual cost</th>
<th>Fibonacci heap amortized cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAKE-HEAP</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>INSERT</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>MINIMUM</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>EXTRACT-MIN</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>MERGE</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>DECREASE-KEY</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>DELETE</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

All these cost bounds hold if $n$ is the size of the heap.

### Binomial Heap: $k/2$ DECREASE-KEY + $k/2$ INSERT

- $c_1 = c_2 = \cdots = c_k = O(\log n)$
- $\sum_{i=1}^{k} c_i = O(k \log n)$

### Fibonacci Heap: $k/2$ DECREASE-KEY + $k/2$ INSERT

- $\tilde{c}_1 = \tilde{c}_2 = \cdots = \tilde{c}_k = O(1)$
- $\sum_{i=1}^{k} c_i \leq \sum_{i=1}^{k} \tilde{c}_i = O(k)$
Actual vs. Amortized Cost

$$\sum_{i=1}^{k} \tilde{c}_i$$

Potential $\geq 0$, but should be also as small as possible.
Actual vs. Amortized Cost

\[ \sum_{i=1}^{k} \tilde{c}_i \quad \sum_{i=1}^{k} c_i \]

Potential \[ \sum_{i=1}^{k} \tilde{c}_i \] \( \geq 0 \), but should be also as small as possible
Actual vs. Amortized Cost

\[ \sum_{i=1}^{k} \tilde{c}_i \quad \sum_{i=1}^{k} c_i \]

Potential \( \sum_{i=1}^{k} \tilde{c}_i \geq 0 \), but should be also as small as possible.
Actual vs. Amortized Cost

\[ \sum_{i=1}^{k} \tilde{c}_i \geq 0, \text{ but should be also as small as possible} \]
Outline

Structure

Operations

Glimpse at the Analysis
Reminder: Binomial Heaps

Binomial Trees

- $B(0)$
- $B(1)$
- $B(2)$
- $B(3)$
- $B(k)$

Binomial Heap is a collection of binomial trees of different orders, each of which obeys the heap property.
Binomial Trees

- Binomial Heap is a collection of binomial trees of different orders, each of which obeys the heap property
- Operations:

**Operations:**

- **MERGE**: Merge two binomial heaps using Binary Addition Procedure
- **INSERT**: Add \( B(0) \) and perform a **MERGE**
- **EXTRACT-MIN**: Find tree with minimum key, cut it and perform a **MERGE**
- **DECREASE-KEY**: The same as in a binary heap
Reminder: Binomial Heaps

Binomial Trees

- $B(0)$
- $B(1)$
- $B(2)$
- $B(3)$
- $B(k)$

Binomial Heaps

- Binomial Heap is a collection of binomial trees of different orders, each of which obeys the *heap property*

- Operations:
  - **MERGE**: Merge two binomial heaps using *Binary Addition Procedure*
  - **INSERT**: Add $B(0)$ and perform a **MERGE**
  - **EXTRACT-MIN**: Find tree with minimum key, cut it and perform a **MERGE**
  - **DECREASE-KEY**: The same as in a binary heap

5.2: Fibonacci Heaps
Merging two Binomial Heaps

\[ \begin{array}{cccccc}
3 & 5 & 15 & + & 1 & 4 \\
6 & 8 & 7 & & 12 & 17  \\
10 & & & & 13 & 16  \\
\end{array} \]

\[ \begin{array}{cccccccc}
0 & 0 & 1 & 1 & 1 & = 7  \\
0 & 1 & 0 & 1 & 1 & = 11  \\
1 & 1 & 1 & 1  \\
\hline
1 & 0 & 0 & 1 & 0 & = 18  \\
\end{array} \]
Merging two Binomial Heaps

\[\begin{array}{ccc}
3 & 5 & 15 \\
6 & 8 & 7 \\
10 & & \\
\end{array} \quad + \quad \begin{array}{ccc}
1 & 4 & 9 \\
11 & 12 & 17 \\
13 & 16 & \\
\end{array} \quad = \begin{array}{ccc}
14 & 18 & \\
2 & & \\
\end{array} \]

\[
\begin{array}{ccccccc}
0 & 0 & 1 & 1 & 1 & 1 & = 7 \\
0 & 1 & 0 & 1 & 1 & & = 11 \\
1 & 1 & 1 & 1 & & & \\
\hline
1 & 0 & 0 & 1 & 0 & & = 18 \\
\end{array}
\]

5.2: Fibonacci Heaps
Merging two Binomial Heaps

3
6
8
10

5
7

15

1
4
9
11

12
17
13
16

14
18

2

0 0 1 1 1 = 7
0 1 0 1 1 = 11
1 1 1 1

1 0 0 1 0 = 18

5.2: Fibonacci Heaps
Merging two Binomial Heaps

\[ \begin{array}{c}
0 & 0 & 1 & 1 & 1 & = 7 \\
0 & 1 & 0 & 1 & 1 & = 11 \\
1 & 1 & 1 & 1 & & \\
1 & 0 & 0 & 1 & 0 & = 18 \\
\end{array} \]
Merging two Binomial Heaps

\[ \begin{array}{c}
3 & 5 & 15 \\
6 & 8 & 7 \\
10 & & \\
\end{array} \quad + \quad \begin{array}{c}
1 & 4 & 9 & 11 \\
12 & 17 & 13 & 16 \\
14 & 18 & & \\
2 & 15 & & \\
\end{array} \]

0 0 1 1 1 = 7
0 1 0 1 1 = 11
1 1 1 1 = 18
1 0 0 1 0 = 18
Merging two Binomial Heaps

\[
\begin{array}{c}
3 \\
6 \\
8 \\
10
\end{array}
\quad \begin{array}{c}
5 \\
7 \\
15
\end{array}
\quad \begin{array}{c}
1 \\
4 \\
9 \\
11
\end{array}
\quad \begin{array}{c}
14 \\
18
\end{array}
\quad \begin{array}{c}
2
\end{array}
\]

\[
+ 
\]

\[
\begin{array}{c}
1 \\
4 \\
9 \\
11
\end{array}
\quad \begin{array}{c}
12 \\
17 \\
13
\end{array}
\quad \begin{array}{c}
16
\end{array}
\]

\[
\begin{array}{c}
1 \\
0 \\
0 \\
1 \\
1 \\
1
\end{array}
\quad \begin{array}{c}
0 \\
1 \\
0 \\
1 \\
1 \\
1
\end{array}
\quad \begin{array}{c}
1 \\
1 \\
1 \\
1
\end{array}
\]

\[
\begin{array}{c}
0 \\
0 \\
1 \\
1 \\
1
\end{array}
\quad \begin{array}{c}
0 \\
1 \\
0 \\
1 \\
1
\end{array}
\quad \begin{array}{c}
1 \\
1 \\
1 \\
1
\end{array}
\]

\[
= 7 \\
= 11 \\
= 18
\]
Merging two Binomial Heaps

\[
\begin{align*}
3 & \quad 5 & \quad 15 \\
6 & \quad 8 & \quad 7 \\
10 & & \\
\hline
1 & \quad 4 & \quad 9 \\
4 & \quad 9 & \quad 11 \\
12 & \quad 17 & \quad 13 \\
16 & & \\
\hline
0 & 0 & 1 & 1 & 1 = 7 \\
0 & 1 & 0 & 1 & 1 = 11 \\
1 & 1 & 1 & 1 \\
\hline
1 & 0 & 0 & 1 & 0 = 18
\end{align*}
\]

