Fibonacci Heaps
Outline for Today

- **Recap from Last Time**
  - Quick refresher on binomial heaps and lazy binomial heaps.

- **The Need for decrease-key**
  - An important operation in many graph algorithms.

- **Fibonacci Heaps**
  - A data structure efficiently supporting *decrease-key*.

- **Representational Issues**
  - Some of the challenges in Fibonacci heaps.
Recap from Last Time
(Lazy) Binomial Heaps

- Last time, we covered the binomial heap and a variant called the lazy binomial heap.
- These are priority queue structures designed to support efficient melding.
- Elements are stored in a collection of binomial trees.
Draw what happens if we *enqueue* the numbers 1, 2, 3, 4, 5, 6, 7, 8, and 9 into each heap.
Draw what happens after performing an \textit{extract-min} in each binomial heap.
Let’s **enqueue** 10, 11, and 12 into both heaps.
Eager Binomial Heap

Lazy Binomial Heap

Draw what happens after we do a `extract-min` from both heaps.
Operation Costs

- Eager Binomial Heap:
  - **enqueue**: $O(\log n)$
  - **meld**: $O(\log n)$
  - **find-min**: $O(\log n)$
  - **extract-min**: $O(\log n)$

- Lazy Binomial Heap:
  - **enqueue**: $O(1)$
  - **meld**: $O(1)$
  - **find-min**: $O(1)$
  - **extract-min**: $O(\log n)^*$
  - *amortized*

**Intuition:** Each **extract-min** has to do a bunch of cleanup for the earlier **enqueue** operations, but then leaves us with few trees.
New Stuff!
The Need for decrease-key
The **decrease-key** Operation

- Some priority queues support the operation **decrease-key**(v, k), which works as follows:

  *Given a pointer to an element v, lower its key (priority) to k. It is assumed that k is less than the current priority of v.*

- This operation is crucial in efficient implementations of Dijkstra's algorithm and Prim's MST algorithm.
Dijkstra and *decrease-key*

• Dijkstra's algorithm can be implemented with a priority queue using
  • O(n) total *enqueue*es,
  • O(n) total *extract-min*es, and
  • O(m) total *decrease-key*es.
Dijkstra and *decrease-key*

- Dijkstra's algorithm can be implemented with a priority queue using
  - $O(n)$ total *enqueue*s,
  - $O(n)$ total *extract-min*s, and
  - $O(m)$ total *decrease-key*s.
- Dijkstra's algorithm runtime is
  \[
  O(n \, T_{\text{enq}} + n \, T_{\text{ext}} + m \, T_{\text{dec}})
  \]
Prim and *decrease-key*

- Prim's algorithm can be implemented with a priority queue using
  - O(n) total *enqueue*es,
  - O(n) total *extract-min*ss, and
  - O(m) total *decrease-key*ss.

![Graph](image)
Prim and *decrease-key*

- Prim's algorithm can be implemented with a priority queue using
  - $O(n)$ total *enqueue*s,
  - $O(n)$ total *extract-min*s, and
  - $O(m)$ total *decrease-key*s.
- Prim's algorithm runtime is
  \[ O(n T_{\text{enq}} + n T_{\text{ext}} + m T_{\text{dec}}) \]
Standard Approaches

• In a binary heap, enqueue, extract-min, and decrease-key can be made to work in time $O(\log n)$ time each.

• Cost of Dijkstra's / Prim's algorithm:

\[
O(n T_{enq} + n T_{ext} + m T_{dec})
\]

\[
= O(n \log n + n \log n + m \log n)
\]

\[
= \mathcal{O}(m \log n)
\]
Standard Approaches

• In a lazy binomial heap, *enqueue* takes amortized time $O(1)$, and *extract-min* and *decrease-key* take amortized time $O(\log n)$.

