Better than Balanced BSTs
Outline for Today

- **Beyond Worst-Case Efficiency**
  - When $O(\log n)$ isn’t enough.
- **Weight-Balanced Trees**
  - Balancing by access probabilities.
- **Finger Search Trees**
  - Picking up where you left off.
- **Iacono’s Working Set Structure**
  - Keeping exciting things accessible.
Can you build a binary search tree where lookups are faster than $O(\log n)$?
**Key Idea:** The guarantees we want from a data structure depend on our model of how that data structure will be used.
Claim: The worst-case lookup cost of a lookup in any BST with \( n \) nodes is at least \( \Omega(\log n) \).

Proof Idea: Every tree with \( n \) nodes has height \( \Omega(\log n) \). Pick the deepest node in the tree.

Model 1: Queries are chosen maliciously.
A binary search tree satisfies the **balance property** if the (amortized) cost of any lookup in that tree is $O(\log n)$.

Any BST with this property is optimal from a *worst-case* perspective.

“Classical” balanced trees (red/black, etc.) are designed to have this property.

**Model 1:** Queries are chosen maliciously.
Model 2: Queries are sampled from a fixed, known probability distribution.

Access Probabilities

Expected comparisons in a lookup: **1.83**

1  2  3  4  5  6  7
20% 10% 40% 8% 1% 1% 20%

Expected comparisons in a lookup: **2.73**

Model 2: Queries are sampled from a fixed, known probability distribution.
Model 2: Queries are sampled from a fixed, known probability distribution.

How do we know when we have a BST that’s optimal with respect to expected lookup costs?
**Model 2:** Queries are sampled from a fixed, known probability distribution.

**Intuition:** Place high probability elements high in the tree.

Keys with access probability $\frac{1}{2}$ or higher probably shouldn't go below here.

Keys with access probability $\frac{1}{16}$ or higher probably shouldn't be below here.
**Intuition:** Place high probability elements high in the tree.

Ideally, a key goes in layer $k$ or above if its access probability is at least $2^{-k}$.

Equivalently, a key with access probability $p$ would ideally be in layer $\lg (\frac{1}{p})$ or higher.

Note that $\lg (\frac{1}{p}) = -\lg p$.

Expected cost of a lookup would then be

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

**Model 2:** Queries are sampled from a fixed, known probability distribution.
Model 2: Queries are sampled from a fixed, known probability distribution.
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

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

If all elements have equal access probability ($p_i = \frac{1}{n}$):

$$H_p = \sum_{i=1}^{n} -p_i \log p_i = \sum_{i=1}^{n} \frac{1}{n} \left( -\log \frac{1}{n} \right) = \sum_{i=1}^{n} \frac{1}{n} \log n = \log n.$$

If only one element is ever accessed ($p_1 = 1, p_i = 0$), then

$$H_p = \sum_{i=1}^{n} -p_i \log p_i = -\log 1 + \sum_{i=2}^{n} 0 \log 0 = 0$$

**Model 2:** Queries are sampled from a fixed, known probability distribution.
**Model 2:** Queries are sampled from a fixed, known probability distribution.

**Theorem:** If accesses are sampled over a discrete distribution, then the expected cost of a lookup in any BST is $\Omega(1 + H)$, where $H$ is the Shannon entropy of the distribution.

A binary search tree has the **entropy property** if the (amortized) expected cost of any lookup on that BST is $O(1 + H)$.

(Any BST with this property is optimal from a expected-case perspective, assuming a fixed probability distribution.)
**Question:** Assuming you know the access probabilities, how could you build a BST with the entropy property?

**Idea 1:** Pick the root to be the highest-probability element. Then, recursively build the subtrees.

**Idea 2:** Pick the root to balance the probabilities of the smaller elements and the bigger elements. Then, recursively build the subtrees.

**Model 2:** Queries are sampled from a fixed, known probability distribution.
**Question:** Assuming you know the access probabilities, how could you build a BST with the entropy property?

**Idea 1:** Pick the root to be the highest-probability element. Then, recursively build the subtrees.

**Idea 2:** Pick the root to balance the probabilities of the smaller elements and the bigger elements. Then, recursively build the subtrees.

**Model 2:** Queries are sampled from a fixed, known probability distribution.
**Question:** Assuming you know the access probabilities, how could you build a BST with the entropy property?