5.2: Fibonacci Heaps
Merging two Binomial Heaps

```
3 6 8 10
5 7
15

+ 1
4 9 11
12 17 13
16

14 18
2
```

```
0 0 1 1 1 = 7
0 1 0 1 1 = 11
1 1 1 1

1 0 0 1 0 = 18
```
Merging two Binomial Heaps

\[ \begin{array}{c}
\begin{array}{c}
3 \\
5 \\
15
\end{array} \\
\begin{array}{c}
6 \\
7 \\
10
\end{array} \\
\begin{array}{c}
12 \\
17 \\
16
\end{array} \\
\begin{array}{c}
4 \\
9 \\
11
\end{array} \\
\begin{array}{c}
13
\end{array}
\end{array} \]

\[ \begin{array}{c}
0 \ 0 \ 1 \ 1 \ 1 = 7 \\
0 \ 1 \ 0 \ 1 \ 1 = 11 \\
1 \ 1 \ 1 \ 1 \\
1 \ 0 \ 0 \ 1 \ 0 = 18
\end{array} \]
Merging two Binomial Heaps

0 0 1 1 1 = 7
0 1 0 1 1 = 11
1 1 1 1

1 0 0 1 0 = 18

3 6 8 10
5 7

1 4 9 11
12 17 13 16
14 18

5.2: Fibonacci Heaps
Merging two Binomial Heaps

\[
\begin{align*}
3 & \quad 5 & \quad 15 \\
6 & \quad 8 & \quad 10 \\
7 & \quad & \\
\hline
0 & 0 & 1 & 1 & 1 & = 7 \\
0 & 1 & 0 & 1 & 1 & = 11 \\
1 & 1 & 1 & 1 & \\
\hline
1 & 0 & 0 & 1 & 0 & = 18
\end{align*}
\]
Merging two Binomial Heaps

\[
\begin{array}{c}
3 \\
6 \\
\quad 8 \\
\quad \quad 10 \\
\end{array}
\quad +
\quad \begin{array}{c}
1 \\
4 \\
9 \\
12 \\
\quad 13 \\
\quad 16 \\
\end{array}
\quad +
\quad \begin{array}{c}
14 \\
18 \\
2 \\
15 \\
\end{array}
\]

\[
\begin{array}{c}
0 \\
1 \\
1 \\
1 \\
1 \\
\end{array}
\qquad = 7
\]

\[
\begin{array}{c}
0 \\
1 \\
0 \\
1 \\
1 \\
\end{array}
\qquad = 11
\]

\[
\begin{array}{c}
1 \\
1 \\
1 \\
1 \\
\end{array}
\qquad = 18
\]

\[
\begin{array}{c}
0 \\
0 \\
1 \\
1 \\
1 \\
\end{array}
\qquad = 18
\]

5.2: Fibonacci Heaps
Merging two Binomial Heaps

![Diagram of merging two binomial heaps](image)

\[0 \ 0 \ 1 \ 1 \ 1 = 7\]
\[0 \ 1 \ 0 \ 1 \ 1 = 11\]
\[1 \ 1 \ 1 \ 1\]
\[1 \ 0 \ 0 \ 1 \ 0 = 18\]
Merging two Binomial Heaps

\[
\begin{align*}
0 & \ 0 & \ 1 & \ 1 & \ 1 & = 7 \\
0 & \ 1 & \ 0 & \ 1 & \ 1 & = 11 \\
1 & \ 1 & \ 1 & \ 1 & & \\
\hline
1 & \ 0 & \ 0 & \ 1 & \ 0 & = 18
\end{align*}
\]
Merging two Binomial Heaps

\[ \begin{array}{c}
3 \\
6 \\
8 \\
10 \\
\hline
5 \\
7 \\
\hline
15 \\
\end{array} \quad + \quad \begin{array}{c}
1 \\
4 \\
9 \\
11 \\
\hline
12 \\
13 \\
17 \\
16 \\
\hline
14 \\
18 \\
\end{array} \]

\[ \begin{array}{c}
0 \\
0 \\
1 \\
1 \\
1 \\
\hline
1 \\
0 \\
0 \\
1 \\
0 \\
\hline
1 \\
0 \\
0 \\
1 \\
0 \\
\end{array} = 7 \]

\[ \begin{array}{c}
0 \\
1 \\
0 \\
1 \\
1 \\
\hline
1 \\
\end{array} = 11 \]

\[ \begin{array}{c}
1 \\
1 \\
1 \\
1 \\
\hline
1 \\
\end{array} = 18 \]
Merging two Binomial Heaps

\[
\begin{align*}
3 & \quad 5 & \quad 15 \\
6 & \quad 8 & \quad 7 \\
10 & & \\
\end{align*}
\quad + 
\begin{align*}
1 & \quad 4 & \quad 9 & \quad 11 & \quad 13 & \quad 16 \\
4 & \quad 9 & \quad 12 & \quad 17 & \quad 13 & \quad 16 \\
& \quad 12 & \quad 17 & \quad 13 & \quad 16 & \quad 18 \\
& \quad 14 & \quad 18 & \quad 2 \\
\end{align*}
\]

\[
\begin{array}{cccccc}
0 & 0 & 1 & 1 & 1 & = 7 \\
0 & 1 & 0 & 1 & 1 & = 11 \\
1 & 1 & 1 & 1 & & \\
\hline
1 & 0 & 0 & 1 & 0 & = 18
\end{array}
\]
Binomial Heap vs. Fibonacci Heap: Structure

Binomial Heap:
- consists of binomial trees, and every order appears at most once
- immediately tidy up after INSERT or MERGE
Binomial Heap vs. Fibonacci Heap: Structure

Binomial Heap:
- consists of binomial trees, and every order appears at most once
- immediately tidy up after INSERT or MERGE

Fibonacci Heap:
- forest of MIN-HEAPs
- lazily defer tidying up; do it on-the-fly when search for the MIN
Structure of Fibonacci Heaps

- Forest of MIN-HEAPs

Fibonacci Heap

58

30

43

41

33

41

54

66

82

10

70

32

51
Structure of Fibonacci Heaps

- **Forest** of MIN-HEAPs
Structure of Fibonacci Heaps

- **Forest** of MIN-HEAPs
- **Nodes can be marked** *(roots are always unmarked)*

Fibonacci Heap

![Diagram of a Fibonacci Heap](image)

5.2: Fibonacci Heaps
Fibonacci Heaps

- Forest of MIN-HEAPs
- Nodes can be marked (roots are always unmarked)
Structure of Fibonacci Heaps

- **Forest** of MIN-HEAPs
- Nodes can be marked (*roots are always unmarked*)
- Tree roots are stored in a circular, doubly-linked list

---

**Fibonacci Heap**

- Forest of MIN-HEAPs
- Nodes can be marked (*roots are always unmarked*)
- Tree roots are stored in a circular, doubly-linked list

---

![Diagram of Fibonacci Heap]
Structure of Fibonacci Heaps

- **Forest** of MIN-HEAPs
- Nodes can be **marked** (roots are always unmarked)
- Tree roots are stored in a circular, doubly-linked list
Structure of Fibonacci Heaps

- **Forest** of MIN-HEAPs
- Nodes can be marked (*roots are always unmarked*)
- Tree roots are stored in a circular, doubly-linked list
- **Min-Pointer** pointing to the smallest element
Structure of Fibonacci Heaps

- **Forest** of MIN-HEAPs
- Nodes can be marked *(roots are always unmarked)*
- Tree roots are stored in a circular, doubly-linked list
- Min-Pointer pointing to the smallest element

Fibonacci Heap

![Fibonacci Heap Diagram]

How do we implement a Fibonacci Heap?

