Balanced Trees
Part Two
Outline for Today

• Recap from Last Time
  • Review of B-trees, 2-3-4 trees, and red/black trees.

• Augmented Binary Search Trees
  • Leveraging red/black trees.

• Joining and Splitting Trees
  • Two powerful BST primitives.
Recap from Last Time
B-Trees

- A **B-tree of order** \( b \) **is** a multiway search tree where
  - each node has between \( b-1 \) and \( 2b-1 \) keys, except the root, which may have as few as one key;
  - each node is either a leaf or has one more child than key; and
  - all leaves are at the same depth.
Red/Black Trees

- A **red/black tree** is a BST with the following properties:
  - Every node is either red or black.
  - The root is black.
  - No red node has a red child.
  - Every root-null path in the tree passes through the same number of black nodes.
- After we hoist red nodes into their parents:
  - Each “meta node” has 1, 2, or 3 keys in it. (No red node has a red child.)
  - Each “meta node” is either a leaf or has one more key than node. (Root-null path property.)
  - Each “meta leaf” is at the same depth. (Root-null path property.)

This is a 2-3-4 tree!
Data Structure Isometries

• Red/black trees are an *isometry* of 2-3-4 trees; they represent the structure of 2-3-4 trees in a different way.

• Accordingly, red/black trees have height $O(\log n)$.

• After inserting or deleting an element from a red/black tree, the tree invariants can be fixed up in time $O(\log n)$ by applying rotations and color flips that simulate a 2-3-4 tree.
Tree Rotations

Diagram showing tree rotations with nodes A and B and various comparisons such as <A, >A, <B, >B.
New Stuff!
Dynamic Problems
Dynamic Problems

• The “classic” algorithms model goes something like this:

Given some input $X$, compute some interesting function $f(X)$.

• This assumes that $X$ is specified up-front and doesn’t change over time.

• These questions typically become more interesting when they’re made dynamic and the model looks more like this:

   Given some input $X$ that changes over time, maintain a data structure that makes it easy to compute $f(X)$ at any instant in time.

• Many data structures can essentially be thought of as solutions to dynamic versions of classical algorithms problems.
Dynamic Order Statistics
Order Statistics

- In a set $S$ of totally ordered values, the $k$th order statistic is the $k$th smallest value in the set.
  - The $0^{th}$ order statistic is the minimum value.
  - The $1^{st}$ order statistic is the second-smallest value.
  - The $(n – 1)^{st}$ order statistic is the maximum value.
- In the static case (when the data set is given to you in advance), algorithms like quickselect and median-of-mediants give (possibly randomized) $O(n)$-time solutions to order statistics.
- **Goal:** Solve this problem efficiently when the data set is changing – that is, the underlying set of elements can have insertions and deletions intermixed with queries.
Finding Order Statistics
Finding Order Statistics

The diagram represents a binary search tree with node values that correspond to the order statistics of a set. Each node contains a value, and the tree structure helps in efficiently finding the order statistics.
Finding Order Statistics
Finding Order Statistics

Problem: After inserting a new value, we may have to update $\Theta(n)$ values.
An Observation

• The exact index of each number is a \textit{global property} of the tree.
  • Depends on all other nodes and their positions.

• Could we find a \textit{local property} that lets us find order statistics?
  • That is, something that depends purely on nearby nodes.
Finding Order Statistics
Finding Order Statistics
Finding Order Statistics
Finding Order Statistics

Each node is annotated with the number of children in its left subtree.
Finding Order Statistics

Since the number just holds the number of nodes in its left subtree, we only need to increment the value for nodes that have the new node in its left subtree.
Finding Order Statistics

How do we update the numbers after the rotation?
Order Statistic Trees

- The tree we just saw is called an order statistics tree.
  - Start with a red/black tree.
  - Tag each node with the number of nodes in its left subtree.
  - Use the preceding update rules to preserve values during rotations.
  - Propagate other changes up to the root of the tree.
- Only $O(\log n)$ values must be updated on an insertion or deletion and each can be updated in time $O(1)$.
- Supports all BST operations plus select (find kth order statistic) and rank (tell index of value) in time $O(\log n)$. 
Generalizing our Idea
The General Pattern

- This data structure works in the appropriate time bounds because values only change on an insertion or deletion:
  - along the root-leaf access path, and
  - during rotations.
- Red/black trees have height $O(\log n)$ and require only $O(\log n)$ rotations per insertion or deletion.
- We can augment red/black trees with any attributes we'd like as long as they obey these properties.
Augmented Red/Black Trees

• Let $f(node)$ be a function with the following properties:
  • $f$ can be computed in time $O(1)$.
  • $f$ can be computed at a node based purely on that node's key and the values of $f$ computed at node's children.

