Binomial Heaps
Outline for this Week

• **Binomial Heaps (Today)**
  • A simple, flexible, and versatile priority queue.

• **Lazy Binomial Heaps (Today)**
  • A powerful building block for designing advanced data structures.

• **Fibonacci Heaps (Thursday)**
  • A heavyweight and theoretically excellent priority queue.
Review: Priority Queues
Priority Queues

• A **priority queue** is a data structure that stores a set of elements annotated with totally-ordered *keys* and allows efficient extraction of the element with the least key.

• More concretely, supports these operations:
  
  • `pq.enqueue(v, k)`, which enqueues element `v` with key `k`;
  
  • `pq.find-min()`, which returns the element with the least key; and
  
  • `pq.extract-min()`, which removes and returns the element with the least key,
Binary Heaps

- Priority queues are frequently implemented as **binary heaps**.
- **enqueue** and **extract-min** run in time $O(\log n)$; **find-min** runs in time $O(1)$.
- We're not going to cover binary heaps this quarter; I assume you've seen them before.
Priority Queues in Practice

• Many graph algorithms directly rely priority queues supporting extra operations:
  
  • $\text{meld}(pq_1, pq_2)$: Destroy $pq_1$ and $pq_2$ and combine their elements into a single priority queue.
  
  • $pq.\text{decrease-key}(v, k')$: Given a pointer to element $v$ already in the queue, lower its key to have new value $k'$.
  
  • $pq.\text{add-to-all}(\Delta k)$: Add $\Delta k$ to the keys of each element in the priority queue (typically used with $\text{meld}$).

• In lecture, we'll cover binomial heaps to efficiently support $\text{meld}$ and Fibonacci heaps to efficiently support $\text{meld}$ and $\text{decrease-key}$.

• You'll design a priority queue supporting efficient $\text{meld}$ and $\text{add-to-all}$ on the problem set.
Meldable Priority Queues

- A priority queue supporting the \textit{meld} operation is called a \textit{meldable priority queue}.
- \textit{meld}(pq_1, pq_2) destructively modifies \textit{pq}_1 and \textit{pq}_2 and produces a new priority queue containing all elements of \textit{pq}_1 and \textit{pq}_2.
Meldable Priority Queues

- A priority queue supporting the *meld* operation is called a **meldable priority queue**.
- **meld**($pq_1$, $pq_2$) destructively modifies $pq_1$ and $pq_2$ and produces a new priority queue containing all elements of $pq_1$ and $pq_2$. 
Efficiently Meldable Queues

- Standard binary heaps do not efficiently support \textit{meld}.

- \textbf{Intuition}: Binary heaps are complete binary trees, and two complete binary binary trees cannot easily be linked to one another.
Binomial Heaps

- The **binomial heap** is an priority queue data structure that supports efficient melding.

- We'll study binomial heaps for several reasons:
  
  - Implementation and intuition is totally different than binary heaps.
  
  - Used as a building block in other data structures (Fibonacci heaps, soft heaps, etc.)
  
  - Has a beautiful intuition; similar ideas can be used to produce other data structures.
Supporting Efficient Melding
The Intuition: *Binary Arithmetic*
Adding Binary Numbers

• Given the binary representations of two numbers $n$ and $m$, we can add those numbers in time $\Theta(\max\{\log m, \log n\})$. 

\[
\begin{array}{cccccc}
1 & 1 & 1 & 1 & 1 \\
1 & 0 & 1 & 1 & 0 \\
+ & 1 & 1 & 1 & 1 \\
\hline
1 & 0 & 0 & 1 & 0 & 1
\end{array}
\]
A Different Intuition

- Represent $n$ and $m$ as a collection of “packets” whose sizes are powers of two.
- Adding together $n$ and $m$ can then be thought of as combining the packets together, eliminating duplicates.
Building a Priority Queue

- **Idea:** Adapt this approach to build a priority queue.
- Store elements in the priority queue in “packets” whose sizes are powers of two.
- Store packets in ascending size order.
- We'll choose a representation of a packet so that two packets of the same size can easily be fused together.
Building a Priority Queue

- What properties must our packets have?
  - Sizes must be powers of two.
  - Can efficiently fuse packets of the same size.
  - Can efficiently find the minimum element of each packet.
Inserting into the Queue

- If we can efficiently meld two priority queues, we can efficiently enqueue elements to the queue.