**Idea 1:** Pick the root to be the highest-probability element. Then, recursively build the subtrees.

**Idea 2:** Pick the root to balance the probabilities of the smaller elements and the bigger elements. Then, recursively build the subtrees.

The entropy here is roughly $\lg n$, but this tree has height $\Theta(n)$.

**Model 2:** Queries are sampled from a fixed, known probability distribution.
**Question:** Assuming you know the access probabilities, how could you build a BST with the entropy property?

**Idea 1:** Pick the root to be the highest-probability element. Then, recursively build the subtrees.

**Idea 2:** Pick the root to balance the probabilities of the smaller elements and the bigger elements. Then, recursively build the subtrees.

**Model 2:** Queries are sampled from a fixed, known probability distribution.
**Question:** Assuming you know the access probabilities, how could you build a BST with the entropy property?

**Idea 1:** Pick the root to be the highest-probability element. Then, recursively build the subtrees.

**Idea 2:** Pick the root to balance the probabilities of the smaller elements and the bigger elements. Then, recursively build the subtrees.

**Model 2:** Queries are sampled from a fixed, known probability distribution.
A **weight-balanced tree** is a BST where the root is chosen to minimize the difference between the weights of the left and the right subtrees.

**Theorem:** In a weight-balanced tree with total weight $W$, the left and right subtrees each have weight at most $2W/3$.  

**Model 2:** Queries are sampled from a fixed, known probability distribution.
A **weight-balanced tree** is a BST where the root is chosen to minimize the difference between the weights of the left and the right subtrees.

**Theorem:** In a weight-balanced tree with total weight $W$, the left and right subtrees each have weight at most $2W / 3$.

**Case 1:** Some key has weight at least $W / 3$.

Picking this root guarantees a split no worse than $2W / 3$.

Remaining weight is at most $2W / 3$.

**Model 2:** Queries are sampled from a fixed, known probability distribution.
A **weight-balanced tree** is a BST where the root is chosen to minimize the difference between the weights of the left and the right subtrees.

*Theorem:* In a weight-balanced tree with total weight $W$, the left and right subtrees each have weight at most $2W/3$.

**Case 2:** All keys have weight less than $W/3$.

Scan from the left to the right until the cumulative weight is at least $W/3$.

Weight here is at least $W/3$ and at most $2W/3$.

This root guarantees a split of at most $2W/3$.

**Model 2:** Queries are sampled from a fixed, known probability distribution.
**Theorem:** Weight-balanced trees have the entropy property.

**Proof:** The expected cost of a lookup in a weight-balanced tree is

$$\sum_{i=1}^{n} p_i \cdot (1 + l_i)$$

where $p_i$ is the access probability of key $x_i$ and $l_i$ is the layer of the weight-balanced tree containing $x_i$.

**Model 2:** Queries are sampled from a fixed, known probability distribution.
**Theorem:** Weight-balanced trees have the entropy property.

**Proof:** The expected cost of a lookup in a weight-balanced tree is

\[ \sum_{i=1}^{n} p_i \cdot (1 + l_i) \]

where \( p_i \) is the access probability of key \( x_i \) and \( l_i \) is the layer of the weight-balanced tree containing \( x_i \).

Focus on some key \( x_i \) and its depth \( l_i \). After taking \( l_i \) steps in the tree, the remaining probability mass is at most \((\frac{2}{3})^{l_i}\).

**Theorem:** In a weight-balanced tree with total weight \( W \), the left and right subtrees each have weight at most \( \frac{2W}{3} \).

**Model 2:** Queries are sampled from a fixed, known probability distribution.
**Theorem:** Weight-balanced trees have the entropy property.

**Proof:** The expected cost of a lookup in a weight-balanced tree is

$$\sum_{i=1}^{n} p_i \cdot (1 + l_i)$$

where $p_i$ is the access probability of key $x_i$ and $l_i$ is the layer of the weight-balanced tree containing $x_i$.

Focus on some key $x_i$ and its depth $l_i$. After taking $l_i$ steps in the tree, the remaining probability mass is at most $(\frac{2}{3})^{l_i}$, so $l_i \leq -\log_{\frac{3}{2}} p_i$. Therefore,

$$\sum_{i=1}^{n} p_i \cdot (1 + l_i) \leq \sum_{i=1}^{n} p_i \cdot (1 - \log_{\frac{3}{2}} p_i)$$

Adding up $p_i$ over a probability distribution.