• **Theorem:** The values of $f$ can be cached in the nodes of a red/black tree without changing the asymptotic runtime of insertions or deletions.

• **Proof sketch:** After inserting or deleting a node, the only values that need to change are along the root-leaf access path, plus values at nodes that were rotated. There are only $O(\log n)$ of these.
Augmented Red/Black Trees

$f$ can be computed at a node based purely on the key in that node and the values of $f$ in it that node's children.
Order Statistics

- **Note:** The approach we took for building order statistic trees does not fall into this framework.

- **Example:** The values below denote the number of nodes in the indicated nodes' left subtrees. What is the correct value of $x$?

![Diagram](image_url)
Order Statistics via Augmentation

• Have each node store two quantities:
  • \textit{numLeft}, the number of nodes in the left subtree.
  • \textit{numRight}, the number of nodes in the right subtree.
• Can compute this information at a node in time \(O(1)\) based on subtree values:
  • \(n.\text{numLeft} = n.\text{left}.\text{numLeft} + n.\text{left}.\text{numRight} + 1\)
  • \(n.\text{numRight} = n.\text{right}.\text{numLeft} + n.\text{right}.\text{numRight} + 1\)
• This fits into our framework, so we know that red/black trees can be augmented this way without needing to reason about tree rotations.
• Useful if we want to show feasibility; we can always optimize later if we need to.
Example: *Hierarchical Clustering*
1D Hierarchical Clustering

<table>
<thead>
<tr>
<th>20</th>
<th>42</th>
<th>44</th>
<th>60</th>
<th>66</th>
<th>71</th>
<th>86</th>
<th>92</th>
<th>100</th>
</tr>
</thead>
</table>

| 20 | 42 | 44 | 60 | 66 | 71 | 86 | 92 | 100 |

| 20 | 42 | 44 | 60 | 66 | 71 | 86 | 92 | 100 |
1D Hierarchical Clustering
1D Hierarchical Clustering