5.2: Fibonacci Heaps T.S. 9
Structure of Fibonacci Heaps

- **Forest** of MIN-HEAPs
- Nodes can be marked (roots are always unmarked)
- Tree roots are stored in a circular, doubly-linked list
- Min-Pointer pointing to the smallest element

How do we implement a Fibonacci Heap?
A single Node

Parent

Previous Sibling

Payload

marked

marked

Next Sibling

One of the Children

degree

degree
Magnifying a Four-Node Portion

5.2: Fibonacci Heaps
Magnifying a Four-Node Portion
Magnifying a Four-Node Portion

5.2: Fibonacci Heaps
Outline

Structure

Operations

Glimpse at the Analysis
Fibonacci Heap: INSERT

Insert

Create a singleton tree
Add to root list
and update min-pointer (if necessary)

Actual Costs: O(1)
Fibonacci Heap: INSERT

- Create a singleton tree
Fibonacci Heap: INSERT

- Create a singleton tree
- Add to root list

 Actual Costs: $O(1)$
Fibonacci Heap: INSERT

- Create a singleton tree
- Add to root list
**Fibonacci Heap: INSERT**

- **INSERT**
  - Create a singleton tree
  - Add to root list and update min-pointer (if necessary)

**Actual Costs:** $O(1)$
Fibonacci Heap: \textbf{INSERT}

- Create a singleton tree
- Add to root list and update min-pointer (if necessary)

Actual Costs: $O(1)$
**Fibonacci Heap: EXTRACT-MIN**

**EXTRACT-MIN**

- **Delete min**: Meld children into root list and unmark them
- **Consolidate**: So that no roots have the same degree
- **Update minimum**

---

**Actual Costs:**

\[ O(\text{trees}(H) + d(n)) \]

Every root becomes child of another root at most once!

\[ d(n) \] is the maximum degree of a root in any Fibonacci heap of size \( n \).
Fibonacci Heap: **EXTRACT-MIN**

- **EXTRACT-MIN**
  - Delete min

5.2: Fibonacci Heaps
Fibonacci Heap: EXTRACT-MIN

- **EXTRACT-MIN**

- Delete min ✓

Actual Costs:

\[ O(\text{trees}(H) + \text{d}(n)) \]

Every root becomes child of another root at most once!

d(n) is the maximum degree of a root in any Fibonacci heap of size n.
**Fibonacci Heap: EXTRACT-MIN**

- Delete min ✓
- Meld children into root list and unmark them

---

**Actual Costs:**

\[ O\left(\text{trees (H)} + d(n)\right) \]

Every root becomes child of another root at most once!

\(d(n)\) is the maximum degree of a root in any Fibonacci heap of size \(n\).

---

5.2: Fibonacci Heaps
Fibonacci Heap: \texttt{EXTRACT-MIN}

- Delete min ✓
- Meld children into root list and unmark them

\begin{itemize}
  \item Every root becomes child of another root at most once!
  \item \(d(n)\) is the maximum degree of a root in any Fibonacci heap of size \(n\)
\end{itemize}

Actual Costs: \(O(\text{trees}(H) + d(n))\)
**Fibonacci Heap: EXTRACT-MIN**

- **EXTRACT-MIN**
  - Delete min ✓
  - Meld children into root list and unmark them

### Actual Costs:

$$O\left(\text{trees}(H) + d(n)\right)$$

Every root becomes child of another root at most once!

$$d(n)$$ is the maximum degree of a root in any Fibonacci heap of size $$n$$. 

5.2: Fibonacci Heaps
Fibonacci Heap: EXTRACT-MIN

- Delete min ✓
- Meld children into root list and unmark them ✓
Fibonacci Heap: EXTRACT-MIN

- Delete min ✓
- Meld children into root list and unmark them ✓
- Consolidate so that no roots have the same degree
**Fibonacci Heap: EXTRACT-MIN**

- Delete min ✓
- Meld children into root list and unmark them ✓
- **Consolidate** so that no roots have the same degree (# children)

---

**Actual Costs:**

\[ O(\text{trees}(H) + d(n)) \]

Every root becomes child of another root at most once!

\( d(n) \) is the maximum degree of a root in any Fibonacci heap of size \( n \).
Fibonacci Heap: EXTRACT-MIN

- Delete min ✓
- Meld children into root list and unmark them ✓
- **Consolidate** so that no roots have the same degree (# children)

![Diagram of Fibonacci Heap]

**degree=2**
Fibonacci Heap: EXTRACT-MIN

- Delete min ✓
- Meld children into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)
Fibonacci Heap: EXTRACT-MIN

- Delete min ✓
- Meld children into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)

degree=0

Actual Costs: $O\left(\text{trees} + d(n)\right)$

Every root becomes child of another root at most once!

d(n) is the maximum degree of a root in any Fibonacci heap of size $n$. 

5.2: Fibonacci Heaps
Fibonacci Heap: EXTRACT-MIN

- Delete min ✓
- Meld children into root list and unmark them ✓
- **Consolidate** so that no roots have the same degree (# children)

Actual Costs:

\[ O\left(\text{trees}(H) + d(n)\right) \]

Every root becomes child of another root at most once!

\[ d(n) \] is the maximum degree of a root in any Fibonacci heap of size \( n \)
Fibonacci Heap: **EXTRACT-MIN**

- **Delete min ✓**
- **Meld children into root list and unmark them ✓**
- **Consolidate** so that no roots have the same degree (# children)

**Actual Costs:** $O(\text{trees}(H) + d(n))$

Every root becomes child of another root at most once!

$d(n)$ is the maximum degree of a root in any Fibonacci heap of size $n$. 

5.2: Fibonacci Heaps
Fibonacci Heap: \textsc{Extract-Min}

- **Delete min** ✓
- **Meld children into root list and unmark them** ✓
- **Consolidate** so that no roots have the same degree (# children)

\begin{center}
\begin{tabular}{c|c|c|c|c}
degree & 0 & 1 & 2 & 3 \\
\hline
\end{tabular}
\end{center}

\begin{center}
\begin{tikzpicture}
    \node[fill=black!20, circle] (7) at (0,0) {7};
    \node[fill=black!20, circle] (30) at (0,-2) {30};
    \node[fill=black!20, circle] (24) at (2,1) {24};
    \node[fill=black!20, circle] (26) at (1,1) {26};
    \node[fill=black!20, circle] (46) at (1,-2) {46};
    \node[fill=black!20, circle] (23) at (4,1) {23};
    \node[fill=black!20, circle] (17) at (4,-2) {17};
    \node[fill=black!20, circle] (18) at (6,0) {18};
    \node[fill=black!20, circle] (52) at (6,-2) {52};
    \node[fill=black!20, circle] (35) at (1.5,-2) {35};
    \node[fill=black!20, circle] (39) at (5.5,-2) {39};
    \node[fill=black!20, circle] (41) at (8,0) {41};
    \node[fill=black!20, circle] (44) at (8,-2) {44};
    \draw (7) -- (24);
    \draw (24) -- (26);
    \draw (26) -- (46);
    \draw (23) -- (17);
    \draw (17) -- (18);
    \draw (18) -- (52);
    \draw (26) -- (35);
    \draw (35) -- (24);
    \draw (39) -- (18);
    \draw (41) -- (52);
    \end{tikzpicture}
\end{center}
Fibonacci Heap: **EXTRACT-MIN**

- **Delete min ✓**
- **Meld children into root list and unmark them ✓**
- **Consolidate** so that no roots have the same degree (# children)

**Actual Costs:**

\[ O(\text{trees}(H) + d(n)) \]

Every root becomes child of another root at most once!

\( d(n) \) is the maximum degree of a root in any Fibonacci heap of size \( n \).
Fibonacci Heap: EXTRACT-MIN

- Delete min ✓
- Meld children into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)

```
<table>
<thead>
<tr>
<th>degree</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>
```

```
7
30
35
```

```
24
26
46
```

```
23
```

```
17
```

```
18
39
```

```
52
```

```
41
44
```

Actual Costs: $O(\text{trees (H)} + \text{d (n)})$

Every root becomes child of another root at most once!

$d (n)$ is the maximum degree of a root in any Fibonacci heap of size $n$. 

---

5.2: Fibonacci Heaps
**Fibonacci Heap: EXTRACT-MIN**

- Delete min ✓
- Meld children into root list and unmark them ✓
- **Consolidate** so that no roots have the same degree (# children)

---

**Actual Costs:**

$$O\left(\text{Trees} \left(\text{H}\right) + \text{d}\left(\text{n}\right)\right)$$

**Every root becomes child of another root at most once!**

\[d\left(\text{n}\right)\] is the maximum degree of a root in any Fibonacci heap of size \(\text{n}\).
Fibonacci Heap: **EXTRACT-MIN**

- **Delete min ✓**
- **Meld children into root list and unmark them ✓**
- **Consolidate** so that no roots have the same degree (# children)

**Actual Costs:** \( O(\text{trees}(H) + d(n)) \)

Every root becomes child of another root at most once!