Shannon entropy, multiplied by some constant

$$1 + \sum_{i=1}^{n} (-p_i \log_{\frac{3}{2}} p_i)$$

**Model 2:** Queries are sampled from a fixed, known probability distribution.
**Theorem:** Weight-balanced trees have the entropy property.

**Proof:** The expected cost of a lookup in a weight-balanced tree is

\[
\sum_{i=1}^{n} p_i \cdot (1 + l_i)
\]

where \( p_i \) is the access probability of key \( x_i \) and \( l_i \) is the layer of the weight-balanced tree containing \( x_i \).

Focus on some key \( x_i \) and its depth \( l_i \). After taking \( l_i \) steps in the tree, the remaining probability mass is at most \((\frac{2}{3})^{l_i}\). This means that \((\frac{2}{3})^{l_i} \geq p_i\), so \( l_i \leq -\log_{\frac{3}{2}} p_i \). Therefore, we see that

\[
\sum_{i=1}^{n} p_i \cdot (1 + l_i) \leq \sum_{i=1}^{n} p_i \cdot (1 - \log_{\frac{3}{2}} p_i)
\]

\[
= 1 + \sum_{i=1}^{n} (-p_i \log_{\frac{3}{2}} p_i)
\]

\[
= O(1+H). \blacksquare
\]

**Model 2:** Queries are sampled from a fixed, known probability distribution.
**Theorem:** Weight-balanced trees have the entropy property.

**Fredman (1975):** Weight-balanced trees can be built in time $O(n \log n)$ in general and time $O(n)$ if the keys are already sorted.

**Knuth (1971):** The absolute best possible BST for a given set of keys can be built in time $O(n^2)$ using dynamic programming.

**Model 2:** Queries are sampled from a fixed, known probability distribution.
It’s possible to visit all the nodes in any BST in sorted order in time $O(n)$ via an inorder traversal, for an average lookup cost of $O(1)$.

The balance property says the average cost of a lookup, across all nodes, is $\Omega(\log n)$. Why doesn’t it apply here?

The entropy property says that, since each item is searched for exactly once, each lookup should take time $\Omega(\log n)$. Why doesn’t it apply here?

**Model 3:** Queries have *spatial locality*. If a key is queried, keys with nearby values will likely be queried.
The balance and entropy properties assume our searches start at the top of the tree.

In an inorder traversal, each search picks up where the last one left off. Therefore, these earlier bounds no longer apply.

**Idea:** Imagine we have a *finger* pointing at the last element of the BST that we’ve visited. After each lookup, the finger moves to the queried item.

**Model 3:** Queries have *spatial locality*. If a key is queried, keys with nearby values will likely be queried.
Suppose our last search was for some key $x_1$. Our next search is for key $x_2$. We know where key $x_1$ is.

Let $\Delta = |rank(x_2) - rank(x_1)|$.

Can we do the search in time $O(\Delta)$?
How about time $O(\log \Delta)$?
How about time $O(\log \log \Delta)$?

If the last key we searched for was 21...

...it should be faster to find 37, which is near...

...than 83, which is far.

If the last key we searched for was 21...

Model 3: Queries have **spatial locality**. If a key is queried, keys with nearby values will likely be queried.
Suppose our last search was for some key $x_1$. Our next search is for key $x_2$. We know where key $x_1$ is.

Let $\Delta = |\text{rank}(x_2) - \text{rank}(x_1)|$.

Can we do the search in time $O(\Delta)$?
How about time $O(\log \Delta)$?
How about time $O(\log \log \Delta)$?

Idea: Just do a simple linear scan.

Model 3: Queries have **spatial locality**. If a key is queried, keys with nearby values will likely be queried.
Suppose our last search was for some key $x_1$. Our next search is for key $x_2$. We know where key $x_1$ is.

Let $\Delta = |\text{rank}(x_2) - \text{rank}(x_1)|$.

Can we do the search in time $O(\Delta)$? How about time $O(\log \Delta)$? How about time $O(\log \log \Delta)$?