<table>
<thead>
<tr>
<th>20</th>
<th>43</th>
<th>60</th>
<th>68.5</th>
<th>86</th>
<th>92</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>44</td>
<td></td>
<td>66</td>
<td>71</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
20  42  44  60  66  71  86  92  100
```
1D Hierarchical Clustering
1D Hierarchical Clustering

<table>
<thead>
<tr>
<th>20</th>
<th>43</th>
<th>65.67</th>
<th>89</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>44</td>
<td>60</td>
<td>66</td>
<td>71</td>
</tr>
<tr>
<td></td>
<td></td>
<td>86</td>
<td>92</td>
<td></td>
</tr>
</tbody>
</table>

Diagram showing the hierarchical clustering process with distances and node connections.
1D Hierarchical Clustering

<table>
<thead>
<tr>
<th></th>
<th>20</th>
<th>43</th>
<th>65.67</th>
<th>92.67</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>42</td>
<td>44</td>
<td>60</td>
<td>86</td>
</tr>
<tr>
<td></td>
<td>66</td>
<td>71</td>
<td>92</td>
<td>100</td>
</tr>
</tbody>
</table>

Diagram showing the hierarchical clustering process.
1D Hierarchical Clustering

<table>
<thead>
<tr>
<th>20</th>
<th>56.6</th>
<th>92.67</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>60</td>
<td>66</td>
</tr>
<tr>
<td>44</td>
<td>71</td>
<td>86</td>
</tr>
<tr>
<td>60</td>
<td>66</td>
<td>92</td>
</tr>
<tr>
<td>71</td>
<td>86</td>
<td>100</td>
</tr>
</tbody>
</table>
1D Hierarchical Clustering
1D Hierarchical Clustering
This tree is called a **dendrogram**.
Analyzing the Runtime

• How efficient is this algorithm?
  • Number of rounds: $\Theta(n)$.
  • Work to find closest pair: $O(n)$.
  • Total runtime: $\Theta(n^2)$.

• Can we do better?
Dynamic 1D Closest Points

• The *dynamic 1D closest points problem* is the following:

  Maintain a set of real numbers undergoing insertion and deletion while efficiently supporting queries of the form “what is the closest pair of points?”

• Can we build a better data structure for this?
Dynamic 1D Closest Points

\[ k \]

\[ \text{max} \]

\[ \text{min} \]
A Tree Augmentation

- Augment each node to store the following:
  - The maximum value in the tree.
  - The minimum value in the tree.
  - The closest pair of points in the tree.
- **Claim:** Each of these properties can be computed in time $O(1)$ from the left and right subtrees.
- These properties can be augmented into a red/black tree so that insertions and deletions take time $O(\log n)$ and “what is the closest pair of points?” can be answered in time $O(1)$. 
Dynamic 1D Closest Points

<table>
<thead>
<tr>
<th>Value</th>
<th>Min</th>
<th>Max</th>
<th>Closest</th>
</tr>
</thead>
<tbody>
<tr>
<td>137</td>
<td>?</td>
<td>?</td>
<td>?, ?</td>
</tr>
<tr>
<td>42</td>
<td>-17</td>
<td>67</td>
<td>15, 21</td>
</tr>
<tr>
<td>271</td>
<td>142</td>
<td>415</td>
<td>300, 310</td>
</tr>
</tbody>
</table>
Dynamic 1D Closest Points

137
Min: -17
Max: 415
Closest: 137, 142

42
Min: -17
Max: 67
Closest: 15, 21

271
Min: 142
Max: 415
Closest: 300, 310
Some Other Questions

• How would you augment this tree so that you can efficiently (in time $O(1)$) compute the appropriate weighted averages?

• **Trickier:** Is this the fastest possible algorithm for this problem?
  • What if you’re guaranteed that the keys are all integers in some nice range?
A Helpful Intuition
Divide-and-Conquer

- Initially, it can be tricky to come up with the right tree augmentations.
- **Useful intuition:** Imagine you're writing a divide-and-conquer algorithm over the elements and have $O(1)$ time per “conquer” step.
Time-Out for Announcements!
Grace Hopper Tickets

- The Stanford CS department will be sponsoring thirty students to attend the Grace Hopper Celebration of Women in Computing.
  - Adjectives used to describe the experience include “life-changing,” “transformative,” “incredible,” etc.
- We’re covering tickets, airfare, hotels, and meals.
- Available only to CS majors / grad students; priority is to students who haven’t attended.
- *Highly recommended*. [Apply online](#).
FastestRMQ Results
Two hours and twenty minutes of time testing later, the results are in!
Honorable Mentions

Dana Murphy and Eric Martin

\(O(n), O(\log n)\) hybrid.
Linearized sparse table.
Honorable Mentions

Adam Pahlavan and Sumer Sao

$O(n), O(\log n)$ hybrid. Populate sparse table with SIMD.
Silver Medalists

Shaggy Goel and Kevin Hu

$\langle O(n), O(n^{0.2} \log n) \rangle$ hybrid. Precompute to-end-of-block queries.
Gold Medalists

Dillon Kanne and Kai Ang

Choose one of the following based on $n$:

- $\langle O(n \log n), O(1) \rangle$ sparse table,
- one-layer $\langle O(n), O(\log n) \rangle$ hybrid, or
- two-layer $\langle O(n), O(\log n) \rangle$ hybrid.
Lessons Learned

- What’s fast in Theoryland isn’t always what’s fastest in practice.
