Splay Trees
Recap from Last Time
### Table: Properties of Binary Search Trees

<table>
<thead>
<tr>
<th><strong>Property</strong></th>
<th><strong>Description</strong></th>
<th><strong>Met by</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Balance</strong></td>
<td>Lookups take time $O(\log n)$.</td>
<td>Traditional balanced BST</td>
</tr>
<tr>
<td><strong>Entropy</strong></td>
<td>Lookups take expected time $O(1 + H)$.</td>
<td>Weight-balanced trees</td>
</tr>
<tr>
<td><strong>Dynamic Finger</strong></td>
<td>Lookups take $O(\log \Delta)$.  [ \Delta \text{ measures distance.} ]</td>
<td>Level-linked BST with finger</td>
</tr>
<tr>
<td><strong>Working Set</strong></td>
<td>Lookups take $O(\log t)$, [ t \text{ measures recency.} ]</td>
<td>Iacono’s structure</td>
</tr>
</tbody>
</table>

What does an “optimal” binary search tree look like?
Consider a discrete probability distribution with elements $x_1, \ldots, x_n$, where element $x_i$ has access probability $p_i$.

The **Shannon entropy** of this probability distribution, denoted $H_p$ (or just $H$, where $p$ is implicit) is the quantity

$$H_p = \sum_{i=1}^{n} -p_i \log p_i.$$ 

**Theorem:** The expected cost of a lookup in *any* BST with keys $x_1, \ldots, x_n$ and access probabilities $p_1, \ldots, p_n$ is $\Omega(1 + H)$.

What does an “optimal” binary search tree look like?
What does an “optimal” binary search tree look like?
What does an “optimal” binary search tree look like?
What does an “optimal” binary search tree look like?
What does an “optimal” binary search tree look like?
What does an “optimal” binary search tree look like?
What does an “optimal” binary search tree look like?
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Met by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Balance</td>
<td>Lookups take time $O(\log n)$.</td>
<td>Traditional balanced BST</td>
</tr>
<tr>
<td>Entropy</td>
<td>Lookups take expected time $O(1 + H)$.</td>
<td>Weight-balanced trees</td>
</tr>
<tr>
<td>Dynamic Finger</td>
<td>Lookups take $O(\log \Delta)$. $\Delta$ measures distance.</td>
<td>Level-linked BST with finger</td>
</tr>
<tr>
<td>Working Set</td>
<td>Lookups take $O(\log t)$, $t$ measures recency.</td>
<td>Iacono’s structure</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
<td>Met by</td>
</tr>
<tr>
<td>---------------------</td>
<td>--------------------------------------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>Balance</td>
<td>Lookups take time $O(\log n)$.</td>
<td>Splay tree*</td>
</tr>
<tr>
<td>Entropy</td>
<td>Lookups take expected time $O(1 + H)$.</td>
<td>Splay tree*</td>
</tr>
<tr>
<td>Dynamic Finger</td>
<td>Lookups take $O(\log \Delta)$. $\Delta$ measures distance.</td>
<td>Splay tree*</td>
</tr>
<tr>
<td>Working Set</td>
<td>Lookups take $O(\log t)$, $t$ measures recency.</td>
<td>Splay tree*</td>
</tr>
</tbody>
</table>

Yes!
New Stuff!
**Idea 1:** Get the working set property by choosing a clever BST shape.

**Problem:** We can always pick a set of hot elements deep in the tree.

How do we build a BST with the working set property?
Idea 2: Get the working set property by adding a finger into our BST.

Problem: What if those keys aren’t near each other in key space?

How do we build a BST with the working set property?
**Idea 3:** Get the working set property by moving nodes around the BST.

**Strategy:** After querying a node, rotate it up to the root of the tree.

How do we build a BST with the working set property?
How do we build a BST with the working set property?

**Idea 3:** Get the working set property by moving nodes around the BST.

**Strategy:** After querying a node, rotate it up to the root of the tree.
How do we build a BST with the working set property? 

**Idea 3:** Get the working set property by moving nodes around the BST.

**Strategy:** After querying a node, rotate it up to the root of the tree.
How do we build a BST with the working set property?

**Idea 3:** Get the working set property by moving nodes around the BST.

**Strategy:** After querying a node, rotate it up to the root of the tree.
Idea 3: Get the working set property by moving nodes around the BST.

Strategy: After querying a node, rotate it up to the root of the tree.

How do we build a BST with the working set property?
How do we build a BST with the working set property?

**Idea 3:** Get the working set property by moving nodes around the BST.

**Strategy:** After querying a node, rotate it up to the root of the tree.
How do we build a BST with the working set property?

**Idea 3:** Get the working set property by moving nodes around the BST.

**Strategy:** After querying a node, rotate it up to the root of the tree.
**Idea 3:** Get the working set property by moving nodes around the BST.

**Strategy:** After querying a node, rotate it up to the root of the tree.

How do we build a BST with the working set property?
Idea 3: Get the working set property by moving nodes around the BST.

Strategy: After querying a node, rotate it up to the root of the tree.

How do we build a BST with the working set property?
How do we build a BST with the working set property?

**Idea 3:** Get the working set property by moving nodes around the BST.

**Strategy:** After querying a node, rotate it up to the root of the tree.
**Idea 3:** Get the working set property by moving nodes around the BST.

**Strategy:** After querying a node, rotate it up to the root of the tree.

How do we build a BST with the working set property?
**Idea 3:** Get the working set property by moving nodes around the BST.

**Strategy:** After querying a node, rotate it up to the root of the tree.

How do we build a BST with the working set property?
How do we build a BST with the working set property?

**Idea 3:** Get the working set property by moving nodes around the BST.

**Strategy:** After querying a node, rotate it up to the root of the tree.
Idea 3: Get the working set property by moving nodes around the BST.

Strategy: After querying a node, rotate it up to the root of the tree.

How do we build a BST with the working set property?
Idea 3: Get the working set property by moving nodes around the BST.

Strategy: After querying a node, rotate it up to the root of the tree.

How do we build a BST with the working set property?
How do we build a BST with the working set property?

**Idea 3:** Get the working set property by moving nodes around the BST.

**Strategy:** After querying a node, rotate it up to the root of the tree.
How do we build a BST with the working set property?

Idea 3: Get the working set property by moving nodes around the BST.

Strategy: After querying a node, rotate it up to the root of the tree.
**Idea 3:** Get the working set property by moving nodes around the BST.

**Strategy:** After querying a node, rotate it up to the root of the tree.

How do we build a BST with the working set property?
How do we build a BST with the working set property?

**Idea 3:** Get the working set property by moving nodes around the BST.

**Strategy:** After querying a node, rotate it up to the root of the tree.
How do we build a BST with the working set property?

**Idea 3:** Get the working set property by moving nodes around the BST.

**Strategy:** After querying a node, rotate it up to the root of the tree.
How do we build a BST with the working set property?

**Idea 3:** Get the working set property by moving nodes around the BST.

**Strategy:** After querying a node, rotate it up to the root of the tree.
How do we build a BST with the working set property?

Idea 3: Get the working set property by moving nodes around the BST.