**Idea:** Use an exponential search to overshoot, then binary search over the range.

**Observation:** This is asymptotically at least as good as a binary search.

**Model 3:** Queries have spatial locality. If a key is queried, keys with nearby values will likely be queried.
Suppose our last search was for some key $x_1$. Our next search is for key $x_2$. We know where key $x_1$ is.

Let $\Delta = |\text{rank}(x_2) - \text{rank}(x_1)|$.

Can we do the search in time $O(\Delta)$? How about time $O(\log \Delta)$? How about time $O(\log \log \Delta)$?

$\Delta = O(n)$.

So if we could do this, we could do all searches in time $O(\log \log n)$, which is impossible.

(Proof idea: A comparison-based search making $k$ comparisons can only have $2^k$ possible outcomes. There are $n$ possible positions where the item could match.)
Suppose our last search was for some key $x_1$. Our next search is for key $x_2$. We know where key $x_1$ is.

Let $\Delta = |\text{rank}(x_2) - \text{rank}(x_1)|$.

Can we do the search in time $O(\Delta)$?
How about time $O(\log \Delta)$?
How about time $O(\log \log \Delta)$?

Question: Can we do this efficiently if the underlying set is changing?


Model 3: Queries have **spatial locality**. If a key is queried, keys with nearby values will likely be queried.

**Claim:** This simulates our earlier search. Runtime is $O(\log \Delta)$.

Scan up, looking at sibling nodes to determine where to search from.
Model 3: Queries have **spatial locality**. If a key is queried, keys with nearby values will likely be queried.

The *level-linked red/black tree* implements this dynamically.
Model 3: Queries have *spatial locality*. If a key is queried, keys with nearby values will likely be queried.

Great exercise: Do this with skiplists in (expected) time $O(\log \Delta)$.

Scan up, looking at sibling nodes to determine where to search from.
**Model 3:** Queries have **spatial locality**. If a key is queried, keys with nearby values will likely be queried.
**Model 4:** Queries have *temporal locality*. If a key is queried, it’s likely going to be queried again soon.
Model 4: Queries have *temporal locality*. If a key is queried, it’s likely going to be queried again soon.
Model 4: Queries have **temporal locality**. If a key is queried, it’s likely going to be queried again soon.

**Goal:** If only $t$ elements are “hot” at a particular time, make accesses to those “hot” elements take time $O(\log t)$, not $O(\log n)$. 
**Model 4:** Queries have *temporal locality*. If a key is queried, it’s likely going to be queried again soon.

**Intuition:** Any tree structure with a fixed shape is going to have a hard time making these queries fast.

**Idea:** What if we move elements around?
**Intuition:** Use a sequence of trees. Keep “hot” elements in the earlier trees.

“Hot” elements (recently accessed)

“Cold” elements (haven’t used in a while)

**Model 4:** Queries have **temporal locality**. If a key is queried, it’s likely going to be queried again soon.
Model 4: Queries have *temporal locality*. If a key is queried, it’s likely going to be queried again soon.

**Intuition:** Use a sequence of trees. Keep “hot” elements in the earlier trees.

To look up an element, search each tree in order, move it to the first tree.

To fill the gap left in the earlier tree, move the least-recently-accessed item from each tree into the next tree until the gap is filled.
Intuition: Use a sequence of trees. Keep “hot” elements in the earlier trees.

To insert an element, put it in the first tree. Then, repeatedly kick the oldest element out of each tree and into the next.

Model 4: Queries have **temporal locality**. If a key is queried, it’s likely going to be queried again soon.
**Intuition:** Use a sequence of trees. Keep “hot” elements in the earlier trees.

**Question:** How efficient is this strategy?

**Answer:** It depends on how big the trees are.

---

**Model 4:** Queries have *temporal locality*. If a key is queried, it’s likely going to be queried again soon.
**Intuition:** Use a sequence of trees. Keep “hot” elements in the earlier trees.

Earlier trees should be small so “hot” items can be found quickly. The cost of a lookup in a tree depends on the height of that tree.

**Idea:** Make each tree’s height double that of the previous tree.

**Model 4:** Queries have *temporal locality*. If a key is queried, it’s likely going to be queried again soon.
**Intuition:** Use a sequence of trees. Keep “hot” elements in the earlier trees.