• Cost of Dijkstra's / Prim's algorithm:
  
  \[
  O(n T_{\text{enq}} + n T_{\text{ext}} + m T_{\text{dec}}) \\
  = O(n + n \log n + m \log n) \\
  = O(m \log n)
  \]
Where We're Going

- The *Fibonacci heap* has these amortized runtimes:
  - *enqueue*: $O(1)$
  - *extract-min*: $O(\log n)$.
  - *decrease-key*: $O(1)$.
- Cost of Prim's or Dijkstra's algorithm:
  \[
  O(n \, T_{\text{enq}} + n \, T_{\text{ext}} + m \, T_{\text{dec}}) = O(n + n \log n + m) = O(m + n \log n)
  \]
- This is theoretically optimal for a comparison-based priority queue in Dijkstra's or Prim's algorithms.
The Challenge of \textit{decrease-key}
How might we implement *decrease-key* in a lazy binomial heap?
How might we implement \textit{decrease-key} in a lazy binomial heap?
How might we implement \textit{decrease-key} in a lazy binomial heap?

If our lazy binomial heap has \( n \) nodes, how tall can the tallest tree be?

Suppose the biggest tree has \( 2^k \) nodes in it.

Then \( 2^k \leq n \).

So \( k = O(\log n) \).

If our lazy binomial heap has \( n \) nodes, how tall can the tallest tree be?
Challenge: Support \textit{decrease-key} in (amortized) time O(1).

If our lazy binomial heap has \( n \) nodes, how tall can the tallest tree be?

Suppose the biggest tree has \( 2^k \) nodes in it.
Then \( 2^k \leq n \).
So \( k = O(\log n) \).

**Challenge:** Support *decrease-key* in (amortized) time $O(1)$.

We cannot have all three of these nice properties at once:

1. *decrease-key* takes time $O(1)$.
2. Our trees are heap-ordered.
3. Our trees are binomial trees.
Challenge: Support *decrease-key* in (amortized) time $O(1)$. 

We cannot have all three of these nice properties at once:

1. *decrease-key* takes time $O(1)$.
2. Our trees are heap-ordered.
3. Our trees are binomial trees.
**Challenge:** Support \textit{decrease-key} in (amortized) time $O(1)$. 
Problem: What do we do in an *extract-min*?
**Problem:** What do we do in an *extract-min*?

---

*What We Used to Do*

<table>
<thead>
<tr>
<th>Order 2</th>
<th>Order 1</th>
<th>Order 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>12</td>
</tr>
</tbody>
</table>

---

> What We Used to Do
Problem: What do we do in an extract-min?

What We Used to Do

This system assumes we can assign an “order” to each tree.
That’s easy with binomial trees.
That’s harder with our new trees.
What should we do here?
Problem: What do we do in an extract-min?

What We Used to Do

Idea 1: A tree has order $k$ if it has $2^k$ nodes.

Idea 2: A tree has order $k$ if its root has $k$ children.
Problem: What do we do in an extract-min?

Idea 1: A tree has order $k$ if it has $2^k$ nodes.

Idea 2: A tree has order $k$ if its root has $k$ children.
**Problem:** What do we do in an *extract-min*?
(1) To do a **decrease-key**, cut the node from its parent.

(2) Do **extract-min** as usual, using child count as order.

**Question:** How efficient is this?
Claim: Our trees can end up with very unusual shapes.

1. enqueue $2^k + 1$ nodes.
2. Do an extract-min.
3. Use decrease-key and extract-min to prune the tree.
**Claim:** Because tree shapes aren’t well-constrained, we can force `extract-min` to take amortized time $\Omega(n^{1/2})$.

**Intuition:** `extract-min` is only fast if it compacts nodes into a few trees.

There are $\Theta(n^{1/2})$ trees here. What happens if we repeatedly `enqueue` and `extract-min` a small value?

Each operation does $\Theta(n^{1/2})$ work, and doesn’t make any future operations any better.
With $n$ nodes, it’s possible to have $\Omega(n^{1/2})$ trees of distinct orders.

**Question:** Why didn’t this happen before?
Binomial tree sizes grow exponentially. With $n$ nodes, we can have at most $O(\log n)$ trees of distinct orders.

**Question:** Why didn’t this happen before?
**Goal:** Make tree sizes grow exponentially with order, but still allow for subtrees to be cut out quickly.

**Intuition:** Allow trees to get somewhat imbalanced, slowly propagating information to the root.

**Rule:** Nodes can lose at most one child. If a node loses two children, cut it from its parent.
**Goal:** Make tree sizes grow exponentially with order, but still allow for subtrees to be cut out quickly.