\( d(n) \) is the maximum degree of a root in any Fibonacci heap of size \( n \).
Fibonacci Heap: \textsc{Extract-Min}

- Delete min ✓
- Meld children into root list and unmark them ✓
- Consolidate so that no roots have the same degree (\# children)

\begin{center}
\begin{tabular}{c|c|c|c|c}
degree & 0 & 1 & 2 & 3 \\
\hline
7 & 24 & 23 & 17 & 18 \\
30 & 26 & 46 & 39 & 52 \\
35 & & & & 41 \\
& & & & 44 \\
\end{tabular}
\end{center}

Actual Costs: $O(\text{trees}(H) + d(n))$

Every root becomes child of another root at most once!

$d(n)$ is the maximum degree of a root in any Fibonacci heap of size $n$. 
Fibonacci Heap: **EXTRACT-MIN**

- Delete min ✓
- Meld children into root list and unmark them ✓
- **Consolidate** so that no roots have the same degree (# children)

```
Actual Costs: O(\text{trees} (H) + d(n))
```

---

Every root becomes child of another root at most once!

\(d(n)\) is the maximum degree of a root in any Fibonacci heap of size \(n\)

---

5.2: Fibonacci Heaps
Fibonacci Heap: **EXTRACT-MIN**

- **EXTRACT-MIN**
  - Delete min ✓
  - Meld children into root list and unmark them ✓
  - **Consolidate** so that no roots have the same degree (# children)

---

**Actual Costs:**

\[
O\left(\sum_{h \in H} d(n)ight)
\]

Every root becomes child of another root at most once!
Fibonacci Heap: **EXTRACT-MIN**

- **EXTRACT-MIN**
  - Delete min ✓
  - Meld children into root list and unmark them ✓
  - Consolidate so that no roots have the same degree (# children)

---

### Degree Table

<table>
<thead>
<tr>
<th>Degree</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

### Actual Costs

\[ O(\text{trees}(H) + d(n)) \]

Every root becomes child of another root at most once!

\[ d(n) \]

is the maximum degree of a root in any Fibonacci heap of size \( n \).
Fibonacci Heap: **EXTRACT-MIN**

- **Delete min ✓**
- **Meld children into root list and unmark them ✓**
- **Consolidate** so that no roots have the same degree (# children)

**Actual Costs:**
\[ O(trees(H) + d(n)) \]

Every root becomes child of another root at most once!

\[ d(n) \]

is the maximum degree of a root in any Fibonacci heap of size \( n \).
**Fibonacci Heap: `EXTRACT-MIN`**

- **Delete min ✓**
- **Meld children into root list and unmark them ✓**
- **Consolidate** so that no roots have the same degree (# children)

![Diagram of a Fibonacci heap showing the `EXTRACT-MIN` operation with nodes and degrees.](image-url)
Fibonacci Heap: **EXTRACT-MIN**

- **EXTRACT-MIN**
  - Delete min ✓
  - Meld children into root list and unmark them ✓
  - **Consolidate** so that no roots have the same degree (# children)

---

**Actual Costs:**

\[
O\left(\text{trees}(H) + d(n)\right)
\]

*Every root becomes child of another root at most once!*  

\[d(n)\] is the maximum degree of a root in any Fibonacci heap of size \(n\).
Fibonacci Heap: **EXTRACT-MIN**

- Delete min ✓
- Meld children into root list and unmark them ✓
- **Consolidate** so that no roots have the same degree (# children)

---

**Actual Costs:**

\[ O(\text{trees}(H) + d(n)) \]

Every root becomes child of another root at most once!

\[ d(n) \] is the maximum degree of a root in any Fibonacci heap of size \( n \).
Fibonacci Heap: **EXTRACT-MIN**

- **Delete min ✓**
- **Meld children into root list and unmark them ✓**
- **Consolidate** so that no roots have the same degree (# children)

---

**Actual Costs:**

\[ O \left( \text{trees}(H) + d(n) \right) \]

Every root becomes child of another root at most once!

\[ d(n) \] is the maximum degree of a root in any Fibonacci heap of size \( n \).
Fibonacci Heap: EXTRACT-MIN

- Delete min ✓
- Meld children into root list and unmark them ✓
- **Consolidate** so that no roots have the same degree (# children)
Fibonacci Heap: EXTRACT-MIN

- Delete min ✓
- Meld children into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)

degree

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Actual Costs: \(O(\text{trees}(H) + d(n))\)

Every root becomes child of another root at most once!

\(d(n)\) is the maximum degree of a root in any Fibonacci heap of size \(n\)
Fibonacci Heap: **EXTRACT-MIN**

- **EXTRACT-MIN**
  - Delete min ✓
  - Meld children into root list and unmark them ✓
  - **Consolidate** so that no roots have the same degree (# children)

<table>
<thead>
<tr>
<th>degree</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

Actual Costs: $O\left(\frac{H}{d(n)}\right)$

Every root becomes child of another root at most once!

$d(n)$ is the maximum degree of a root in any Fibonacci heap of size $n$.
Fibonacci Heap: EXTRACT-MIN

- **Delete min ✓**
- **Meld children into root list and unmark them ✓**
- **Consolidate** so that no roots have the same degree (# children)

**Actual Costs:**

\[ O(\text{trees}(H) + d(n)) \]

Every root becomes child of another root at most once!

\[ d(n) \] is the maximum degree of a root in any Fibonacci heap of size \( n \)

5.2: Fibonacci Heaps
**Fibonacci Heap: EXTRACT-MIN**

- **EXTRACT-MIN**
  - Delete min ✓
  - Meld children into root list and unmark them ✓
  - Consolidate so that no roots have the same degree (# children)

---

**Actual Costs:**

\[ O(\left(\text{trees} (H) + d(n)) \right) \]

*Every root becomes child of another root at most once!*

\[ d(n) \] is the maximum degree of a root in any Fibonacci heap of size \( n \).
Fibonacci Heap: EXTRACT-MIN

- Delete min ✓
- Meld children into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children)

Actual Costs: $O \left( \sum \text{trees} \left( H \right) + \sum \text{degree} \left( n \right) \right)$

Every root becomes child of another root at most once!

$d \left( n \right)$ is the maximum degree of a root in any Fibonacci heap of size $n$. 

degree

0 1 2 3

5.2: Fibonacci Heaps
**Fibonacci Heap: EXTRACT-MIN**

- **EXTRACT-MIN**
  - Delete min ✓
  - Meld children into root list and unmark them ✓
  - **Consolidate** so that no roots have the same degree (# children)

```
degree
0 1 2 3
```

```
7
  17
  30
  24
   26
    23
    30
     35

18
  39
  52

41
  44
```

**Actual Costs:**
\[ O(\text{trees}(H) + d(n)) \]

- Every root becomes child of another root at most once!
- \( d(n) \) is the maximum degree of a root in any Fibonacci heap of size \( n \).
**Fibonacci Heap: EXTRACT-MIN**

- **EXTRACT-MIN**
  - Delete min ✓
  - Meld children into root list and unmark them ✓
  - **Consolidate** so that no roots have the same degree (# children)

**Actual Costs:**
\[ O(\text{trees}(H) + d(n)) \]

Every root becomes child of another root at most once!

\[ d(n) \] is the maximum degree of a root in any Fibonacci heap of size \( n \)

---

### Diagram

```
degree
0 1 2 3

7
17 30
24
23
30

18
52
39
41
44

26
46
35
```

5.2: Fibonacci Heaps
Fibonacci Heap: \textsc{Extract-Min}

- **Delete min ✓**
- **Meld children into root list and unmark them ✓**
- **Consolidate** so that no roots have the same degree (# children)

