Better than Balanced BSTs

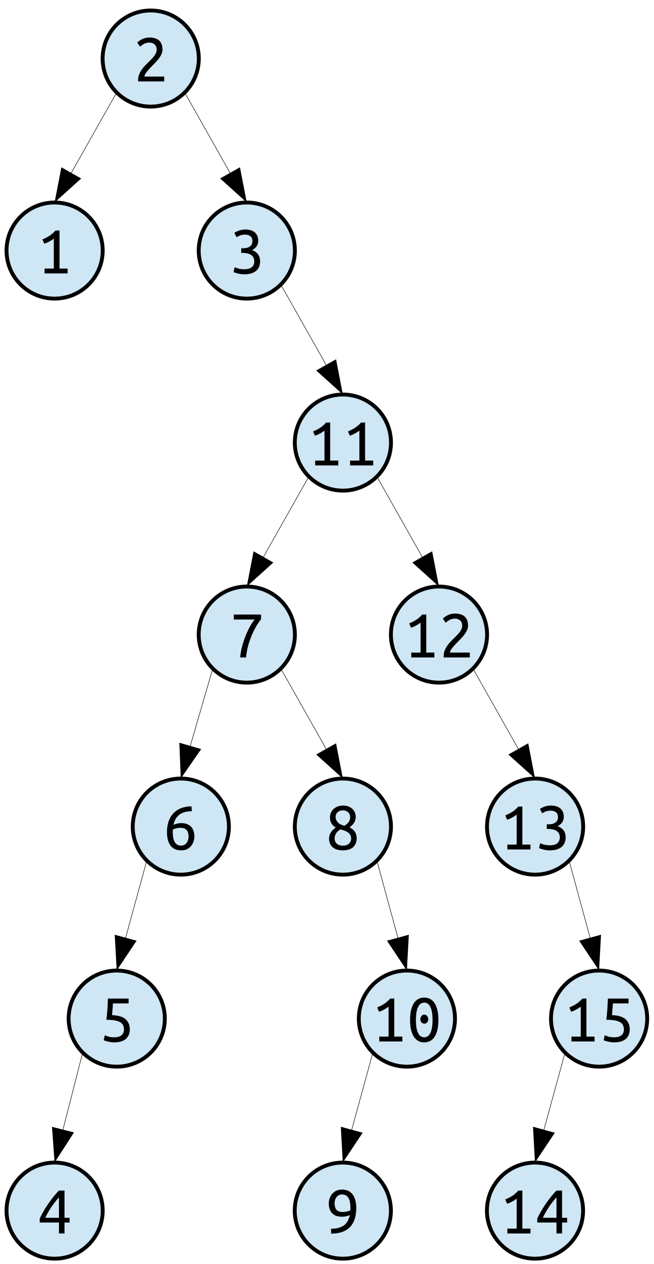
Outline for Today

- ***Beyond Worst-Case Efficiency***
  - When O(log *n*) isn't enough.
- ***Weight-Equalized Trees***
  - Balancing by access probabilities.
- ***Finger Search Trees***
  - Picking up where you left off.
- ***Iacono's Working Set Structure***
  - Keeping exciting things accessible.