Strategy: After querying a node, rotate it up to the root of the tree.
How do we build a BST with the working set property?

**Idea 3:** Get the working set property by moving nodes around the BST.

**Strategy:** After querying a node, rotate it up to the root of the tree.
We have a mechanical description of how we reshape the tree. Can we get an operational description?

**Question:** Does rotating each accessed key to the root guarantee good overall performance?
We have a *mechanical* description of how we reshape the tree. Can we get an *operational* description?

**Question:** Does rotating each accessed key to the root guarantee good overall performance?
Question: Does rotating each accessed key to the root guarantee good overall performance?
Question: Does rotating each accessed key to the root guarantee good overall performance?
We have a mechanical description of how we reshape the tree. Can we get an operational description?

**Question:** Does rotating each accessed key to the root guarantee good overall performance?
Question: Does rotating each accessed key to the root guarantee good overall performance?
We have a mechanical description of how we reshape the tree. Can we get an operational description?

**Question:** Does rotating each accessed key to the root guarantee good overall performance?
We have a *mechanical* description of how we reshape the tree. Can we get an *operational* description?

**Question:** Does rotating each accessed key to the root guarantee good overall performance?
We have a *mechanical* description of how we reshape the tree. Can we get an *operational* description?

**Question:** Does rotating each accessed key to the root guarantee good overall performance?
Does rotating each accessed key to the root guarantee good overall performance?

Question: Does rotating each accessed key to the root guarantee good overall performance?
We have a *mechanical* description of how we reshape the tree. Can we get an *operational* description?

**Question:** Does rotating each accessed key to the root guarantee good overall performance?
We have a mechanical description of how we reshape the tree. Can we get an operational description?

Group $R$-type and $L$-type nodes into two chains. Join them together using the rotated node.

**Question:** Does rotating each accessed key to the root guarantee good overall performance?
**Observation 1:** This works really well on zig-zag-shaped trees.

**Question:** Does rotating each accessed key to the root guarantee good overall performance?
**Question:** Does rotating each accessed key to the root guarantee good overall performance?

**Observation 1:** This works really well on zig-zag-shaped trees.

Group R-type and L-type nodes into two chains. Join them together using the rotated node.
**Observation 1:** This works really well on zig-zag-shaped trees.

Group $R$-type and $L$-type nodes into two chains. Join them together using the rotated node.

**Question:** Does rotating each accessed key to the root guarantee good overall performance?
Observation 1: This works really well on zig-zag-shaped trees.

Group R-type and L-type nodes into two chains. Join them together using the rotated node.

This tree is about half as tall as it started. Most nodes on the access path are much closer to the root.

Question: Does rotating each accessed key to the root guarantee good overall performance?
Observation 2: This does not work well at all on long chains.

Question: Does rotating each accessed key to the root guarantee good overall performance?
Observation 2: This does not work well at all on long chains.

Question: Does rotating each accessed key to the root guarantee good overall performance?
**Observation 2:** This does not work well at all on long chains.

Group $R$-type and $L$-type nodes into two chains. Join them together using the rotated node.

**Question:** Does rotating each accessed key to the root guarantee good overall performance?
**Observation 2:** This does not work well at all on long chains.

Group $R$-type and $L$-type nodes into two chains. Join them together using the rotated node.

**Question:** Does rotating each accessed key to the root guarantee good overall performance?
**Observation 2:** This does not work well at all on long chains.

Group \(R\)-type and \(L\)-type nodes into two chains.

Join them together using the rotated node.

**Question:** Does rotating each accessed key to the root guarantee good overall performance?
Observation 2: This does not work well at all on long chains.

Question: Does rotating each accessed key to the root guarantee good overall performance?
Observation 2: This does not work well at all on long chains.

Group $R$-type and $L$-type nodes into two chains. Join them together using the rotated node.

Question: Does rotating each accessed key to the root guarantee good overall performance?
Observation 2: This does not work well at all on long chains.

Question: Does rotating each accessed key to the root guarantee good overall performance?
**Observation 2:** This does not work well at all on long chains.

Group $R$-type and $L$-type nodes into two chains. Join them together using the rotated node.

**Question:** Does rotating each accessed key to the root guarantee good overall performance?
**Observation 2:** This does not work well at all on long chains.

**Group** $R$-type and $L$-type nodes into two chains. Join them together using the rotated node.

**Question:** Does rotating each accessed key to the root guarantee good overall performance?
**Observation 2:** This does not work well at all on long chains.

Group \( R \)-type and \( L \)-type nodes into two chains.

Join them together using the rotated node.

**Question:** Does rotating each accessed key to the root guarantee good overall performance?
Question: Does rotating each accessed key to the root guarantee good overall performance?

Observation 2: This does not work well at all on long chains.

Group R-type and L-type nodes into two chains. Join them together using the rotated node.
**Observation 2:** This does not work well at all on long chains.

Group $R$-type and $L$-type nodes into two chains. Join them together using the rotated node.

**Question:** Does rotating each accessed key to the root guarantee good overall performance?
Observation 2: This does not work well at all on long chains.

**Question:** Does rotating each accessed key to the root guarantee good overall performance?
**Question:** Does rotating each accessed key to the root guarantee good overall performance?

**Observation 2:** This does not work well at all on long chains.

Group $R$-type and $L$-type nodes into two chains. Join them together using the rotated node.
**Observation 2:** This does not work well at all on long chains.

Group $R$-type and $L$-type nodes into two chains. Join them together using the rotated node.

**Question:** Does rotating each accessed key to the root guarantee good overall performance?
**Observation 2:** This does not work well at all on long chains.

Group \( R \)-type and \( L \)-type nodes into two chains.
Join them together using the rotated node.

**Question:** Does rotating each accessed key to the root guarantee good overall performance?
**Observation 2:** This does not work well at all on long chains.

**Question:** Does rotating each accessed key to the root guarantee good overall performance?
**Observation 2:** This does not work well at all on long chains.

Group $R$-type and $L$-type nodes into two chains. Join them together using the rotated node.

**Question:** Does rotating each accessed key to the root guarantee good overall performance?
**Observation 2:** This does not work well at all on long chains.

Group \( R \)-type and \( L \)-type nodes into two chains. Join them together using the rotated node.

**Question:** Does rotating each accessed key to the root guarantee good overall performance?
Observation 2: This does not work well at all on long chains.

Group $R$-type and $L$-type nodes into two chains. Join them together using the rotated node.

Question: Does rotating each accessed key to the root guarantee good overall performance?
Observation 2: This does not work well at all on long chains.

Question: Does rotating each accessed key to the root guarantee good overall performance?
**Observation 2:** This does not work well at all on long chains.

Group $R$-type and $L$-type nodes into two chains. Join them together using the rotated node.

**Question:** Does rotating each accessed key to the root guarantee good overall performance?
**Observation 2:** This does not work well at all on long chains.

Group $R$-type and $L$-type nodes into two chains. Join them together using the rotated node.

**Question:** Does rotating each accessed key to the root guarantee good overall performance?
**Question:** Does rotating each accessed key to the root guarantee good overall performance?