**Idea:** Each tree’s height is double that of the previous tree.

Tree heights:

\[ 2^0, 2^1, 2^2, 2^3, \ldots, \]

Nodes per tree (roughly):

\[ 2^{2^0}, 2^{2^1}, 2^{2^2}, 2^{2^3}, \ldots \]

**Model 4:** Queries have *temporal locality*. If a key is queried, it’s likely going to be queried again soon.
**Intuition:** Use a sequence of trees. Keep “hot” elements in the earlier trees.

To insert an element, put it in the first tree. Then, repeatedly kick the oldest element out of each tree and into the next.

**Question:** How much time does this take, as a function of $n$?

**Model 4:** Queries have **temporal locality**. If a key is queried, it’s likely going to be queried again soon.
**Intuition:** Use a sequence of trees. Keep “hot” elements in the earlier trees.

To insert an element, put it in the first tree. Then, repeatedly kick the oldest element out of each tree and into the next.

Cost:

\[
\log 2^0 + \ldots + \log 2^{2 \log \log n} = 2^0 + 2^1 + \ldots + 2^{\log \log n} = 2^{1+\log \log n} - 1 = 2 \log n - 1 = O(\log n)
\]

---

**Model 4:** Queries have **temporal locality**. If a key is queried, it’s likely going to be queried again soon.
*Intuition:* Use a sequence of trees. Keep “hot” elements in the earlier trees.

To look up an element, search each tree in order, move it to the first tree, then kick older elements back.

Elements are roughly sorted by access time.

*Question:* How long does it take to look up an element here?

2^{2^0} \rightarrow 2^{2^1} \rightarrow 2^{2^2} \rightarrow \ldots \rightarrow 2^{2^{\lg \lg n}}

*Model 4:* Queries have *temporal locality*. If a key is queried, it’s likely going to be queried again soon.
**Intuition:** Use a sequence of trees. Keep “hot” elements in the earlier trees.

The cost of looking up an item $x$ depends on how long it’s been since we last queried it.

Suppose that we have queried $t$ total items since we last queried $x$.

Then $x$ is in, at most, the $(\lg \lg t)$th tree.

Cost of querying $x$:

$$\log 2^{2^0} + \ldots + \log 2^{2^{\lg \lg t}} = O(\log t)$$

**Model 4:** Queries have **temporal locality**. If a key is queried, it’s likely going to be queried again soon.
**Intuition:** Use a sequence of trees. Keep “hot” elements in the earlier trees.

A BST has the **working set property** if the (amortized) cost of looking up an element is $O(\log t)$, where $t$ is the number of items looked up more recently than the queried element.

This data structure is called **Iacono’s working set structure**, after its inventor.

**Model 4:** Queries have **temporal locality**. If a key is queried, it’s likely going to be queried again soon.
These models are in tension with one another.
These models are in tension with one another.

<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</td>
<td>Lookups take $O(\log \Delta)$; $\Delta$ is rank distance.</td>
<td>Level-linked BST with finger Iacono’s</td>
</tr>
<tr>
<td>Finger</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Working Set</td>
<td>Lookups take $O(\log t)$; $t$ is recency.</td>
<td>Iacono’s structure</td>
</tr>
</tbody>
</table>
These models are in tension with one another.
<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 $O(\log \Delta)$; $\Delta$ is rank distance.</td>
<td>Level-linked BST with finger</td>
</tr>
<tr>
<td>Working Set</td>
<td>Lookups take $O(\log t)$; $t$ is recency.</td>
<td>Iacono’s structure</td>
</tr>
<tr>
<td><strong>Property</strong></td>
<td><strong>Description</strong></td>
<td><strong>Met by</strong></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) is rank distance.</td>
<td>Splay tree*</td>
</tr>
<tr>
<td>Working Set</td>
<td>Lookups take (O(\log t)); (t) is recency.</td>
<td>Splay tree*</td>
</tr>
</tbody>
</table>

Yes!
Next Time

- **Splay Trees**
  - A simple, fast, flexible BST.
- **Splitting and Joining Trees**
  - Combining trees together, or breaking them apart.
- **Sum-of-Logs Potentials**
  - Analyzing the efficiency of splay trees.
- **The Dynamic Optimality Conjecture**
  - Is there a single best BST?