**Intuition:** Allow trees to get somewhat imbalanced, slowly propagating information to the root.

**Rule:** Nodes can lose at most one child. If a node loses two children, cut it from its parent.
**Goal:** Make tree sizes grow exponentially with order, but still allow for subtrees to be cut out quickly.

**Intuition:** Allow trees to get somewhat imbalanced, slowly propagating information to the root.

**Rule:** Nodes can lose at most one child. If a node loses two children, cut it from its parent.
**Goal:** Make tree sizes grow exponentially with order, but still allow for subtrees to be cut out quickly.

**Intuition:** Allow trees to get somewhat imbalanced, slowly propagating information to the root.

**Rule:** Nodes can lose at most one child. If a node loses two children, cut it from its parent.
**Goal:** Make tree sizes grow exponentially with order, but still allow for subtrees to be cut out quickly.

**Intuition:** Allow trees to get somewhat imbalanced, slowly propagating information to the root.

**Rule:** Nodes can lose at most one child. If a node loses two children, cut it from its parent.
**Intuition:** Allow trees to get somewhat imbalanced, slowly propagating information to the root.

**Rule:** Nodes can lose at most one child. If a node loses two children, cut it from its parent.

**Goal:** Make tree sizes grow exponentially with order, but still allow for subtrees to be cut out quickly.
**Goal:** Make tree sizes grow exponentially with order, but still allow for subtrees to be cut out quickly.

**Intuition:** Allow trees to get somewhat imbalanced, slowly propagating information to the root.

**Rule:** Nodes can lose at most one child. If a node loses two children, cut it from its parent.
**Goal:** Make tree sizes grow exponentially with order, but still allow for subtrees to be cut out quickly.

**Intuition:** Allow trees to get somewhat imbalanced, slowly propagating information to the root.

**Rule:** Nodes can lose at most one child. If a node loses two children, cut it from its parent.
**Intuition:** Allow trees to get somewhat imbalanced, slowly propagating information to the root.

**Rule:** Nodes can lose at most one child. If a node loses two children, cut it from its parent.

**Question:** Does this guarantee exponential tree size?
Maximally-Damaged Trees

- Here’s a binomial tree of order 4. That is, the root has four children.

- **Question**: Using our marking scheme, how many nodes can we remove without changing the order of the tree?

- Equivalently: how many nodes can we remove without removing any direct children of the root?
A maximally-damaged tree of order $k$ is a node whose children are maximally-damaged trees of orders $0, 0, 1, 2, 3, \ldots, k-2$. 
Maximally-Damaged Trees

Claim: The minimum number of nodes in a tree of order $k$ is $F_{k+2}$
Maximally-Damaged Trees

- **Theorem:** The number of nodes in a maximally-damaged tree of order $k$ is $F_{k+2}$.

- **Proof:** Induction.
Maximally-Damaged Trees

• **Theorem:** The number of nodes in a maximally-damaged tree of order \( k \) is \( F_{k+2} \).

• **Proof:** Induction.

Recall: \( F_k = \Theta(\phi^k) \)

The number of nodes in a tree grows exponentially!
A **Fibonacci heap** is a lazy binomial heap with **decrease-key** implemented using the marking scheme described earlier.
How fast are the operations on Fibonacci heaps?
Each *enqueue* slowly introduces trees.
Each *extract-min* rapidly cleans them up.

\[
\Phi = t
\]

where
\[
t \text{ is the number of trees.}
\]

Actual cost: $O(1)$
\[
\Delta \Phi: +1
\]

Amortized cost: $O(1)$. 
Each \textit{enqueue} slowly introduces trees.
Each \textit{extract-min} rapidly cleans them up.

\[ \Phi = t \]
where
\[ t \] is the number of trees.

This is the same analysis from last lecture!