**Observation 2:** This does not work well at all on long chains.

Group $R$-type and $L$-type nodes into two chains.
Join them together using the rotated node.

We’re right back where we started!
Total rotations: $\Theta(n^2)$. 
**Question:** How do we fix rotate-to-root to work well with long chains of nodes?

**Intuition:** We already handle zig-zags well. Let’s just fix the linear case.
**Intuition:** We already handle zig-zags well. Let’s just fix the linear case.

**Question:** How do we fix rotate-to-root to work well with long chains of nodes?
**Intuition:** We already handle zig-zags well. Let’s just fix the linear case.

**Question:** How do we fix rotate-to-root to work well with long chains of nodes?
**Intuition:** We already handle zig-zags well. Let’s just fix the linear case.

**Question:** How do we fix rotate-to-root to work well with long chains of nodes?
**Intuition:** We already handle zig-zags well. Let’s just fix the linear case.

**Question:** How do we fix rotate-to-root to work well with long chains of nodes?
**Intuition:** We already handle zig-zags well. Let’s just fix the linear case.

**Question:** How do we fix rotate-to-root to work well with long chains of nodes?
**Intuition:** We already handle zig-zags well. Let’s just fix the linear case.

**Question:** How do we fix rotate-to-root to work well with long chains of nodes?
**Intuition:** We already handle zig-zags well. Let’s just fix the linear case.

**Question:** How do we fix rotate-to-root to work well with long chains of nodes?
**Intuition:** We already handle zig-zags well. Let’s just fix the linear case.

**Question:** How do we fix rotate-to-root to work well with long chains of nodes?
**Intuition:** We already handle zig-zags well. Let’s just fix the linear case.

**Question:** How do we fix rotate-to-root to work well with long chains of nodes?
**Intuition:** We already handle zig-zags well. Let’s just fix the linear case.

**Question:** How do we fix rotate-to-root to work well with long chains of nodes?
**Intuition:** We already handle zig-zags well. Let’s just fix the linear case.

**Question:** How do we fix rotate-to-root to work well with long chains of nodes?
**Intuition:** We already handle zig-zags well. Let’s just fix the linear case.

**Question:** How do we fix rotate-to-root to work well with long chains of nodes?
**Intuition:** We already handle zig-zags well. Let’s just fix the linear case.

**Question:** How do we fix rotate-to-root to work well with long chains of nodes?
**Intuition:** We already handle zig-zags well. Let’s just fix the linear case.

**Question:** How do we fix rotate-to-root to work well with long chains of nodes?
**Question:** How do we fix rotate-to-root to work well with long chains of nodes?

**Intuition:** We already handle zig-zags well. Let’s just fix the linear case.
**Question:** How do we fix rotate-to-root to work well with long chains of nodes?

**Intuition:** We already handle zig-zags well. Let's just fix the linear case.
**Intuition:** We already handle zig-zags well. Let’s just fix the linear case.

**Question:** How do we fix rotate-to-root to work well with long chains of nodes?
**Question:** How do we fix rotate-to-root to work well with long chains of nodes?

**Intuition:** We already handle zig-zags well. Let’s just fix the linear case.
**Intuition:** We already handle zig-zags well. Let’s just fix the linear case.

**Observation:** This new rule roughly cuts the height of the access path in half.

**Question:** How do we fix rotate-to-root to work well with long chains of nodes?
This procedure for moving a node to the root of the tree is called **splaying**.

**Intuition:** Use rotate-to-root, except when nodes chain in the same direction.

**Mechanics:** Look back two steps in the tree and apply the appropriate rotation rules.

---

**Question:** How do we fix rotate-to-root to work well with long chains of nodes?
A splay tree is a regular BST where we splay the last node touched after each operation.

**Theorem:** The amortized cost of splaying a node is $O(\log n)$.

**Claim:** Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

**Insert:** Add as usual, then splay the new node.

Splaying dramatically simplifies BST operations.
A **splay tree** is a regular BST where we splay the last node touched after each operation.

**Theorem:** The amortized cost of splaying a node is $O(\log n)$.

**Claim:** Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

Splaying dramatically simplifies BST operations.
A *splay tree* is a regular BST where we splay the last node touched after each operation.

**Theorem:** The amortized cost of splaying a node is $O(\log n)$.

**Claim:** Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

Splaying dramatically simplifies BST operations.

**Insert:** Add as usual, then splay the new node.
A **splay tree** is a regular BST where we splay the last node touched after each operation.

**Theorem:** The amortized cost of splaying a node is $O(\log n)$.

**Claim:** Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

Splaying dramatically simplifies BST operations.
A splay tree is a regular BST where we splay the last node touched after each operation.

**Theorem:** The amortized cost of splaying a node is $O(\log n)$.

**Claim:** Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

Insert: Add as usual, then splay the new node.

Splaying dramatically simplifies BST operations.
A **splay tree** is a regular BST where we splay the last node touched after each operation.

**Theorem:** The amortized cost of splaying a node is $O(\log n)$.

**Claim:** Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

Splaying dramatically simplifies BST operations.
A *splay tree* is a regular BST where we splay the last node touched after each operation.

**Theorem:** The amortized cost of splaying a node is $O(\log n)$.

**Claim:** Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

*Splaying* dramatically simplifies BST operations.

**Insert:** Add as usual, then splay the new node.
A **splay tree** is a regular BST where we splay the last node touched after each operation.

**Theorem:** The amortized cost of splaying a node is $O(\log n)$.

**Claim:** Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

**Insert:** Add as usual, then splay the new node.

Splaying dramatically simplifies BST operations.
A **splay tree** is a regular BST where we splay the last node touched after each operation.

**Theorem:** The amortized cost of splaying a node is $O(\log n)$.

**Claim:** Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

Splaying dramatically simplifies BST operations.
A **splay tree** is a regular BST where we splay the last node touched after each operation.

**Theorem:** The amortized cost of splaying a node is $O(\log n)$.

**Claim:** Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

Splaying dramatically simplifies BST operations.
A **splay tree** is a regular BST where we splay the last node touched after each operation.

**Theorem:** The amortized cost of splaying a node is $O(\log n)$.

**Claim:** Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

**Insert:** Add as usual, then splay the new node.

Splaying dramatically simplifies BST operations.
A **splay tree** is a regular BST where we splay the last node touched after each operation.

**Theorem:** The amortized cost of splaying a node is $O(\log n)$.

**Claim:** Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

**Lookup:** Search, then splay the last node seen.

Splaying dramatically simplifies BST operations.
A *splay tree* is a regular BST where we splay the last node touched after each operation.

**Theorem:** The amortized cost of splaying a node is $O(\log n)$.

**Claim:** Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

**Lookup:** Search, then splay the last node seen.

Splaying dramatically simplifies BST operations.
A **splay tree** is a regular BST where we splay the last node touched after each operation.

**Theorem:** The amortized cost of splaying a node is $O(\log n)$.

**Claim:** Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

**Lookup:** Search, then splay the last node seen.

Splaying dramatically simplifies BST operations.
A **splay tree** is a regular BST where we splay the last node touched after each operation.