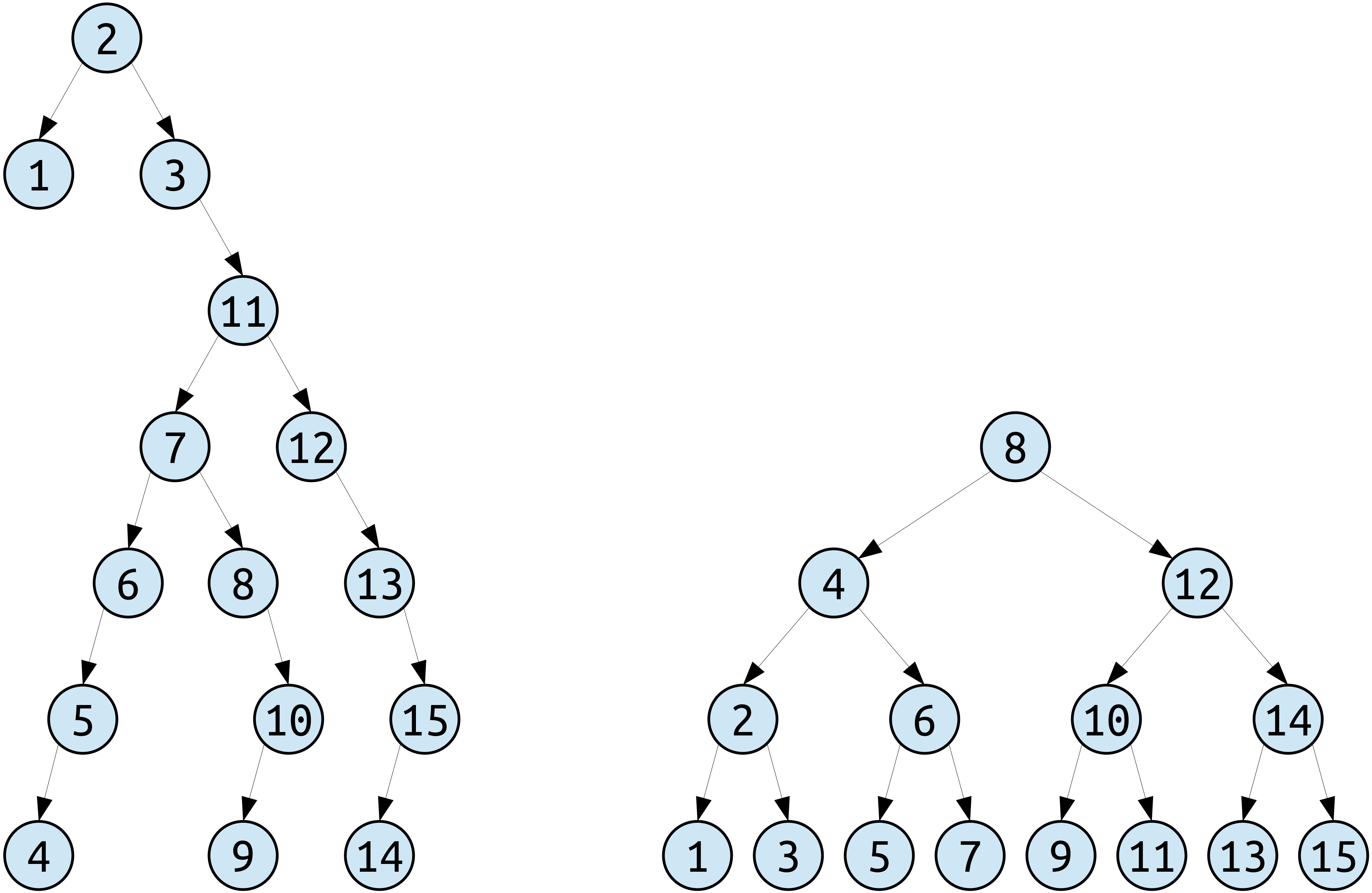
Can you build a binary search tree where lookups are faster than O(log n)?

Key Idea: The guarantees we want from a data structure depend on our model of how that data structure will be used.