### Degree Table

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>39</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>52</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>41</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Actual Costs:

\[ O(\text{forrest}(H) + d(n)) \]

Every root becomes child of another root at most once!

\[ d(n) \] is the maximum degree of a root in any Fibonacci heap of size \( n \)
### Extract-Min

- Delete min ✓
- Meld children into root list and unmark them ✓
- **Consolidate** so that no roots have the same degree (number of children)

---

**Actual Costs:**

$$O\left(\sum_{i=1}^{H} d(n)\right)$$

**Every root becomes child of another root at most once!**

$$d(n)$$ is the maximum degree of a root in any Fibonacci heap of size $$n$$.
**Fibonacci Heap: EXTRACT-MIN**

- **EXTRACT-MIN**
  - Delete min ✓
  - Meld children into root list and unmark them ✓
  - **Consolidate** so that no roots have the same degree (# children)

---

**Actual Costs:**

\[O(\text{trees}(H)) + d(n))\]

Every root becomes child of another root at most once!

\[d(n)\] is the maximum degree of a root in any Fibonacci heap of size \(n\):

- **5.2: Fibonacci Heaps**
Fibonacci Heap: \textsc{Extract-Min}

- Delete min ✓
- Meld children into root list and unmark them ✓
- \textbf{Consolidate} so that no roots have the same degree (# children)

Actual Costs:

\[ O(\text{trees}(H) + d(n)) \]

\( d(n) \) is the maximum degree of a root in any Fibonacci heap of size \( n \)

Every root becomes child of another root at most once!
Fibonacci Heap: **EXTRACT-MIN**

- **Delete min ✓**
- **Meld children into root list and unmark them ✓**
- **Consolidate** so that no roots have the same degree (# children)

![Diagram of Fibonacci Heap](image)

---

**Actual Costs:**

$O\left(\sum_{H} d(n)\right)$

Every root becomes child of another root at most once!

$d(n)$ is the maximum degree of a root in any Fibonacci heap of size $n$. 

---

5.2: Fibonacci Heaps
**Fibonacci Heap: EXTRACT-MIN**

- **Delete min ✓**
- **Meld children into root list and unmark them ✓**
- **Consolidate** so that no roots have the same degree (# children) ✓

---

**Actual Costs:**

$O\left(\text{trees}(H) + d(n)\right)$

*Every root becomes child of another root at most once!*
Fibonacci Heap: EXTRACT-MIN

- **EXTRACT-MIN**
  - Delete min ✓
  - Meld children into root list and unmark them ✓
  - **Consolidate** so that no roots have the same degree (# children) ✓
  - Update minimum

---

**Actual Costs:**

\[ O(\text{trees}(H) + d(n)) \]

Every root becomes child of another root at most once!

\[ d(n) \]

is the maximum degree of a root in any Fibonacci heap of size \( n \)

---

5.2: Fibonacci Heaps
**Fibonacci Heap: EXTRACT-MIN**

- **EXTRACT-MIN**
  - Delete min ✓
  - Meld children into root list and unmark them ✓
  - **Consolidate** so that no roots have the same degree (# children) ✓
  - Update minimum ✓

---

**Actual Costs:**

\[ O\left(\text{trees}(H) + d(n)\right) \]

Every root becomes child of another root at most once!

\[ d(n) \]

is the maximum degree of a root in any Fibonacci heap of size \( n \)

---

5.2: Fibonacci Heaps
Fibonacci Heap: EXTRACT-MIN

- Delete min ✓
- Meld children into root list and unmark them ✓
- Consolidate so that no roots have the same degree (# children) ✓
- Update minimum ✓

Actual Costs:

```
min: 18
17 30 24 26 46 41 39 44 52
```

Actual Costs: \( O(\text{trees}(H) + d(n)) \)

Every root becomes child of another root at most once!

\( d(n) \) is the maximum degree of a root in any Fibonacci heap of size \( n \).
### Extract-Min

- Delete min ✓
- Meld children into root list and unmark them ✓
- **Consolidate** so that no roots have the same degree (# children) ✓
- Update minimum ✓

Every root becomes child of another root at most once!

**Actual Costs:**

```
O(trees(H) + d(n))
```

$d(n)$ is the maximum degree of a root in any Fibonacci heap of size $n$.

5.2: Fibonacci Heaps
**Fibonacci Heap: \textsc{Extract-Min}**

- Delete min ✓
- Meld children into root list and unmark them ✓
- **Consolidate** so that no roots have the same degree (# children) ✓
- Update minimum ✓

$d(n)$ is the maximum degree of a root in any Fibonacci heap of size $n$

**Actual Costs:** $\mathcal{O}(\text{trees}(H) + d(n))$

---

5.2: Fibonacci Heaps

T.S.
**DECREASE-KEY of node $x$**

- Decrease the key of $x$ (given by a pointer)

![Diagram of a Fibonacci heap with nodes and keys]

- **min**
  - 7
  - 24
  - 17
  - 23
  - 26
  - 46
  - 30
  - 35
  - 18
  - 21
  - 39
  - 41
  - 52
  - 38

**Peculiar Constraint:** Make sure that each non-root node loses at most one child before becoming root.
Fibonacci Heap: DECREASE-KEY (First Attempt)

**DECREASE-KEY of node x**

- Decrease the key of x (given by a pointer)

Diagram:

```
min

7

17  23

26  46  30

35

18

21  39

52

38

41
```

Peculiar Constraint: Make sure that each non-root node loses at most one child before becoming root.
DECREASE-KEY of node \( x \):

- Decrease the key of \( x \) (given by a pointer)
Fibonacci Heap: \textsc{Decrease-Key} (First Attempt)

\textbf{DECREASE-K\textsc{EY} of node $x$}

- Decrease the key of $x$ (given by a pointer)
- Check if heap-order is violated

\begin{itemize}
  \item [min]
  \begin{itemize}
    \item 7
    \item 20
    \item 26
    \item 35
    \item 17
    \item 23
    \item 46
    \item 30
    \item 18
    \item 21
    \item 39
    \item 41
    \item 38
    \item 52
  \end{itemize}
\end{itemize}
**Fibonacci Heap: DECREASE-KEY (First Attempt)**

**DECREASE-KEY of node x**

- Decrease the key of x (given by a pointer)
- Check if heap-order is violated

![Diagram of a Fibonacci heap with keys: 7, 18, 38, 20, 17, 23, 21, 39, 41, 26, 46, 30, 35, 26, 46, 30, 52, 35. The minimum key is 5.2.](image)
**Fibonacci Heap: DECREASE-KEY (First Attempt)**

**DECREASE-KEY of node** \( x \)

- Decrease the key of \( x \) (given by a pointer)
- Check if heap-order is violated
  - If not

---

Wide and shallow tree

Degree = 3,
Nodes = 4

Peculiar Constraint: Make sure that each non-root node loses at most one child before becoming root.
**DECREASE-KEY of node $x$**

- Decrease the key of $x$ (given by a pointer)
- Check if heap-order is violated
  - If not, then done.