**Theorem:** The amortized cost of splaying a node is \(O(\log n)\).

**Claim:** Every splay tree operation cost is bounded by \(O(1)\) splays and takes amortized time \(O(\log n)\).

Lookup: Search, then splay the last node seen.

Splaying dramatically simplifies BST operations.
A **splay tree** is a regular BST where we splay the last node touched after each operation.

**Theorem:** The amortized cost of splaying a node is $O(\log n)$.

**Claim:** Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

**Lookup:** Search, then splay the last node seen.

Splaying dramatically simplifies BST operations.
A **splay tree** is a regular BST where we splay the last node touched after each operation.

**Theorem:** The amortized cost of splaying a node is $O(\log n)$.

**Claim:** Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

**Lookup:** Search, then splay the last node seen.

Splaying dramatically simplifies BST operations.
A **splay tree** is a regular BST where we splay the last node touched after each operation.

**Theorem:** The amortized cost of splaying a node is $O(\log n)$.

**Claim:** Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

**Lookup:** Search, then splay the last node seen.

Splaying dramatically simplifies BST operations.
A splay tree is a regular BST where we splay the last node touched after each operation.

**Theorem:** The amortized cost of splaying a node is $O(\log n)$.

**Claim:** Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

Splaying dramatically simplifies BST operations.
A splay tree is a regular BST where we splay the last node touched after each operation.

**Theorem:** The amortized cost of splaying a node is $O(\log n)$.

**Claim:** Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

**Split:** Search for the smallest value bigger than the split point. Splay it to the root and cut one link.

Splaying dramatically simplifies BST operations.
A **splay tree** is a regular BST where we splay the last node touched after each operation.

**Theorem:** The amortized cost of splaying a node is $O(\log n)$.

**Claim:** Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

**Split:** Search for the smallest value bigger than the split point. Splay it to the root and cut one link.

Splaying dramatically simplifies BST operations.
A splay tree is a regular BST where we splay the last node touched after each operation.

**Theorem:** The amortized cost of splaying a node is $O(\log n)$.

**Claim:** Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

**Split:** Search for the smallest value bigger than the split point. Splay it to the root and cut one link.

Splaying dramatically simplifies BST operations.
A splay tree is a regular BST where we splay the last node touched after each operation.

**Theorem:** The amortized cost of splaying a node is $O(\log n)$.

**Claim:** Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

**Split:** Search for the smallest value bigger than the split point. Splay it to the root and cut one link.

Splaying dramatically simplifies BST operations.
A splay tree is a regular BST where we splay the last node touched after each operation.

**Theorem:** The amortized cost of splaying a node is $O(\log n)$.

**Claim:** Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

**Join:** Splay the largest value in the left tree to the root, then add the right tree as its right child.

Splaying dramatically simplifies BST operations.
A **splay tree** is a regular BST where we splay the last node touched after each operation.

**Theorem:** The amortized cost of splaying a node is $O(\log n)$.

**Claim:** Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

**Join:** Splay the largest value in the left tree to the root, then add the right tree as its right child.

Splaying dramatically simplifies BST operations.
A **splay tree** is a regular BST where we splay the last node touched after each operation.

**Theorem:** The amortized cost of splaying a node is $O(\log n)$.

**Claim:** Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

**Join:** Splay the largest value in the left tree to the root, then add the right tree as its right child.

Splaying dramatically simplifies BST operations.
A splay tree is a regular BST where we splay the last node touched after each operation.

**Theorem:** The amortized cost of splaying a node is $O(\log n)$.

**Claim:** Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

**Join:** Splay the largest value in the left tree to the root, then add the right tree as its right child.

Splaying dramatically simplifies BST operations.
A splay tree is a regular BST where we splay the last node touched after each operation.

**Theorem:** The amortized cost of splaying a node is $O(\log n)$.

**Claim:** Every splay tree operation cost is bounded by $O(1)$ splays and takes amortized time $O(\log n)$.

**Join:** Splay the largest value in the left tree to the root, then add the right tree as its right child.

Splaying dramatically simplifies BST operations.
Question: Why is splaying fast?
Let $s(x)$ denote the number of nodes in the subtree rooted at $x$. 

**Question:** Why is splaying fast?
Let \( s(x) \) denote the number of nodes in the subtree rooted at \( x \).

**Question:** Why is splaying fast?
Let $s(x)$ denote the number of nodes in the subtree rooted at $x$.

Mark each edge as blue or red:
- $s(\text{child}) \leq \frac{1}{2} \cdot s(\text{parent})$
- $s(\text{child}) > \frac{1}{2} \cdot s(\text{parent})$

Blue edges make lots of progress.

**Question:** Why is splaying fast?
Let $s(x)$ denote the number of nodes in the subtree rooted at $x$.

Mark each edge as blue or red:
- $s(child) \leq \frac{1}{2} \cdot s(parent)$
- $s(child) > \frac{1}{2} \cdot s(parent)$

Blue edges make lots of progress.

**Question:** Why is splaying fast?
Why is splaying fast?

Let $s(x)$ denote the number of nodes in the subtree rooted at $x$.

Mark each edge as blue or red:
- $s(\text{child}) \leq \frac{1}{2} \cdot s(\text{parent})$
- $s(\text{child}) > \frac{1}{2} \cdot s(\text{parent})$

Blue edges make lots of progress.

Cost of visiting a node:
$O(\#\text{blue-used} + \#\text{red-used})$

Question: Why is splaying fast?
Let $s(x)$ denote the number of nodes in the subtree rooted at $x$.\\
Mark each edge as blue or red:
- Blue edges: $s(child) \leq \frac{1}{2} \cdot s(parent)$
- Red edges: $s(child) > \frac{1}{2} \cdot s(parent)$

Blue edges make lots of progress.

Cost of visiting a node:
$$O(#blue-used + #red-used)$$

**Idea:** Bound the cost of blue edges, then amortize away the cost of red edges. This is called a heavy/light decomposition.

**Question:** Why is splaying fast?
Let \( s(x) \) denote the number of nodes in the subtree rooted at \( x \).

Mark each edge as blue or red:
- Blue edges: \( s(\text{child}) \leq \frac{1}{2} \cdot s(\text{parent}) \)
- Red edges: \( s(\text{child}) > \frac{1}{2} \cdot s(\text{parent}) \)

Blue edges make lots of progress.

Cost of visiting a node:
\[ O(#\text{blue-used} + #\text{red-used}) \]

**Question:** Why is splaying fast?
Let $s(x)$ denote the number of nodes in the subtree rooted at $x$.

Mark each edge as blue or red:

- Blue edges:
  - $s(\text{child}) \leq \frac{1}{2} \cdot s(\text{parent})$
  
- Red edges:
  - $s(\text{child}) > \frac{1}{2} \cdot s(\text{parent})$

Blue edges make lots of progress.

Cost of visiting a node:

$O(\log n + \#\text{red-used})$

**Intuition:** Blue edges discard half the remaining nodes. You can only do that $O(\log n)$ times before running out of nodes.