  - None of the top ten solutions used Fischer-Heun.
- Locality of reference has a huge effect on runtime.
  - Two of the top four solutions aggressively optimized the sparse table for cache friendliness.
  - The top ten solutions all used linear scans as a key part of their strategy.
- Tuning parameters has a huge effect on runtime.
  - Two of the top three solutions changed the block size. The second-fastest solution abandoned $O(\log n)$ for $O(n^{0.2})$.
- Combine lots of different solutions together.
  - Three of the top eight solutions (including the top solution) selected different data structures based on input size.

*Thanks to everyone who submitted something!*
Back to CS166!
Joining and Splitting Trees
Joining Trees

- The operation $\text{join}(T_1, k, T_2)$ takes as input
  - a BST $T_1$;
  - a key $k$, where $k$ is greater than all keys in $T_1$; and
  - a BST $T_2$, where $k$ is less than all keys in $T_2$; then
  destructively modifies $T_1$ and $T_2$ to produce a new BST containing all keys in $T_1$ and $T_2$ and the key $k$. 

![Diagram showing the join operation on two trees](image-url)
Joining Trees

- The operation $\textit{join}(T_1, k, T_2)$ takes as input
  - a BST $T_1$;
  - a key $k$, where $k$ is greater than all keys in $T_1$; and
  - a BST $T_2$, where $k$ is less than all keys in $T_2$; then de destructively modifies $T_1$ and $T_2$ to produce a new BST containing all keys in $T_1$ and $T_2$ and the key $k$. 

\[ T \]
Splitting Trees

- The operation \textit{split}(T, k) takes as input
  a BST \( T \), and
  a key \( k \), then
  destructively modifies BST \( T \) and forms BSTs \( T_1 \) and \( T_2 \)
  such that all keys in \( T_1 \) are less than or equal to \( k \) and
  all keys in \( T_2 \) are greater than \( k \).
Splitting Trees

• The operation \textit{split}(T, k) takes as input
  
  a BST $T$, and
  
  a key $k$, then
  
  destructively modifies BST $T$ and forms BSTs $T_1$ and $T_2$
  such that all keys in $T_1$ are less than or equal to $k$
  and all keys in $T_2$ are greater than $k$. 

\begin{tikzpicture}
  \node (T1) at (2,4) {$T_1$};
  \node (T2) at (6,4) {$T_2$};
  \node (k) at (4,0) {$k$};
  \draw[red, line width=2pt] (T1) -- (k) -- (T2);
\end{tikzpicture}
The Runtimes

- Both of these operations can be implemented in time $O(n)$ by completely rebuilding the trees from scratch.
  - Good exercise: determine how to do this.
- Amazingly, using augmented red/black trees:
  - $\textit{join}(T_1, k, T_2)$ can be made to run in time $\Theta(1 + |bh_1 - bh_2|)$, where $bh_1$ and $bh_2$ are the number of black nodes on any root-null path in $T_1$ and $T_2$, respectively, and
  - $\textit{split}(T, k)$ can be made to run in time $O(\log n)$.
- How is this possible?
Joining 2-3-4 Trees

• The isometry between 2-3-4 trees and red/black trees is very useful here.

• Let's see how to join two 2-3-4 trees and a key together.

• Based on what we find, we'll develop an efficient algorithm for joining red/black trees.
Joining 2-3-4 Trees
Joining 2-3-4 Trees
Joining 2-3-4 Trees
Joining 2-3-4 Trees
Joining 2-3-4 Trees
Joining 2-3-4 Trees
Joining 2-3-4 Trees

• To $\text{join}(T_1, k, T_2)$:
  • Assume that $T_1$ is the taller of the two trees; if not, do the following, but mirrored.
  • Walk down the right spine of $T_1$ until a node $v$ is found whose height is the height of $T_2$.
  • Add $k$ as a final key of $v$'s parent with $T_2$ as a right child.
  • Continue as if you were inserting $k$ into $v$'s parent – possibly split the node and propagate upward, etc.
Analyzing the Runtime

• Assume all 2-3-4 tree nodes are annotated with their heights.

• What is the runtime of $\text{join}(T_1, k, T_2)$?

• Runtime is $\Theta(1 + |h_1 - h_2|)$.
Joining Red/Black Trees
Joining Red/Black Trees
Joining Red/Black Trees
Joining Red/Black Trees
Joining 2-3-4 Trees

- Define the **black height** of a node to be the number of black nodes on any root-null path starting at that node.

- To **join**($T_1$, $k$, $T_2$):
  - Assume that $T_1$ is the tree with larger black height; if not, do the following, but mirrored.
  - Walk down the right spine of $T_1$ until a black node $v$ is found whose black height is the black height of $T_2$.
  - Insert a new node with key $k$, left child $v$, and right child $T_2$.
  - Make this new node the right child of $v$'s old parent.
  - Continue as if you had just inserted $k$.

Keep applying fixup rules to $k$. 

- Diagram: 
  - Red circle indicates the root.
  - Black node $v$ is found.
  - New node inserted with $k$ as key, $v$ as left child, and $T_2$ as right child.
  - Path from root to $v$ is now root-null.
Runtime Analysis

- Need to augment the red/black tree to store the black height of each node.