- **Idea:** Meld together the queue and a new queue with a single packet.
Deleting the Minimum

- Our analogy with arithmetic breaks down when we try to remove the minimum element.
- After losing an element, the packet will not necessarily hold a number of elements that is a power of two.
Fracturing Packets

- If we have a packet with $2^k$ elements in it and remove a single element, we are left with $2^k - 1$ remaining elements.

- **Fun fact**: $2^k - 1 = 1 + 2 + 4 + \ldots + 2^{k-1}$.

- **Idea**: “Fracture” the packet into $k - 1$ smaller packets, then add them back in.
Fracturing Packets

- We can *extract-min* by fracturing the packet containing the minimum and adding the fragments back in.
Fracturing Packets

- We can *extract-min* by fracturing the packet containing the minimum and adding the fragments back in.

- Runtime is $O(\log n)$ fuses in *meld*, plus fragment cost.
Building a Priority Queue

• What properties must our packets have?
  • Size must be a power of two.
  • Can efficiently fuse packets of the same size.
  • Can efficiently find the minimum element of each packet.
  • Can efficiently “fracture” a packet of $2^k$ nodes into packets of 1, 2, 4, 8, ..., $2^{k-1}$ nodes.
• What representation of packets will give us these properties?
Binomial Trees

- A **binomial tree of order** $k$ is a type of tree recursively defined as follows:

  A binomial tree of order $k$ is a single node whose children are binomial trees of order $0, 1, 2, ..., k - 1$.

- Here are the first few binomial trees:
Binomial Trees

- **Theorem:** A binomial tree of order $k$ has exactly $2^k$ nodes.

- **Proof:** Induction on $k$. Assuming that binomial trees of orders 0, 1, 2, ..., $k - 1$ have $2^0, 2^1, 2^2, ..., 2^{k-1}$ nodes, then the number of nodes in an order-$k$ binomial tree is

$$2^0 + 2^1 + ... + 2^{k-1} + 1 = 2^k - 1 + 1 = 2^k$$

So the claim holds for $k$ as well. ■
Binomial Trees

- A **heap-ordered binomial tree** is a binomial tree whose nodes obey the heap property: all nodes are less than or equal to their descendants.
- We will use heap-ordered binomial trees to implement our “packets.”

```
      5
     / \
    2  2
   /    \
  9     1
 /  \\  /  \
3   7 5
```
Binomial Trees

- What properties must our packets have?
  - Size must be a power of two. ✓
  - Can efficiently fuse packets of the same size.
  - Can efficiently find the minimum element of each packet.
  - Can efficiently “fracture” a packet of $2^k$ nodes into packets of 1, 2, 4, 8, ..., $2^{k-1}$ nodes.
Binomial Trees

- What properties must our packets have?
  - Size must be a power of two. ✓
  - Can efficiently fuse packets of the same size. ✓
  - Can efficiently find the minimum element of each packet.
  - Can efficiently “fracture” a packet of $2^k$ nodes into packets of 1, 2, 4, 8, ..., $2^{k-1}$ nodes.

Make the binomial tree with the larger root the first child of the tree with the smaller root.
Binomial Trees

- What properties must our packets have?
  - Size must be a power of two. ✓
  - Can efficiently fuse packets of the same size. ✓
  - Can efficiently find the minimum element of each packet. ✓
  - Can efficiently “fracture” a packet of $2^k$ nodes into packets of 1, 2, 4, 8, ..., $2^{k-1}$ nodes.
Binomial Trees

What properties must our packets have?

- Size must be a power of two. ✓
- Can efficiently fuse packets of the same size. ✓
- Can efficiently find the minimum element of each packet. ✓
- Can efficiently “fracture” a packet of $2^k$ nodes into packets of $1, 2, 4, 8, \ldots, 2^{k-1}$ nodes. ✓
The Binomial Heap

- A **binomial heap** is a collection of heap-ordered binomial trees stored in ascending order of size.