**Question:** Why is splaying fast?
Let $s(x)$ denote the number of nodes in the subtree rooted at $x$.

Mark each edge as blue or red:
- $s(\text{child}) \leq \frac{1}{2} \cdot s(\text{parent})$
- $s(\text{child}) > \frac{1}{2} \cdot s(\text{parent})$

Blue edges make lots of progress.

Cost of visiting a node:
$O(\log n + \#\text{red-used})$

**Goal:** Find a potential function that penalizes red edges and rewards blue edges.

**Question:** Why is splaying fast?
Question: Why is splaying fast?

Let \( s(x) \) denote the number of nodes in the subtree rooted at \( x \).

Mark each edge as blue or red:
- \( \lg s(\text{child}) \leq \lg s(\text{parent}) - 1 \)
- \( \lg s(\text{child}) > \lg s(\text{parent}) - 1 \)

Blue edges make lots of progress.

Cost of visiting a node:
\[ O(\log n + \#\text{red-used}) \]

Goal: Find a potential function that penalizes red edges and rewards blue edges.
Question: Why is splaying fast?

Let $s(x)$ denote the number of nodes in the subtree rooted at $x$.

Mark each edge as blue or red:
- Blue edges make lots of progress.

\[
\text{Cost of visiting a node: } O(\log n + \#\text{red-used})
\]

Observation: If there are a lot of red edges, then $\lg s(x)$ will frequently be large.
Let $s(x)$ denote the number of nodes in the subtree rooted at $x$.

Mark each edge as blue or red:
- $\lg s(\text{child}) \leq \lg s(\text{parent}) - 1$
- $\lg s(\text{child}) > \lg s(\text{parent}) - 1$

Blue edges make lots of progress.

Cost of visiting a node:
$O(\log n + \#\text{red-used})$

Choose our potential to be
\[
\Phi = \sum_{i=1}^{n} \lg s(x_i).
\]

**Question:** Why is splaying fast?
Cost of visiting a node:
$O(\log n + \#\text{red-used})$

Choose our potential to be
$$\Phi = \sum_{i=1}^{n} \lg s(x_i).$$

Proving $\Phi$ amortizes away the $\#\text{red-used}$ term involves some detail-oriented math. Check the Sleator-Tarjan paper for details.

**Theorem:** The amortized cost of a splay operation is $O(\log n)$.

---

**Question:** Why is splaying fast?
Question: Why is splaying fast?

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Balance</strong></td>
<td>Lookups take time $O(\log n)$.</td>
</tr>
<tr>
<td><strong>Entropy</strong></td>
<td>Lookups take expected time $O(1 + H)$.</td>
</tr>
<tr>
<td><strong>Dynamic Finger</strong></td>
<td>Lookups take $O(\log \Delta)$. $\Delta$ measures distance.</td>
</tr>
<tr>
<td><strong>Working Set</strong></td>
<td>Lookups take $O(\log t)$, $t$ measures recency.</td>
</tr>
</tbody>
</table>
Some nodes are more important than others. Assign each a weight $w_i$ and let the total weight be $W$.

**Question:** Why is splaying fast?
Some nodes are more important than others. Assign each a weight $w_i$ and let the total weight be $W$.

Let $s(x_i)$ be the sum of the weights in the tree rooted at $x_i$.

**Question:** Why is splaying fast?
Some nodes are more important than others. Assign each a weight $w_i$ and let the total weight be $W$.

Let $s(x_i)$ be the sum of the weights in the tree rooted at $x_i$.

**Question:** Why is splaying fast?
Some nodes are more important than others. Assign each a weight $w_i$ and let the total weight be $W$.

Let $s(x_i)$ be the sum of the weights in the tree rooted at $x_i$.

Mark each edge as blue or red:

- Blue: $\lg s(\text{child}) \leq \lg s(\text{parent}) - 1$
- Red: $\lg s(\text{child}) > \lg s(\text{parent}) - 1$

**Question:** Why is splaying fast?
Some nodes are more important than others. Assign each a weight \( w_i \) and let the total weight be \( W \).

Let \( s(x_i) \) be the sum of the weights in the tree rooted at \( x_i \).

Mark each edge as blue or red:

- \( \lg s(\text{child}) \leq \lg s(\text{parent}) - 1 \)
- \( \lg s(\text{child}) > \lg s(\text{parent}) - 1 \)

Cost of visiting a node:

\[ O(\#\text{blue-used} + \#\text{red-used}) \]

**Question:** How do we bound \( \#\text{blue-used} \) in this case?

**Question:** Why is splaying fast?
Some nodes are more important than others. Assign each a weight $w_i$ and let the total weight be $W$.

Let $s(x_i)$ be the sum of the weights in the tree rooted at $x_i$.

Mark each edge as blue or red:
- $\lg s(\text{child}) \leq \lg s(\text{parent}) - 1$
- $\lg s(\text{child}) > \lg s(\text{parent}) - 1$

Cost of visiting a node:
$O(\log (W / w_i) + \text{#red-used})$

**Question:** Why is splaying fast?
Some nodes are more important than others. Assign each a weight \( w_i \) and let the total weight be \( W \).

Let \( s(x_i) \) be the sum of the weights in the tree rooted at \( x_i \).

Mark each edge as blue or red:
- \( \lg s(\text{child}) \leq \lg s(\text{parent}) - 1 \)
- \( \lg s(\text{child}) > \lg s(\text{parent}) - 1 \)

Cost of visiting a node:
\[
O(\log \left( \frac{W}{w_i} \right) + \#\text{red-used})
\]

Set \( \Phi = \sum_{i=1}^{n} \lg s(x_i) \).

**Question**: Why is splaying fast?
Theorem: Using the sum-of-logs potential, the amortized cost of splaying a node with weight $w_i$ is $O(\log (W / w_i))$, where $W$ is the sum of all the weights.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Balance</td>
<td>Lookups take time $O(\log n)$.</td>
</tr>
<tr>
<td>Entropy</td>
<td>Lookups take expected time $O(1 + H)$.</td>
</tr>
<tr>
<td>Dynamic Finger</td>
<td>Lookups take $O(\log \Delta)$. $\Delta$ measures distance.</td>
</tr>
<tr>
<td>Working Set</td>
<td>Lookups take $O(\log t)$, $t$ measures recency.</td>
</tr>
</tbody>
</table>
**Theorem:** Using the sum-of-logs potential, the amortized cost of splaying a node with weight $w_i$ is $O(\log (W / w_i))$, where $W$ is the sum of all the weights.

**Balance Property:** The cost of any lookup in the binary search tree is $O(\log n)$, where $n$ is the number of nodes.

Assign each node weight $\frac{1}{n}$.

$W = 1$

$w_i = \frac{1}{n}$

$W / w_i = n$

Amortized cost of a lookup:

$O(\log (W / w_i)) = O(\log n)$
**Entropy Property:** Expected cost of a lookup is $O(1 + H)$, assuming lookups are drawn from a fixed distribution.

$$H = \sum_{i=1}^{n} -p_i \log p_i.$$  

**Goal:** Make this the cost of looking up key $x_i$.

**Theorem:** Using the sum-of-logs potential, the amortized cost of splaying a node with weight $w_i$ is $O(\log (W / w_i))$, where $W$ is the sum of all the weights.
**Theorem:** Using the sum-of-logs potential, the amortized cost of splaying a node with weight $w_i$ is $O(\log (W / w_i))$, where $W$ is the sum of all the weights.