![Fibonacci Heap Diagram](image-url)
Fibonacci Heap: \textsc{Decrease-Key} (First Attempt)

**\textsc{Decrease-Key} of node } x**

- Decrease the key of } x \text{ (given by a pointer)}
- Check if heap-order is violated
  - If not, then done.
  - Otherwise,

```
min
```

```
7
\downarrow
--
20 17 23
\downarrow
26 46 30
\downarrow
35
```

```
18
\downarrow
21 39
\downarrow
52
```

```
38
```

```
Wide and shallow tree
```

Degree = 3,
Nodes = 4

Peculiar Constraint: Make sure that each non-root node loses at most one child before becoming root
Fibonacci Heap: **DECREASE-KEY** (First Attempt)

**DECREASE-KEY** of node $x$

- Decrease the key of $x$ (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise,

![Diagram of Fibonacci Heap]

- **min**
- **7**
- **20**
- **26**
- **35**
- **17**
- **30**
- **23**
- **41**
- **18**
- **21**
- **52**
- **38**
- **39**

Wide and shallow tree

Degree = 3, Nodes = 4

Peculiar Constraint: Make sure that each non-root node loses at most one child before becoming root.
**Fibonacci Heap: DECREASE-KEY (First Attempt)**

**DECREASE-KEY of node x**

- Decrease the key of x (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise,

---

**min**

```
7
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>26</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>35</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>17</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>23</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>18</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>21</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>52</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>39</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>38</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>41</td>
</tr>
</tbody>
</table>
```

**Peculiar Constraint**: Make sure that each non-root node loses at most one child before becoming root.

---

5.2: Fibonacci Heaps
**Fibonacci Heap: DECREASE-KEY (First Attempt)**

**DECREASE-KEY of node x**

- Decrease the key of x (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise,

```
min

7  18  38
20  17  23  21  39  41
26  15  30
35
```

Wide and shallow tree

Degree = 3, Nodes = 4

Peculiar Constraint: Make sure that each non-root node loses at most one child before becoming root.
**Fibonacci Heap: DECREASE-KEY (First Attempt)**

**DECREASE-KEY of node x**
- Decrease the key of x (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise,

**Diagram:**
- The diagram shows a Fibonacci heap with a rooted tree structure.
- The root node is labeled with the value 7.
- Other nodes include 20, 17, 23, 18, 38, 41, 26, 15, 30, 52, 35.
- The min value is 5.
- The degree of the tree is 3, and the number of nodes is 4.
- A peculiar constraint ensures each non-root node loses at most one child before becoming root.
**Fibonacci Heap: DECREASE-KEY (First Attempt)**

**DECREASE-KEY of node x**

- Decrease the key of $x$ (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at $x$ and meld into root list (update min).

![Diagram of Fibonacci Heap with Decrease-Key operation]

- **min**
- Nodes:
  - 7, 20, 26, 15, 30, 23, 17, 21, 39, 41, 38, 52, 35, 18
Fibonacci Heap: DECREASE-KEY (First Attempt)

**DECREASE-KEY of node x**

- Decrease the key of \( x \) (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at \( x \) and meld into root list (update min).

![Diagram of Fibonacci Heap](image)
**Fibonacci Heap: DECREASE-KEY (First Attempt)**

**DECREASE-KEY of node x**

- Decrease the key of x (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at x and meld into root list (update min).

```
min
```

```
    7
   /|
  20 17 23
   |
  26
```

```
    18
   /|
  21 39 52
   |
  41
```

```
    38
   /|
   15
```

Wide and shallow tree

Degree = 3,
Nodes = 4

Peculiar Constraint: Make sure that each non-root node loses at most one child before becoming root
**Fibonacci Heap: DECREASE-KEY (First Attempt)**

**DECREASE-KEY of node x**

- Decrease the key of x (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at x and meld into root list (update min).
Decrease-Key of node $x$

- Decrease the key of $x$ (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at $x$ and meld into root list (update min).
Fibonacci Heap: **DECREASE-KEY** (First Attempt)

**DECREASE-KEY** of node \( x \)

- Decrease the key of \( x \) (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at \( x \) and meld into root list (update min).

![Diagram of Fibonacci Heap]

- **min**
- **7**
- **20**
- **26**
- **5**
- **17**
- **30**
- **23**
- **52**
- **18**
- **21**
- **39**
- **38**
- **41**
- **15**

Degree = 3, Nodes = 4

Peculiar Constraint: Make sure that each non-root node loses at most one child before becoming root.
Fibonacci Heap: DECREASE-KEY (First Attempt)

**DECREASE-KEY** of node $x$

- Decrease the key of $x$ (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at $x$ and meld into root list (update min).

```
min

7
  20
    26
      5
  17
  23

18
  21
    39
      30
  41
  38
    15

min

Wide and shallow tree
Degree = 3,
Nodes = 4
Peculiar Constraint: Make sure that each non-root node loses at most one child before becoming root
```
Fibonacci Heap: **DECREASE-KEY** (First Attempt)

**DECREASE-KEY of node** \( x \)

- Decrease the key of \( x \) (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at \( x \) and meld into root list (update min).
**Decrease-Key** of node $x$

- Decrease the key of $x$ (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at $x$ and meld into root list (update min).
**Fibonacci Heap: DECREASE-KEY (First Attempt)**

**DECREASE-KEY of node x**

- Decrease the key of x (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at x and meld into root list (update min).

![Diagram of Fibonacci Heap]

- **min**
- **7** -> **20**, **17**, **23**
- **18** -> **21**, **39**
- **38**
- **15**
- **5**
**Fibonacci Heap: DECREASE-KEY (First Attempt)**

**DECREASE-KEY of node x**

- Decrease the key of x (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at x and meld into root list (update min).

![Diagram of Fibonacci Heap with node x highlighted]

- **Nodes**: 4
- **Degree**: 3
- **Wide and shallow tree**

```plaintext
min

7 -- 20 -- 19
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
</table>
17   17
23   23

18 -- 21
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
</table>
39   39
41   41

38 -- 15 -- 5
```
Fibonacci Heap: DECREASE-KEY (First Attempt)

**DECREASE-KEY of node x**

- Decrease the key of x (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at x and meld into root list (update min).
Fibonacci Heap: **DECREASE-KEY** (First Attempt)

**DECREASE-KEY** of node \( x \):

- Decrease the key of \( x \) (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at \( x \) and meld into root list (update min).
**Fibonacci Heap: DECREASE-KEY (First Attempt)**

**DECREASE-KEY of node x**

- Decrease the key of x (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at x and meld into root list (update min).

![Diagram of Fibonacci Heap](image-url)
**DECREASE-KEY** of node $x$

- Decrease the key of $x$ (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at $x$ and meld into root list (update min).
**DECREASE-KEY** of node $x$

- Decrease the key of $x$ (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at $x$ and meld into root list (update min).
**DECREASE-KEY of node x**

- Decrease the key of x (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at x and meld into root list (update min).
Fibonacci Heap: DECREASE-KEY (First Attempt)

- Decrease the key of \( x \) (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at \( x \) and meld into root list (update min).
Fibonacci Heap: **DECREASE-KEY** (First Attempt)

**DECREASE-KEY** of node $x$

- Decrease the key of $x$ (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at $x$ and meld into root list (update min).
**DECREASE-KEY of node x**

- Decrease the key of x (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at x and meld into root list (update min).
**Fibonacci Heap: DECREASE-KEY (First Attempt)**

**DECREASE-KEY of node x**

- Decrease the key of x (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at x and meld into root list (update min).

---

Wide and shallow tree

\[
\text{Degree} = 3, \quad \text{Nodes} = 4
\]
**Fibonacci Heap: DECREASE-KEY (First Attempt)**

**DECREASE-KEY of node x**

- Decrease the key of $x$ (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at $x$ and meld into root list (update min).

---

**Diagram:**

- **Wide and shallow tree**

**Min heap:**

- Min node: 5
- Nodes: 7, 18, 20, 17, 23, 15, 38, 39, 41, 19, 12

---

5.2: Fibonacci Heaps T.S. 15
Fibonacci Heap: **DECREASE-KEY** (First Attempt)

**DECREASE-KEY of node x**

- Decrease the key of x (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at x and meld into root list (update min).

Wide and shallow tree

Degree = 3, Nodes = 4

```
7
  20
  17
  23

18
  21
  39
  38

41
  15
  19

52

min
```

5.2: Fibonacci Heaps
DECREASE-KEY of node $x$

- Decrease the key of $x$ (given by a pointer)
- Check if heap-order is violated
  - If not, then done.
  - Otherwise, cut tree rooted at $x$ and meld into root list (update min).