- Operations defined as follows:
  - **meld**($pq_1, pq_2$): Use addition to combine all the trees.
    - Fuses $O(\log n)$ trees. Total time: $O(\log n)$.
  - $pq$.enqueue($v, k$): Meld $pq$ and a singleton heap of $(v, k)$.
    - Total time: $O(\log n)$.
  - $pq$.find-min(): Find the minimum of all tree roots.
    - Total time: $O(\log n)$.
  - $pq$.extract-min(): Find the min, delete the tree root, then meld together the queue and the exposed children.
    - Total time: $O(\log n)$.
An Issue of Representation

- Binomial trees are *logically* multiway trees, but are typically *implemented* as binary trees.
- We use the *left-child/right-sibling* representation.
- Each node’s left pointer points to its first child.
- Each node’s right pointer points to its next sibling.
An Issue of Representation

- Binomial trees are logically multiway trees, but are typically implemented as binary trees.
- We use the left-child/right-sibling representation.
- Each node’s left pointer points to its first child.
- Each node’s right pointer points to its next sibling.
An Issue of Representation

- The LCRS representation of binomial trees improves efficiency.
- Fusion takes time $O(1)$.
- Fracturing takes time $O(\log n)$. 

![Diagram](image-url)
An Issue of Representation

- The LCRS representation of binomial trees improves efficiency.
- Fusion takes time $O(1)$.
- Fracturing takes time $O(\log n)$. 
An Issue of Representation

- The LCRS representation of binomial trees improves efficiency.
- Fusion takes time $O(1)$.
- Fracturing takes time $O(\log n)$. 
Time-Out for Announcements!
Stanford Women in Computer Science

Casual Dinner

Tuesday, May 1st from 6-7 PM at Gates 403

Come mingle with CS professors and friends with delicious Lotus Thai!
The Final Project

• We’ve just posted information online (and in hardcopy here) about the CS166 final project.

• The quick summary:
  • Work in teams of two or three.
  • Pick a data structure, algorithm, or technique of your choice.
  • Become experts on it. Put together a writeup and presentation on the topic.
  • Do something “interesting” with it. You have broad latitude how to interpret what “interesting” means – pick something you’re excited about!

• Projects and presentations are due in the last week of class. They’re usually a highlight of the quarter for everyone involved!
Project Proposals

- Before working on the project, you’ll need to submit a proposal about what you’d like to work on.
- Your proposal should consist of a ranked list of five data structures you’d be interested in exploring, along with some preliminary information about each one.
- The proposal is due next \textit{Thursday, May 10}^{\text{th}} \text{ at} 2:30 PM.
- We’ll do a global matchmaking to assign topics over that weekend.
Problem Sets

- Problem Set Three is due Thursday at 2:30PM.
  - There’s plenty of space to ask us questions – let us know what we can do to help out!
- Problem Set Four will go out next Tuesday. You’ll have a little gap between those problem sets.
  - We recommend using this gap to work on or think about your final project proposals.
Back to CS166!
Analyzing Insertions

- Each *enqueue* into a binomial heap takes time $O(\log n)$, since we have to meld the new node into the rest of the trees.

- However, it turns out that the amortized cost of an insertion is lower in the case where we do a series of $n$ insertions.
Adding One

- Suppose we want to execute $n++$ on the binary representation of $n$.

- Do the following:
  - Find the longest span of 1's at the right side of $n$.
  - Flip those 1's to 0's.
  - Set the preceding bit to 1.

- Runtime: $\Theta(b)$, where $b$ is the number of bits flipped.
An Amortized Analysis

- **Claim**: Starting at zero, the amortized cost of adding one to the total is $O(1)$.
- **Idea**: Use as a potential function the number of 1's in the number.
An Amortized Analysis

• **Claim:** Starting at zero, the amortized cost of adding one to the total is $O(1)$.

• **Idea:** Use as a potential function the number of 1's in the number.

\[
\Phi = 1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1
\]

- Actual cost: 1
- $\Delta \Phi$: +1
- Amortized cost: 2
An Amortized Analysis

- **Claim:** Starting at zero, the amortized cost of adding one to the total is $O(1)$.