**Entropy Property:** Expected cost of a lookup is $O(1 + H)$, assuming lookups are drawn from a fixed distribution.

$$H = \sum_{i=1}^{n} -p_i \lg p_i.$$  

If $\log(W / w_i) = O(-\log p_i)$, then we have the entropy property.

Pick $w_i = p_i$.

$W = 1$.

$W / w_i = 1 / p_i$.

$O(\log (W / w_i)) = O(\log (1/p_i)) = O(-\log p_i)$. 

**Theorem:** Using the sum-of-logs potential, the amortized cost of splaying a node with weight $w_i$ is $O(\log (W / w_i))$, where $W$ is the sum of all the weights.
**Theorem:** Using the sum-of-logs potential, the amortized cost of splaying a node with weight $w_i$ is $O(\log (W / w_i))$, where $W$ is the sum of all the weights.

**Working Set Property:**
Lookups take time $O(\log t)$, where $t$ is the number of keys queried since the last time element was queried.

It doesn’t immediately seem like we can use the theorem below, since the value of $t$ depends on what accesses have been done recently.

For now, let’s set that aside and focus on one snapshot in time.

Each key $x_i$ is annotated with its value of $t_i$. How do we pick weights?
**Theorem:** Using the sum-of-logs potential, the amortized cost of splaying a node with weight \( w_i \) is \( O(\log (W / w_i)) \), where \( W \) is the sum of all the weights.

**Working Set Property:** Lookups take time \( O(\log t) \), where \( t \) is the number of keys queried since the last time an element was queried.

**Reasoning by analogy:**

**Balance:** Target is \( O(\log n) \).
   Picked \( w_i = \frac{1}{n} \).

**Entropy:** Target is \( O(\log (1/p_i)) \).
   Picked \( w_i = p_i \).

**Working Set:** Target is \( O(\log t_i) \).
   Pick \( w_i = 1 / t_i \).

**Question:** Does this work?
**Theorem:** Using the sum-of-logs potential, the amortized cost of splaying a node with weight $w_i$ is $O(\log (W / w_i))$, where $W$ is the sum of all the weights.

**Working Set Property:** Lookups take time $O(\log t)$, where $t$ is the number of keys queried since the last time an element was queried.

$w_i = \frac{1}{t_i}$

$W = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$

$= \Theta(\log n)$

This is the *nth harmonic number*, denoted $H_n$.

Useful fact:

$\ln (n+1) \leq H_n \leq (\ln n) + 1$. 

**Theorem:** Using the sum-of-logs potential, the amortized cost of splaying a node with weight $w_i$ is $O(\log (W / w_i))$, where $W$ is the sum of all the weights.
**Theorem:** Using the sum-of-logs potential, the amortized cost of splaying a node with weight $w_i$ is $O(\log (W / w_i))$, where $W$ is the sum of all the weights.

$$w_i = 1 / t_i$$

$$W = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} = \Theta(\log n)$$

$$W / w_i = \Theta(t_i \log n).$$

$$O(\log (t_i \log n)) = O(\log t_i + \log \log n).$$

Close! can we do better?
**Theorem:** Using the sum-of-logs potential, the amortized cost of splaying a node with weight $w_i$ is $O(\log (W / w_i))$, where $W$ is the sum of all the weights.

**Working Set Property:** Lookups take time $O(\log t)$, where $t$ is the number of keys queried since the last time the element was queried.

The sum of the weights is too large for $W / w_i$ to work out the way we want. Can we pick weights so that $W = O(1)$ and $\log (1 / w_i) = O(\log t_i)$?
**Working Set Property:**
Lookups take time $O(\log t)$, where $t$ is the number of keys queried since the last time element was queried.

$w_i = 1 / t_i^2$

$W = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \ldots + \frac{1}{n^2}$

$= O(1)$

**Useful fact:**

$$\sum_{i=1}^{\infty} \frac{1}{i^2} = \frac{\pi^2}{6}$$

**Theorem:** Using the sum-of-logs potential, the amortized cost of splaying a node with weight $w_i$ is $O(\log (W / w_i))$, where $W$ is the sum of all the weights.
Working Set Property:
Lookups take time $O(\log t)$, where $t$ is the number of keys queried since the last time element was queried.

$$w_i = 1 / t_i^2$$

$$W = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \ldots + \frac{1}{n^2}$$

$$= O(1)$$

$$W / w_i = O(1) \cdot t_i^2$$

$$O(\log (O(1) \cdot t_i^2))$$

$$= O(\log t_i).$$

But we’re not done just yet.

Theorem: Using the sum-of-logs potential, the amortized cost of splaying a node with weight $w_i$ is $O(\log (W / w_i))$, where $W$ is the sum of all the weights.
Theorem: Using the sum-of-logs potential, the amortized cost of splaying a node with weight $w_i$ is $O(\log (W / w_i))$, where $W$ is the sum of all the weights.

**Working Set Property:**
Lookups take time $O(\log t)$, where $t$ is the number of keys queried since the last time element was queried.

If we pick a fixed snapshot in time and assign each key weight $1/t_i^2$, then the amortized cost of a lookup, at that snapshot, is $O(\log t_i)$.

But after doing this, all the $t_i$ values change. What happens as a result?
**Theorem:** Using the sum-of-logs potential, the amortized cost of splaying a node with weight $w_i$ is $O(\log (W / w_i))$, where $W$ is the sum of all the weights.

**Working Set Property:**
Lookups take time $O(\log t)$, where $t$ is the number of keys queried since the last time an element was queried.

If we pick a fixed snapshot in time and assign each key weight $1/t_i^2$, then the amortized cost of a lookup, at that snapshot, is $O(\log t_i)$.

But after doing this, all the $t_i$ values change. What happens as a result?
**Theorem:** Using the sum-of-logs potential, the amortized cost of splaying a node with weight \( w_i \) is \( O(\log (W / w_i)) \), where \( W \) is the sum of all the weights.

**Working Set Property:**
Lookups take time \( O(\log t) \), where \( t \) is the number of keys queried since the last time element was queried.

If we pick a fixed snapshot in time and assign each key weight \( 1/t_i^2 \), then the amortized cost of a lookup, at that snapshot, is \( O(\log t_i) \).

But after doing this, all the \( t_i \) values change. What happens as a result?
**Working Set Property:**
Lookups take time $O(\log t)$, where $t$ is the number of keys queried since the last time element was queried.

If we pick a fixed snapshot in time and assign each key weight $1/t_i^2$, then the amortized cost of a lookup, at that snapshot, is $O(\log t_i)$.

But after doing this, all the $t_i$ values change. What happens as a result?

**Theorem:** Using the sum-of-logs potential, the amortized cost of splaying a node with weight $w_i$ is $O(\log (W / w_i))$, where $W$ is the sum of all the weights.
**Theorem:** Using the sum-of-logs potential, the amortized cost of splaying a node with weight $w_i$ is $O(\log (W / w_i))$, where $W$ is the sum of all the weights.