**Peculiar Constraint:** Make sure that each non-root node loses at most one child before becoming root.

Wide and shallow tree
Fibonacci Heap: **DECREASE-KEY**

**DECREASE-KEY** of node $x$

- Decrease the key of $x$ (given by a pointer)

```
Actual Cost: $O(\# \text{ cuts})$
```

```
1. DECREASE-KEY 46 $\Rightarrow$ 15
2. DECREASE-KEY 35 $\Rightarrow$ 5
5.2: Fibonacci Heaps
```
Fibonacci Heap: **DECREASE-KEY**

**DECREASE-KEY** of node $x$:

- Decrease the key of $x$ (given by a pointer)
- (Here we consider only cases where heap-order is violated)

**Diagram**:

![Fibonacci Heap Diagram](image)

**Actual Cost**: $O(\# \text{ cuts})$

1. **DECREASE-KEY** $46 \Rightarrow 15$
Fibonacci Heap: **DECREASE-KEY**

**DECREASE-KEY** of node $x$

- Decrease the key of $x$ (given by a pointer)
- (Here we consider only cases where heap-order is violated)
  
  $\Rightarrow$ Cut tree rooted at $x$, unmark $x$, meld into root list

---

**Example**

Min:

1. **DECREASE-KEY** 46 $\leadsto$ 15

Diagram:

```
min

7

24
26
35

17
30
15

23

18
21
52

39

38

41
```
Fibonacci Heap: DECREASE-KEY

**DECREASE-KEY of node x**

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)
  ⇒ Cut tree rooted at x, unmark x, meld into root list

1. **DECREASE-KEY 46 ⇝ 15**
**Fibonacci Heap: DECREASE-KEY**

**DECREASE-KEY of node x**

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

⇒ Cut tree rooted at x, unmark x, meld into root list

---

1. **DECREASE-KEY 46 ⇝ 15**

---

![Diagram of Fibonacci Heap](image)
Fibonacci Heap: **DECREASE-KEY**

- **DECREASE-KEY of node** $x$
  - Decrease the key of $x$ (given by a pointer)
  - (Here we consider only cases where heap-order is violated)
  - Cut tree rooted at $x$, unmark $x$, meld into root list

![Diagram of a Fibonacci heap with nodes labeled 7, 24, 26, 17, 30, 23, 18, 21, 35, 38, 39, 41, and 52. The node with key 15 is highlighted, and the nodes with keys 46 and 15 in a separate box.]

1. **DECREASE-KEY 46 \(\leadsto 15\)**
Fibonacci Heap: **DECREASE-KEY**

**DECREASE-KEY of node** $x$

- Decrease the key of $x$ (given by a pointer)
- (Here we consider only cases where heap-order is violated)

⇒ Cut tree rooted at $x$, unmark $x$, meld into root list and:

1. **DECREASE-KEY** $46 \Rightarrow 15$

---

Actual Cost: $O(\# \text{ cuts})$
**Fibonacci Heap: DECREASE-KEY**

**DECREASE-KEY** of node $x$

- Decrease the key of $x$ (given by a pointer)
- (Here we consider only cases where heap-order is violated)

$\Rightarrow$ Cut tree rooted at $x$, unmark $x$, meld into root list and:
- Check if parent node is marked

```
1. DECREASE-KEY 46 $\Rightarrow$ 15
```
**Fibonacci Heap: DECREASE-KEY**

- **DECREASE-KEY of node x**
  - Decrease the key of x (given by a pointer)
  - (Here we consider only cases where heap-order is violated)
  - Cut tree rooted at x, unmark x, meld into root list and:
    - Check if parent node is marked
      - If unmarked, mark it (unless it is a root)

```
1. DECREASE-KEY 46 ⇝ 15
```
Fibonacci Heap: **DECREASE-KEY**

**DECREASE-KEY of node x**

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

⇒ Cut tree rooted at x, unmark x, meld into root list and:
- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)

---

1. **DECREASE-KEY** 46 ↦ 15
**Fibonacci Heap: **DECREASE-KEY**

- Decrease the key of \( x \) (given by a pointer)
- (Here we consider only cases where heap-order is violated)

\( \Rightarrow \) Cut tree rooted at \( x \), unmark \( x \), meld into root list **and:**
- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)

---

**Diagram:**

```
min

7
24 17 23
26 30
35

18
21 39
41
52

38
15
```

1. **DECREASE-KEY 46 \(\rightarrow\) 15 \(\checkmark\)
Fibonacci Heap: **DECREASE-KEY**

**DECREASE-KEY of node x**

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

⇒ Cut tree rooted at x, unmark x, meld into root list *and*:
  - Check if parent node is marked
    - If unmarked, mark it (unless it is a root)

![Diagram of Fibonacci Heap]

**Actual Cost:** \(O(\#\text{ cuts})\)

1. **DECREASE-KEY 46 \(\leadsto 15 \checkmark\)**
2. **DECREASE-KEY 35 \(\leadsto 5\)**
**Fibonacci Heap: DECREASE-KEY**

**DECREASE-KEY of node x**

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

⇒ Cut tree rooted at x, unmark x, meld into root list and:
  - Check if parent node is marked
    - If unmarked, mark it (unless it is a root)

---

**Diagram:**

```
min

7

24 17 23

26 30

5

18

15

38

41

21 39

52

1. DECREASE-KEY 46 ⇝ 15 ✓
2. DECREASE-KEY 35 ⇝ 5
```

---

5.2: Fibonacci Heaps

T.S. 16
Fibonacci Heap: DECREASE-KEY

**DECREASE-KEY of node x**

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)
- Cut tree rooted at x, unmark x, meld into root list and:
  - Check if parent node is marked
    - If unmarked, mark it (unless it is a root)

---

```
min
```

```
  7
  /  
 24  17  23
/  \
26  30
/  \
5  
```

```
  18
  /  
 21  39
/  \
52
```

```
  38  15
```

```
 1. DECREASE-KEY 46 \(\sim\) 15 ✓
 2. DECREASE-KEY 35 \(\sim\) 5
```
Fibonacci Heap: \textsc{Decrease-Key}

\textbf{Decrease-Key of node } \textit{x} \\
- Decrease the key of \textit{x} (given by a pointer)  
- (Here we consider only cases where heap-order is violated)  
  \Rightarrow \text{Cut tree rooted at } \textit{x}, \text{ unmark } \textit{x}, \text{ meld into root list and:}  
  - Check if parent node is marked  
    - If unmarked, mark it (unless it is a root)

1. \textsc{Decrease-Key} 46 $\leadsto$ 15 $\checkmark$  
2. \textsc{Decrease-Key} 35 $\leadsto$ 5
Fibonacci Heap: **DECREASE-KEY**

**DECREASE-KEY of node x**

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

⇒ Cut tree rooted at x, unmark x, meld into root list **and**:
- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)

---

**Actual Cost:** $O(\# \text{ cuts})$

1. **DECREASE-KEY** 46 $\leadsto$ 15 ✓

2. **DECREASE-KEY** 35 $\leadsto$ 5

---

5.2: Fibonacci Heaps
**Fibonacci Heap: DECREASE-KEY**

**DECREASE-KEY** of node $x$

- Decrease the key of $x$ (given by a pointer)
- (Here we consider only cases where heap-order is violated)

⇒ Cut tree rooted at $x$, unmark $x$, meld into root list and:
  - Check if parent node is marked
    - If unmarked, mark it (unless it is a root)

---

**Example**

![Diagram of Fibonacci Heap]

- **min**
  - $7$ (root)
  - $24$, $17$, $23$
  - $26$, $30$
  - $5$

- **Actual Cost:** $O(\# \text{c}ut)$

- **1. DECREASE-KEY** 46 $\rightarrow$ 15 $✓$
- **2. DECREASE-KEY** 35 $\rightarrow$ 5
Fibonacci Heap: DECREASE-KEY

**DECREASE-KEY of node x**

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)
  
  ⇒ Cut tree rooted at x, unmark x, meld into root list and:
  - Check if parent node is marked
    - If unmarked, mark it (unless it is a root)
    - If marked,