Cost: \( O(t + \log n) \).
\( \Delta \Phi: O(-t + \log n) \).
Amortized cost: \( O(\log n) \).
Each *decrease-key* may trigger a chain of cuts. Those chains happen due to previous *decrease-keys*.

\[
\Phi = t
\]

*where*

\[
t \text{ is the number of trees.}
\]
Each *decrease-key* may trigger a chain of cuts. Those chains happen due to previous *decrease-keys*.

\[ \Phi = t \]

*where*

\[ t \text{ is the number of trees.} \]
Each **decrease-key** may trigger a chain of cuts. Those chains happen due to previous **decrease-keys**.

\[ \Phi = t \]

*where*

\[ t \]

is the number of trees.
Each *decrease-key* may trigger a chain of cuts. Those chains happen due to previous *decrease-keys*.

\[ \Phi = t \]

where

\[ t \] is the number of trees.
Each *decrease-key* may trigger a chain of cuts. Those chains happen due to previous *decrease-keys*.

\[ \Phi = t \]

where

\( t \) is the number of trees.
Each *decrease-key* may trigger a chain of cuts. Those chains happen due to previous *decrease-keys*.
Idea: Factor the number of marked nodes into our potential to offset the cost of cascading cuts.

\[ \Phi = t + m \]

where

\( t \) is the number of trees and \( m \) is the number of marked nodes.
**Idea:** Factor the number of marked nodes into our potential to offset the cost of cascading cuts.

\[ \Phi = t + m \]

where

\( t \) is the number of trees and
\( m \) is the number of marked nodes.
**Idea:** Factor the number of marked nodes into our potential to offset the cost of cascading cuts.

\[ \Phi = t + m \]

where

- \( t \) is the number of trees and
- \( m \) is the number of marked nodes.

Actual cost: \( O(1) \)
\[ \Delta \Phi: +2. \]

Amortized cost: \( O(1) \).
Idea: Factor the number of marked nodes into our potential to offset the cost of cascading cuts.

\[ \Phi = t + m \]

where

\( t \) is the number of trees and
\( m \) is the number of marked nodes.

Idea: Factor the number of marked nodes into our potential to offset the cost of cascading cuts.
**Idea:** Factor the number of marked nodes into our potential to offset the cost of cascading cuts.

\[ \Phi = t + m \]

\textit{where}

\( t \) is the number of trees and \( m \) is the number of marked nodes.

Actual cost: \( O(1) \)

\( \Delta \Phi: +2. \)

Amortized cost: \( O(1) \).

**Idea:** Factor the number of marked nodes into our potential to offset the cost of cascading cuts.
Idea: Factor the number of marked nodes into our potential to offset the cost of cascading cuts.

\[
\Phi = t + m
\]

where

\( t \) is the number of trees and \( m \) is the number of marked nodes.
Idea: Factor the number of marked nodes into our potential to offset the cost of cascading cuts.

\[ \Phi = t + m \]

where

- \( t \) is the number of trees and
- \( m \) is the number of marked nodes.

**Suppose this operation did \( C \) total cuts.**

**Actual cost:** \( O(C) \)

**\( \Delta \Phi: +1 \)**

**Amortized cost:** \( O(C) \).

**Idea:** Factor the number of marked nodes into our potential to offset the cost of cascading cuts.
Φ = \(t + 2m\)

where

\(t\) is the number of trees and \(m\) is the number of marked nodes.

**Idea 2:** Each *decrease-key* hurts twice: once in a cascading cut, and once in an *extract-min*. 
**Idea 2:** Each `decrease-key` hurts twice: once in a cascading cut, and once in an `extract-min`.

\[ \Phi = t + 2m \]

where

- \( t \) is the number of trees and
- \( m \) is the number of marked nodes.
Idea 2: Each decrease-key hurts twice: once in a cascading cut, and once in an extract-min.

\[ \Phi = t + 2m \]

where

- \( t \) is the number of trees and
- \( m \) is the number of marked nodes.

Actual cost: \( O(1) \)

\( \Delta \Phi: +3. \)

Amortized cost: \( O(1) \).
Idea 2: Each *decrease-key* hurts twice: once in a cascading cut, and once in an *extract-min*.

\[ \Phi = t + 2m \]

where

- \( t \) is the number of trees and
- \( m \) is the number of marked nodes.
Idea 2: Each *decrease-key* hurts twice: once in a cascading cut, and once in an *extract-min*.

\[ \Phi = t + 2m \]

where

- \( t \) is the number of trees and
- \( m \) is the number of marked nodes.