**Working Set Property:**
Lookups take time $O(\log t)$, where $t$ is the number of keys queried since the last time an element was queried.

If we pick a fixed snapshot in time and assign each key weight $1/t_i^2$, then the amortized cost of a lookup, at that snapshot, is $O(\log t_i)$.

But after doing this, all the $t_i$ values change. What happens as a result?
**Theorem:** Using the sum-of-logs potential, the amortized cost of splaying a node with weight $w_i$ is $O(\log (W / w_i))$, where $W$ is the sum of all the weights.

**Working Set Property:**
Lookups take time $O(\log t)$, where $t$ is the number of keys queried since the last time element was queried.

If we pick a fixed snapshot in time and assign each key weight $1/t_i^2$, then the amortized cost of a lookup, at that snapshot, is $O(\log t_i)$.

But after doing this, all the $t_i$ values change. What happens as a result?
**Theorem:** Using the sum-of-logs potential, the amortized cost of splaying a node with weight $w_i$ is $O(\log (W / w_i))$, where $W$ is the sum of all the weights.

**Working Set Property:** Lookups take time $O(\log t)$, where $t$ is the number of keys queried since the last time element was queried.

If we pick a fixed snapshot in time and assign each key weight $1/t_i^2$, then the amortized cost of a lookup, at that snapshot, is $O(\log t_i)$.

But after doing this, all the $t_i$ values change. What happens as a result?
**Theorem:** Using the sum-of-logs potential, the amortized cost of splaying a node with weight $w_i$ is $O(\log (W / w_i))$, where $W$ is the sum of all the weights.

**Working Set Property:**
Lookups take time $O(\log t)$, where $t$ is the number of keys queried since the last time an element was queried.

If we pick a fixed snapshot in time and assign each key weight $1/t_i^2$, then the amortized cost of a lookup, at that snapshot, is $O(\log t_i)$. But after doing this, all the $t_i$ values change. What happens as a result?
**Working Set Property:**
Lookups take time $O(\log t)$, where $t$ is the number of keys queried since the last time element was queried.

If we pick a fixed snapshot in time and assign each key weight $1/t_i^2$, then the amortized cost of a lookup, at that snapshot, is $O(\log t_i)$.

But after doing this, all the $t_i$ values change. What happens as a result?

**Theorem:** Using the sum-of-logs potential, the amortized cost of splaying a node with weight $w_i$ is $O(\log (W / w_i))$, where $W$ is the sum of all the weights.
**Theorem:** Using the sum-of-logs potential, the amortized cost of splaying a node with weight $w_i$ is $O(\log (W / w_i))$, where $W$ is the sum of all the weights.

**Working Set Property:** Lookups take time $O(\log t)$, where $t$ is the number of keys queried since the last time element was queried.

**Recall:** Each node’s size is the weight of its subtree.

$$\Phi = \sum_{i=1}^{n} \lg s(x_i).$$

Changing weights this way only decreases $s(x_i)$ for each node, so $\Phi$ drops after the splay.
**Theorem:** Using the sum-of-logs potential, the amortized cost of splaying a node with weight $w_i$ is $O(\log (W / w_i))$, where $W$ is the sum of all the weights.

**Working Set Property:** Lookups take time $O(\log t)$, where $t$ is the number of keys queried since the last time element was queried.

**Recall:** Each node’s size is the weight of its subtree.

$$\Phi = \sum_{i=1}^{n} \lg s(x_i).$$

Changing weights this way only decreases $s(x_i)$ for each node, so $\Phi$ drops after the splay.

Size of the root is still $\frac{1}{1^2} + \ldots + \ldots \frac{1}{n^2}$
**Theorem:** Using the sum-of-logs potential, the amortized cost of splaying a node with weight $w_i$ is $O(\log (W / w_i))$, where $W$ is the sum of all the weights.

**Working Set Property:**
Lookups take time $O(\log t)$, where $t$ is the number of keys queried since the last time element was queried.

**Recall:** Each node’s size is the weight of its subtree.

$$\Phi = \sum_{i=1}^{n} \lg s(x_i).$$

Changing weights this way only decreases $s(x_i)$ for each node, so $\Phi$ drops after the splay.
**Theorem:** Using the sum-of-logs potential, the amortized cost of splaying a node with weight $w_i$ is $O(\log \left( \frac{W}{w_i} \right))$, where $W$ is the sum of all the weights.

**Working Set Property:**
Lookups take time $O(\log t)$, where $t$ is the number of keys queried since the last time an element was queried.

**Recall:** Each node’s size is the weight of its subtree.

$$\Phi = \sum_{i=1}^{n} \lg s(x_i).$$

Changing weights this way only decreases $s(x_i)$ for each node, so $\Phi$ drops after the splay.

So the amortized cost of each operation is still $O(\log t_i)$, even in the dynamic case!

**Amortized Cost**

$$= \text{Real Cost} + \Delta \Phi$$

Decreasing $\Phi$ after the operation can only reduce the amortized cost.

**Theorem:** Using the sum-of-logs potential, the amortized cost of splaying a node with weight $w_i$ is $O(\log (W / w_i))$, where $W$ is the sum of all the weights.
**Theorem:** Using the sum-of-logs potential, the amortized cost of splaying a node with weight $w_i$ is $O(\log (W / w_i))$, where $W$ is the sum of all the weights.

**Dynamic Finger Property:** Lookups take time $O(\log \Delta)$, where $\Delta$ is the number of keys between the last key queried and the current key queried.

It doesn’t immediately seem like we can use the theorem below, since the value of $\Delta$ depends on what accesses have been done recently.

For now, let’s set that aside and focus on one snapshot in time.

Each key $x_i$ is annotated with its value $\Delta_i$, the rank difference to the last element. How do we pick weights?
**Theorem:** Using the sum-of-logs potential, the amortized cost of splaying a node with weight $w_i$ is $O(\log (W / w_i))$, where $W$ is the sum of all the weights.

**Dynamic Finger Property:** Lookups take time $O(\log \Delta)$, where $\Delta$ is the number of keys between the last key queried and the current key queried.

Pick $w_i = 1 / \Delta_i^2$

$$W \leq \frac{2}{1^2} + \frac{2}{2^2} + \frac{2}{3^2} + \ldots + \frac{2}{n^2}$$

$$= O(1)$$

There are at most two keys at distance $k$ from the finger, one in each direction.
**Theorem:** Using the sum-of-logs potential, the amortized cost of splaying a node with weight $w_i$ is $O(\log (W / w_i))$, where $W$ is the sum of all the weights.

**Dynamic Finger Property:**
Lookups take time $O(\log \Delta)$, where $\Delta$ is the number of keys between the last key queried and the current key queried.

Pick $w_i = 1 / \Delta_i^2$

$$W \leq \frac{2}{1^2} + \frac{2}{2^2} + \frac{2}{3^2} + \ldots + \frac{2}{n^2} = O(1)$$

$$W / w_i = O(1) \cdot \Delta_i^2$$

$$O(\log (O(1) \cdot \Delta_i^2)) = O(\log \Delta_i).$$

But we’re not done just yet.
**Dynamic Finger Property:**
Lookups take time $O(\log \Delta)$, where $\Delta$ is the number of keys between the last key queried and the current key queried.