- **Idea:** Use as a potential function the number of 1's in the number.

\[
\Phi = 1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 0 \quad 0
\]

Actual cost: 2
\[
\Delta \Phi: 0
\]
Amortized cost: 2
An Amortized Analysis

- **Claim:** Starting at zero, the amortized cost of adding one to the total is $O(1)$.
- **Idea:** Use as a potential function the number of 1's in the number.

<table>
<thead>
<tr>
<th>Actual cost</th>
<th>$\Delta \Phi$</th>
<th>Amortized cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

$\Phi = 2$
An Amortized Analysis

- **Claim**: Starting at zero, the amortized cost of adding one to the total is $O(1)$.
- **Idea**: Use as a potential function the number of 1's in the number.

$\Phi = 1$

$0 \quad 0 \quad 1 \quad 0 \quad 0$

Actual cost: 3  
$\Delta \Phi$: -1  
Amortized cost: 2
Properties of Binomial Heaps

- Starting with an empty binomial heap, the amortized cost of each insertion into the heap is O(1), assuming there are no deletions.

- **Rationale:** Binomial heap operations are isomorphic to integer arithmetic.

- Since the amortized cost of incrementing a binary counter starting at zero is O(1), the amortized cost of enqueuing into an initially empty binomial heap is O(1).
Binomial vs Binary Heaps

• Interesting comparison:
  • The cost of inserting $n$ elements into a binary heap, one after the other, is $\Theta(n \log n)$ in the worst-case.
  • If $n$ is known in advance, a binary heap can be constructed out of $n$ elements in time $\Theta(n)$.
  • The cost of inserting $n$ elements into a binomial heap, one after the other, is $\Theta(n)$, even if $n$ is not known in advance!
A Catch

- This amortized time bound does not hold if *enqueue* and *extract-min* are intermixed.
- **Intuition:** Can force expensive insertions to happen repeatedly.
**Question:** Can we make insertions amortized O(1), regardless of whether we do deletions?
Where's the Cost?

- Why does *enqueue* take time $O(\log n)$?
- **Answer**: May have to combine together $O(\log n)$ different binomial trees together into a single tree.

- **New Question**: What happens if we don't combine trees together?
- That is, what if we just add a new singleton tree to the list?
Lazy Melding

• More generally, consider the following lazy melding approach:

  To meld together two binomial heaps, just combine the two sets of trees together.

• If we assume the trees are stored in doubly-linked lists, this can be done in time $O(1)$. 

![Diagram of trees for melding](image)
The Catch: Part One

- When we use eager melding, the number of trees is $O(\log n)$.
- Therefore, \texttt{find-min} runs in time $O(\log n)$.
- \textbf{Problem:} \texttt{find-min} no longer runs in time $O(\log n)$ because there can be $\Theta(n)$ trees.
A Solution

- Have the binomial heap store a pointer to the minimum element.
- Can be updated in time $O(1)$ after doing a meld by comparing the minima of the two heaps.
The Catch: Part Two

- Even with a pointer to the minimum, deletions might now run in time $\Theta(n)$.
- **Rationale:** Need to update the pointer to the minimum.
Resolving the Issue

**Idea:** When doing an *extract-min*, coalesce all of the trees so that there's at most one tree of each order.

**Intuitively:**

- The number of trees in a heap grows slowly (only during an insert or meld).
- The number of trees in a heap drops rapidly after coalescing (down to $O(\log n)$).
- Can backcharge the work done during an *extract-min* to *enqueue* or *meld*. 
Coalescing Trees

- Our eager melding algorithm assumes that
  - there is either zero or one tree of each order, and that
  - the trees are stored in ascending order.
- **Challenge:** When coalescing trees in this case, neither of these properties necessarily hold.
Wonky Arithmetic

• Compute the number of bits necessary to hold the sum.
  • Only $O(\log n)$ bits are needed.
• Create an array of that size, initially empty.
• For each packet:
  • If there is no packet of that size, place the packet in the array at that spot.
  • If there is a packet of that size:
    − Fuse the two packets together.
    − Recursively add the new packet back into the array.
Now With Trees!