Actual cost: \( O(1) \)

\[ \Delta \Phi: +3. \]

Amortized cost: \( O(1) \).
Ideas:

1. Each decrease-key hurts twice: once in a cascading cut, and once in an extract-min.

\[ \Phi = t + 2m \]

where

- \( t \) is the number of trees and
- \( m \) is the number of marked nodes.

Idea 2: Each \textbf{decrease-key} hurts twice: once in a cascading cut, and once in an \textbf{extract-min}.
Idea 2: Each *decrease-key* hurts twice: once in a cascading cut, and once in an *extract-min*.

\[
\Phi = t + 2m
\]

*where*

\[
t \text{ is the number of trees and } m \text{ is the number of marked nodes.}
\]

Actual cost: \(O(C)\)

\[
\Delta \Phi: -C + 1
\]

Amortized cost: \(O(1)\).
The Overall Analysis

- Here’s the final scorecard for the Fibonacci heap.
- These are excellent theoretical runtimes. There’s minimal room for improvement!
- Later work made all these operations worst-case efficient at a significant increase in both runtime and intellectual complexity.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>enqueue</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>find-min</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>meld</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>extract-min</td>
<td>$O(\log n)^*$</td>
</tr>
<tr>
<td>decrease-key</td>
<td>$O(1)^*$</td>
</tr>
</tbody>
</table>

*amortized
Representation Issues
Representing Trees

• The trees in a Fibonacci heap must be able to do the following:
  • During a merge: Add one tree as a child of the root of another tree.
  • During a cut: Cut a node from its parent in time $O(1)$.

• **Claim:** This is trickier than it looks.
Representing Trees
Representing Trees

Finding this pointer might take time $\Theta(\log n)$!
The Solution

Each node stores a pointer to its parent.

The parent stores a pointer to an arbitrary child.

The children of each node are in a circularly, doubly-linked list.
To cut a node from its parent, if it isn't the representative child, just splice it out of its linked list.
The Solution

If it is the representative, change the parent's representative child to be one of the node's siblings.
Awful Linked Lists

- Trees are stored as follows:
  - Each node stores a pointer to some child.
  - Each node stores a pointer to its parent.
  - Each node is in a circularly-linked list of its siblings.

- The following possible are now possible in time $O(1)$:
  - Cut a node from its parent.
  - Add another child node to a node.
Fibonacci Heap Nodes

- Each node in a Fibonacci heap stores
  - A pointer to its parent.
  - A pointer to the next sibling.
  - A pointer to the previous sibling.
  - A pointer to an arbitrary child.
  - A bit for whether it's marked.
  - Its order.
  - Its key.
  - Its element.
In Practice

- In practice, the constant factors on Fibonacci heaps make it slower than other heaps, except on huge graphs or workflows with tons of decrease-keys.

- Why?
  - Huge memory requirements per node.
  - High constant factors on all operations.
  - Poor locality of reference and caching.
In Theory

- That said, Fibonacci heaps are worth knowing about for several reasons:
  - Clever use of a two-tiered potential function shows up in lots of data structures.
  - Implementation of *decrease-key* forms the basis for many other advanced priority queues.
  - Gives the theoretically optimal comparison-based implementation of Prim's and Dijkstra's algorithms.
More to Explore

• Since the development of Fibonacci heaps, there have been a number of other priority queues with similar runtimes.
  • In 1986, a powerhouse team (Fredman, Sedgewick, Sleator, and Tarjan) invented the pairing heap. It’s much simpler than a Fibonacci heap, is fast in practice, but its runtime bounds are unknown!
  • In 2012, Brodal et al. invented the strict Fibonacci heap. It has the same time bounds as a Fibonacci heap, but in a worst-case rather than amortized sense.
  • In 2013, Chan invented the quake heap. It matches the asymptotic bounds of a Fibonacci heap but uses a totally different strategy.
• Also interesting to explore: if the weights on the edges in a graph are chosen from a continuous distribution, the expected number of decrease-keys in Dijkstra’s algorithm is $O(n \log (m / n))$. That might counsel another heap structure!
Next Time

- **Randomized Data Structures**
  - Doing well on average, broadly speaking.

- **Frequency Estimation**
  - Counting in sublinear space.

- **Count-Min Sketches**
  - A simple, elegant, fast, and widely-used data structure.