  - This fits into our augmentation framework – can be computed from the black heights of the left and right children and from the node's own color.
- Via the isometry with 2-3-4 trees, the runtime is $O(1 + |bh_1 - bh_2|)$.
- This is $O(\log n_1 + \log n_2)$ in the worst-case.
Joining Two Trees

• What if you want to join two red/black trees but don't have a key to join them with?

• Delete the minimum value from the second tree in time $O(\log n)$, then use that to join the two trees.
Implementing \textit{split} Efficiently
Splitting Trees is Hard

- **Challenge 1:** The split procedure might cut the existing tree into lots of smaller pieces.
Splitting Trees is Hard

- **Challenge 1:** The split procedure might cut the existing tree into lots of smaller pieces.

- **Challenge 2:** Cutting a red/black tree into two pieces doesn't necessarily give you two red/black trees.
An Observation

- Suppose we want to perform a split on some key $k$.
- Begin by searching for $k$. If we find it, search for its inorder successor.
- Cut all links found along the way.
An Observation

- Suppose we want to perform a split on some key $k$.
- Begin by searching for $k$. If we find it, search for its inorder successor.
- Cut all links found along the way.
An Observation

- Notice that we're left with a collection of pennants, trees whose roots have just one child.
An Observation

• Let's imagine uncoloring all of these pennant roots.

• The trees below them are *almost* red/black trees, but their roots might be red.

• Let's recolor all the roots black.
An Observation

- Let's imagine uncoloring all of these pennant roots.
- The trees below them are *almost* red/black trees, but their roots might be red.
- Let's recolor all the roots black.
An Observation

- We now have a bunch of red/black trees hanging off of pennants.
- **Key idea:** Find a way to *join* these trees back together to form the two trees we want.
Fleshing Out the Algorithm

- Do a search for the inorder successor of \( k \), cutting each link followed.

- For each pennant, color its child black. We now have a collection of red/black trees hanging off of random nodes.

- Categorize each hanging tree as of type \( L \) or type \( R \) depending on whether it's a left or right child of its pennant.
Observation 1: Look at any two consecutive $L$ trees or $R$ trees and the root of the pennant of the first tree. Then the key in the pennant root is strictly between all the values of those two trees.

Observation 2: There are at most two trees of each black height hanging off of the pennants.
Fleshing Out the Algorithm

All keys here are less than or equal to $k$.

All keys here are greater than $k$. 
Key idea: join all the L trees back together and all the R trees back together, using the nodes at the root of the pennants as the joining key. Because the height differences are low, the runtime works out to $O(\log n)$. 
Analyzing the Runtime

• Suppose there is one tree of each black height in $L$.
• What is the runtime of $join$ing the trees in reverse order of black heights?
• Each $join$ takes time $O(1 + |bh_1 - bh_2|) = O(1)$.
• At most $O(\log n)$ $join$s (the access path has length $O(\log n)$)
• Runtime is $O(\log n)$.
Analyzing the Runtime

- Suppose there are trees of very different black heights.
- What is the runtime of joining the trees in reverse order of black heights?
- Each join takes time $O(1 + bh_{s+1} - bh_s)$
- Summing across all joins:

$$\sum_{i=1}^{k-1} O(1+bh_{i+1}-bh_i) = O\left(\sum_{i=1}^{k-1} (1+bh_{i+1}-bh_i)\right)$$

$$= O\left(k+\sum_{i=1}^{k-1} (bh_{i+1}-bh_i)\right)$$

$$= O\left(k+bh_k-bh_1\right)$$

- The number of trees ($k$) is $O(\log n)$ and the maximum black height is $O(\log n)$. Runtime: $O(\log n)$. 
The Split Algorithm

• Split the tree into $L$ pennants and $R$ pennants, as before.

• Iterate across the pennants in ascending order of heights, *join* ing each of the corresponding trees together using the pennant node as the join key. This takes time $O(\log n)$.

• There will be $O(1)$ leftover pennant nodes. Insert them in time $O(\log n)$ into the proper trees.

• Net runtime: $O(\log n)$. 
The Big Summary
Thinking about balanced multiway search trees made it easier for us to think about balanced binary search trees.
Multiway trees, and *B-trees* in particular, are great choices for storing large data structures on disk.
Augmented BSTs let us solve challenging dynamic problems.
Red/black trees can be \textit{split} and \textit{joined} together using some clever algorithms.
Next Time

• **Amortized Analysis**
  • Lying about runtime costs in an honest manner.

• **Frameworks for Amortization**
  • How can we think about assigning costs?

• **Revisiting Earlier Structures**
  • Queues, Cartesian trees, and 2-3-4 trees.