- Compute the number of trees necessary to hold the nodes.
  - Only $O(\log n)$ trees are needed.
- Create an array of that size, initially empty.
- For each tree:
  - If there is no tree of that size, place the tree in the array at that spot.
  - If there is a tree of that size:
    - Fuse the two trees together.
    - Recursively add the new tree back into the array.
Coalescing Trees

Total number of nodes: 15
(Can compute in time $\Theta(T)$, where $T$ is the number of trees, if each tree is tagged with its order)

Bits needed: 4
Coalescing Trees
Analyzing Coalesce

• Suppose there are \( T \) trees.
• We spend \( \Theta(T) \) work iterating across the main list of trees twice:
  • Pass one: Count up number of nodes (if each tree stores its order, this takes time \( \Theta(T) \)).
  • Pass two: Place each node into the array.
• Each merge takes time \( O(1) \).
• The number of merges is \( O(T) \).
• Total work done: \( \Theta(T) \).
• In the worst case, this is \( O(n) \).
The Story So Far

- A binomial heap with lazy melding has these worst-case time bounds:
  - **enqueue**: \( O(1) \)
  - **meld**: \( O(1) \)
  - **find-min**: \( O(1) \)
  - **extract-min**: \( O(n) \).

- These are **worst-case** time bounds. What about an **amortized** time bounds?
An Observation

• The expensive step here is $\textit{extract-min}$, which runs in time proportional to the number of trees.

• Each tree can be traced back to one of three sources:
  • An $\textit{enqueue}$.
  • A $\textit{meld}$ with another heap.
  • A tree exposed by an $\textit{extract-min}$.

• Let's use an amortized analysis to shift the blame for the $\textit{extract-min}$ performance to other operations.
The Potential Method

• We will use the potential method in this analysis.

• When analyzing insertions with eager merges, we set $\Phi(D)$ to be the number of trees in $D$.

• Let's see what happens if we use this $\Phi$ here.
Analyzing an Insertion

- To *enqueue* a key, we add a new binomial tree to the forest and possibly update the min pointer.

  Actual time: $O(1)$. $\Delta \Phi: +1$

  Amortized cost: $O(1)$.
Analyzing a Meld

- Suppose that we **meld** two lazy binomial heaps $B_1$ and $B_2$. Actual cost: $O(1)$.
- Let $\Phi_{B_1}$ and $\Phi_{B_2}$ be the initial potentials of $B_1$ and $B_2$.
- The new heap $B$ has potential $\Phi_{B_1} + \Phi_{B_2}$ and $B_1$ and $B_2$ have potential 0.
- $\Delta \Phi$ is zero.
- Amortized cost: $O(1)$. 

```
  3
  /\  \\
 6   4
  \\
  8   
```

```
    7     5
  /\  1 \\
 2   3
  \\
  9
```
Analyzing a Find-Min

- Each $\text{find-min}$ does $O(1)$ work and does not add or remove trees.
- Amortized cost: $O(1)$. 
Analyzing Extract-Min

- Suppose we perform an *extract-min* on a binomial heap with $T$ trees in it.
- Initially, we expose the children of the minimum element. This increases the number of trees to $T + O(\log n)$.
- The runtime for coalescing these trees is $O(T + \log n)$.
- When we're done merging, there will be $O(\log n)$ trees remaining, so $\Delta \Phi = -T + O(\log n)$.
- Amortized cost is

  \[
  \Theta(T + \log n) + O(1) \cdot (-T + O(\log n))
  = \Theta(T) - O(1) \cdot T + O(1) \cdot O(\log n)
  = O(\log n).
  \]
The Overall Analysis

- The *amortized* costs of the operations on a lazy binomial heap are as follows:
  - *enqueue*: $O(1)$
  - *meld*: $O(1)$
  - *find-min*: $O(1)$
  - *extract-min*: $O(\log n)$

- Any series of $e$ *enqueue*es mixed with $d$ *extract-min*es will take time $O(e + d \log e)$. 
Why This Matters

- Lazy binomial heaps are a powerful building block used in many other data structures.
- We'll see one of them, the *Fibonacci heap*, when we come back on Thursday.
- You'll see another (supporting *add-to-all*) on the problem set.
Next Time

- **The Need for decrease-key**
  - A powerful and versatile operation on priority queues.

- **Fibonacci Heaps**
  - A variation on lazy binomial heaps with efficient decrease-key.

- **Implementing Fibonacci Heaps**
  - ... is harder than it looks!