---

**Graph: Fibonacci Heap**

1. **DECREASE-KEY 46 ➞ 15 ✓**
2. **DECREASE-KEY 35 ➞ 5**
**Fibonacci Heap: DECREASE-KEY**

**DECREASE-KEY of node x**
- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

⇒ Cut tree rooted at x, unmark x, meld into root list and:
- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)
  - If marked, unmark and meld it into root list and recurse (Cascading Cut)

---

1. DECREASE-KEY 46 \(\rightsquigarrow\) 15 \(\checkmark\)
2. DECREASE-KEY 35 \(\rightsquigarrow\) 5

---

5.2: Fibonacci Heaps
**Fibonacci Heap: DECREASE-KEY**

- Decrease the key of $x$ (given by a pointer)
- (Here we consider only cases where heap-order is violated)

⇒ Cut tree rooted at $x$, unmark $x$, meld into root list and:
  - Check if parent node is marked
    - If unmarked, mark it (unless it is a root)
    - If marked, unmark and meld it into root list and recurse (Cascading Cut)

1. DECREASE-KEY 46 $\rightarrow$ 15 ✓
2. DECREASE-KEY 35 $\rightarrow$ 5
Fibonacci Heap: **DECREASE-KEY**

**DECREASE-KEY of node x**

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

⇒ Cut tree rooted at x, unmark x, meld into root list and:
- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)
  - If marked, unmark and meld it into root list and recurse (**Cascading Cut**)

---

**Actual Cost:** \(O(\# \text{ cuts})\)

1. **DECREASE-KEY 46 ⇝ 15 ✓
2. **DECREASE-KEY 35 ⇝ 5**
**Fibonacci Heap: DECREASE-KEY**

**DECREASE-KEY of node x**

- Decrease the key of $x$ (given by a pointer)
- (Here we consider only cases where heap-order is violated)

⇒ Cut tree rooted at $x$, unmark $x$, meld into root list and:
- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)
  - If marked, unmark and meld it into root list and recurse (Cascading Cut)

---

1. **DECREASE-KEY 46 $\rightsquigarrow$ 15 ✓
2. **DECREASE-KEY 35 $\rightsquigarrow$ 5
**Fibonacci Heap: DECREASE-KEY**

**DECREASE-KEY of node x**

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

⇒ Cut tree rooted at x, unmark x, meld into root list and:
- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)
  - If marked, unmark and meld it into root list and recurse (Cascading Cut)

---

![Diagram of Fibonacci Heap with key values and decrease-key operations]

1. **DECREASE-KEY 46 ↦ 15 ✓**
2. **DECREASE-KEY 35 ↦ 5**

**5.2: Fibonacci Heaps**
**Fibonacci Heap: DECREASE-KEY**

**DECREASE-KEY of node x**

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

⇒ Cut tree rooted at x, unmark x, meld into root list and:
- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)
  - If marked, unmark and meld it into root list and recurse (Cascading Cut)

---

### Example

1. **DECREASE-KEY 46 ⇝ 15 ✓**
2. **DECREASE-KEY 35 ⇝ 5**
Fibonacci Heap: **DECREASE-KEY**

**DECREASE-KEY of node x**

- Decrease the key of \( x \) (given by a pointer)
- (Here we consider only cases where heap-order is violated)

⇒ Cut tree rooted at \( x \), unmark \( x \), meld into root list and:
  - Check if parent node is marked
    - If unmarked, mark it (unless it is a root)
    - If marked, unmark and meld it into root list and recurse (Cascading Cut)

---

1. **DECREASE-KEY 46 \( \leadsto 15 \) ✓**
2. **DECREASE-KEY 35 \( \leadsto 5 \)**

---

min

17 23

30

21 39

52

41

5 26 24

Actual Cost: \( O(\# \text{ cuts}) \)
**Fibonacci Heap: DECREASE-KEY**

**DECREASE-KEY** of node $x$

- Decrease the key of $x$ (given by a pointer)
- (Here we consider only cases where heap-order is violated)

⇒ Cut tree rooted at $x$, unmark $x$, meld into root list and:
- Check if parent node is marked
  - If unmarked, mark it (unless it is a root)
  - If marked, unmark and meld it into root list and recurse (Cascading Cut)

---

**Diagram:**

- $7$ connected to $17$ and $23$, $17$ connected to $30$, $30$ connected to a dashed line.
- $18$ connected to $21$ and $39$, $21$ connected to $52$, $52$ connected to a dashed line.
- $38$ connected to $41$ and $39$.
- $15$ connected to a dashed line.
- $5$ is the root.
- $26$ and $24$ are connected to a dashed line.

**Actual Cost:** $O(\# \text{ cuts})$

1. **DECREASE-KEY** $46 \rightarrow 15 \checkmark$
2. **DECREASE-KEY** $35 \rightarrow 5 \checkmark$
Fibonacci Heap: **DECREASE-KEY**

**DECREASE-KEY** of node \( x \)

- Decrease the key of \( x \) (given by a pointer)
- (Here we consider only cases where heap-order is violated)

⇒ Cut tree rooted at \( x \), unmark \( x \), meld into root list and:
  - Check if parent node is marked
    - If unmarked, mark it (unless it is a root)
    - If marked, unmark and meld it into root list and recurse (Cascading Cut)

---

Actual Cost:

1. **DECREASE-KEY** 46 ➝ 15 ✓
2. **DECREASE-KEY** 35 ➝ 5 ✓
**Fibonacci Heap: DECREASE-KEY**

**DECREASE-KEY of node x**

- Decrease the key of x (given by a pointer)
- (Here we consider only cases where heap-order is violated)

⇒ Cut tree rooted at x, unmark x, meld into root list and:
  - Check if parent node is marked
    - If unmarked, mark it (unless it is a root)
    - If marked, unmark and meld it into root list and recurse (Cascading Cut)

**Actual Cost: \( \mathcal{O}(\# \text{ cuts}) \)**

1. DECREASE-KEY 46 \( \rightsquigarrow 15 \) ✓
2. DECREASE-KEY 35 \( \rightsquigarrow 5 \) ✓
Outline

Structure

Operations

Glimpse at the Analysis
Amortized Analysis via Potential Method

- **INSERT**: actual $O(1)$
- **EXTRACT-MIN**: actual $O(\text{trees}(H) + d(n))$
- **DECREASE-KEY**: actual $O(\# \text{ cuts}) \leq O(\text{marks}(H))$
Amortized Analysis via Potential Method

- **INSERT:** actual $O(1)$
- **EXTRACT-MIN:** actual $O(\text{trees}(H) + d(n))$
- **DECREASE-KEY:** actual $O(\# \text{ cuts}) \leq O(\text{marks}(H))$

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$
Amortized Analysis via Potential Method

- **INSERT**: actual $O(1)$
- **EXTRACT-MIN**: actual $O(\text{trees}(H) + d(n))$
- **DECREASE-KEY**: actual $O(\# \text{cuts}) \leq O(\text{marks}(H))$

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$
Amortized Analysis via Potential Method

- **INSERT**: actual $O(1)$
- **EXTRACT-MIN**: actual $O(\text{trees}(H) + d(n))$
- **DECREASE-KEY**: actual $O(\# \text{cuts}) \leq O(\text{marks}(H))$

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$
Amortized Analysis via Potential Method

- **INSERT:** actual $O(1)$
- **EXTRACT-MIN:** actual $O(\text{trees}(H) + d(n))$
- **DECREASE-KEY:** actual $O(\# \text{ cuts}) \leq O(\text{marks}(H))$

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$
Amortized Analysis via Potential Method

- **INSERT**: actual $O(1)$  
  amortized $O(1)$
- **EXTRACT-MIN**: actual $O(\text{trees}(H) + d(n))$  
  amortized $O(d(n))$
- **DECREASE-KEY**: actual $O(\# \text{ cuts}) \leq O(\text{marks}(H))$  
  amortized $O(1)$

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$