If we pick a fixed snapshot in time and assign each key weight $1/\Delta_i^2$, then the amortized cost of a lookup, at that snapshot, is $O(\log \Delta_i)$.

But after doing this, all the $\Delta_i$ values change. What happens as a result?

**Theorem:** Using the sum-of-logs potential, the amortized cost of splaying a node with weight $w_i$ is $O(\log (W / w_i))$, where $W$ is the sum of all the weights.
**Theorem:** Using the sum-of-logs potential, the amortized cost of splaying a node with weight $w_i$ is $O(\log (W / w_i))$, where $W$ is the sum of all the weights.

**Dynamic Finger Property:**
Lookups take time $O(\log \Delta)$, where $\Delta$ is the number of keys between the last key queried and the current key queried.

If we pick a fixed snapshot in time and assign each key weight $1/\Delta^2$, then the amortized cost of a lookup, at that snapshot, is $O(\log \Delta_i)$.

But after doing this, all the $\Delta_i$ values change. What happens as a result?
**Dynamic Finger Property:**
Lookups take time $O(\log \Delta)$, where $\Delta$ is the number of keys between the last key queried and the current key queried.

If we pick a fixed snapshot in time and assign each key weight $1/\Delta_i^2$, then the amortized cost of a lookup, at that snapshot, is $O(\log \Delta_i)$.

But after doing this, all the $\Delta_i$ values change. What happens as a result?

**Theorem:** Using the sum-of-logs potential, the amortized cost of splaying a node with weight $w_i$ is $O(\log (W / w_i))$, where $W$ is the sum of all the weights.
Theorem: Using the sum-of-logs potential, the amortized cost of splaying a node with weight $w_i$ is $O(\log (W / w_i))$, where $W$ is the sum of all the weights.

Dynamic Finger Property: Lookups take time $O(\log \Delta)$, where $\Delta$ is the number of keys between the last key queried and the current key queried.

If we pick a fixed snapshot in time and assign each key weight $1/\Delta_i^2$, then the amortized cost of a lookup, at that snapshot, is $O(\log \Delta_i)$. But after doing this, all the $\Delta_i$ values change. What happens as a result?
**Theorem:** Using the sum-of-logs potential, the amortized cost of splaying a node with weight $w_i$ is $O(\log (W / w_i))$, where $W$ is the sum of all the weights.

**Dynamic Finger Property:** Lookups take time $O(\log \Delta)$, where $\Delta$ is the number of keys between the last key queried and the current key queried.

If we pick a fixed snapshot in time and assign each key weight $1/\Delta_i^2$, then the amortized cost of a lookup, at that snapshot, is $O(\log \Delta_i)$. But after doing this, all the $\Delta_i$ values change. What happens as a result?
**Theorem:** Using the sum-of-logs potential, the amortized cost of splaying a node with weight $w_i$ is $O(\log (W / w_i))$, where $W$ is the sum of all the weights.

**Dynamic Finger Property:** Lookups take time $O(\log \Delta)$, where $\Delta$ is the number of keys between the last key queried and the current key queried.

**Problem:** Unlike before, the sizes of subtrees can both grow and shrink after splaying. There isn’t a clear way to proceed.
**Theorem:** Using the sum-of-logs potential, the amortized cost of splaying a node with weight $w_i$ is $O(\log (W / w_i))$, where $W$ is the sum of all the weights.

**Dynamic Finger Property:**
Lookups take time $O(\log \Delta)$, where $\Delta$ is the number of keys between the last key queried and the current key queried.

**Problem:** Unlike before, the sizes of subtrees can both grow and shrink after splaying. There isn’t a clear way to proceed.
Theorem: Using the sum-of-logs potential, the amortized cost of splaying a node with weight $w_i$ is $O(\log (W / w_i))$, where $W$ is the sum of all the weights.

Dynamic Finger Property: Lookups take time $O(\log \Delta)$, where $\Delta$ is the number of keys between the last key queried and the current key queried.

Problem: Unlike before, the sizes of subtrees can both grow and shrink after splaying. There isn’t a clear way to proceed.

However, we did just prove the static finger property: if you fix some key in advance and let $\delta_i$ be the number of keys between $x_i$ and that key, then lookups take time $O(\log \delta_i)$.
**Dynamic Finger Property:** Lookups take time $O(\log \Delta)$, where $\Delta$ is the number of keys between the last key queried and the current key queried.

**Theorem:** Splay trees have the dynamic finger property.


**Open Problem:** Find a simpler proof that splay trees have the dynamic finger property.

**Theorem:** Using the sum-of-logs potential, the amortized cost of splaying a node with weight $w_i$ is $O(\log (W / w_i))$, where $W$ is the sum of all the weights.
Just how fast are splay trees?

Is all the creativity that goes into each of these structures captured by a single, simple binary search tree?
Pick any (long) sequence of operations. Pick any BST $T$, including one that, like a splay tree, is allowed to reshape itself.

**Dynamic Optimality Conjecture:**
Cost of performing those operations on a splay tree

\[
\leq \quad O(1) \cdot \text{Cost of performing those operations on } T
\]

Stated differently: no matter how clever you are with your BST design, you will never be able to beat a splay tree by more than a constant factor.

*This is an open problem! And it’s a big one!*

Just how fast *are* splay trees?
So... if splay trees are so great, why aren’t we using them everywhere instead of other tree structures?

1. Amortized versus worst-case bounds are not always acceptable in practice.

2. Poor support for concurrency, especially in lookup-heavy loads.

3. Slightly higher constant factors than some other trees, due to the number of memory writes per operation.

Many of drawbacks can be mitigated in practice, and we do see splay trees used fairly extensively in practice alongside red/black and B-trees.

**Excellent Idea:** Code up splay trees and measure their performance!

Just how fast *are* splay trees?
• Worst-case efficiency (the *balance property*) isn’t the only metric we can use to measure BST performance.

• Specialized data structures like weight-balanced trees, level-linked finger search trees, and Iacono’s structure can be designed to meet these bounds.

• For a BST to have all these properties at once, it needs to be able to move nodes around.

• Rotate-to-root is a plausible but inefficient mechanism for reordering nodes.

• Splaying corrects for rotate-to-root by handling linear chains more intelligently.

• Splaying provides simple implementations of all common BST operations.

• By using a heavy/light decomposition, we can isolate the effects of poor tree shapes.

• Using a sum-of-logs potential allows us to amortize away heavy edges.

• Splay trees have the balance property, entropy property, dynamic finger property, and working set property.

• It’s an open problem in data structure theory whether it’s possible to improve upon splay trees in an amortized sense.

To Summarize…
Next Time

- **Integer Data Structures**
  - Harnessing the intrinsic parallelism of processor instructions.

- **x-Fast Tries**
  - Tries + Cuckoo Hashing

- **y-Fast Tries**
  - Macro/Micro Decompositions + Splay Trees + Amortized Analysis