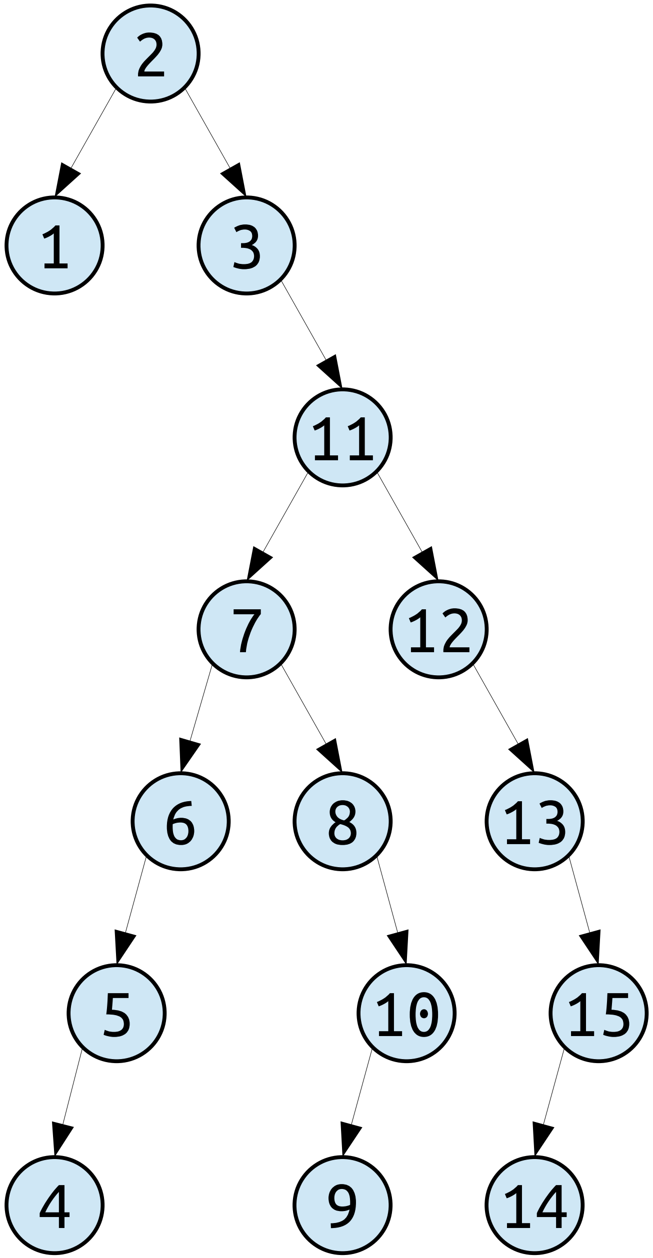
Model 1: Queries are chosen maliciously.



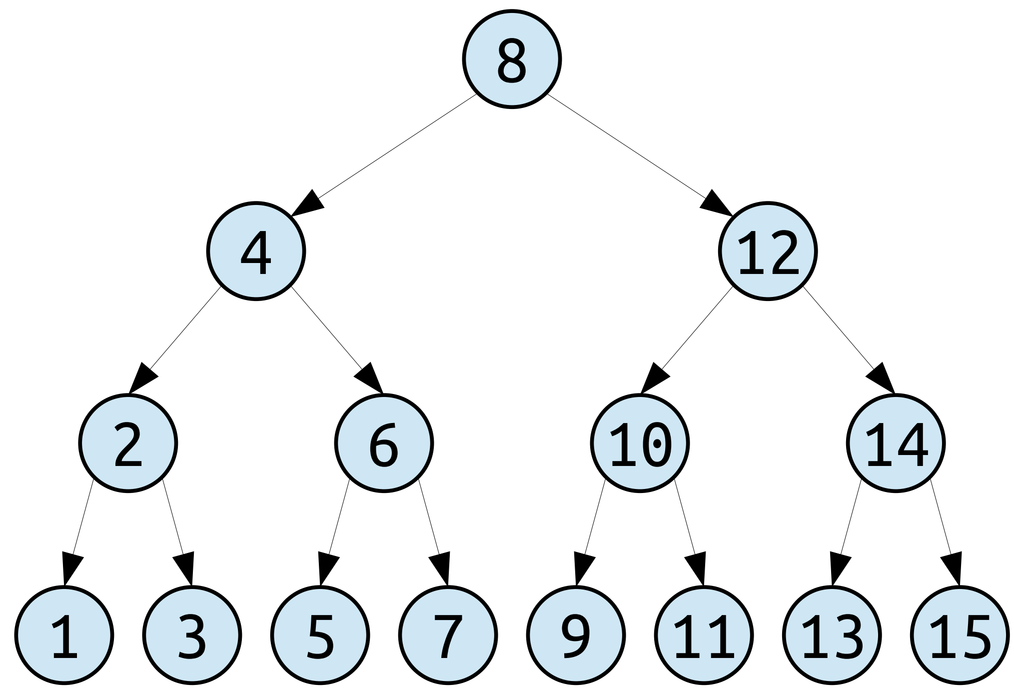
Model 1: Queries are chosen maliciously.



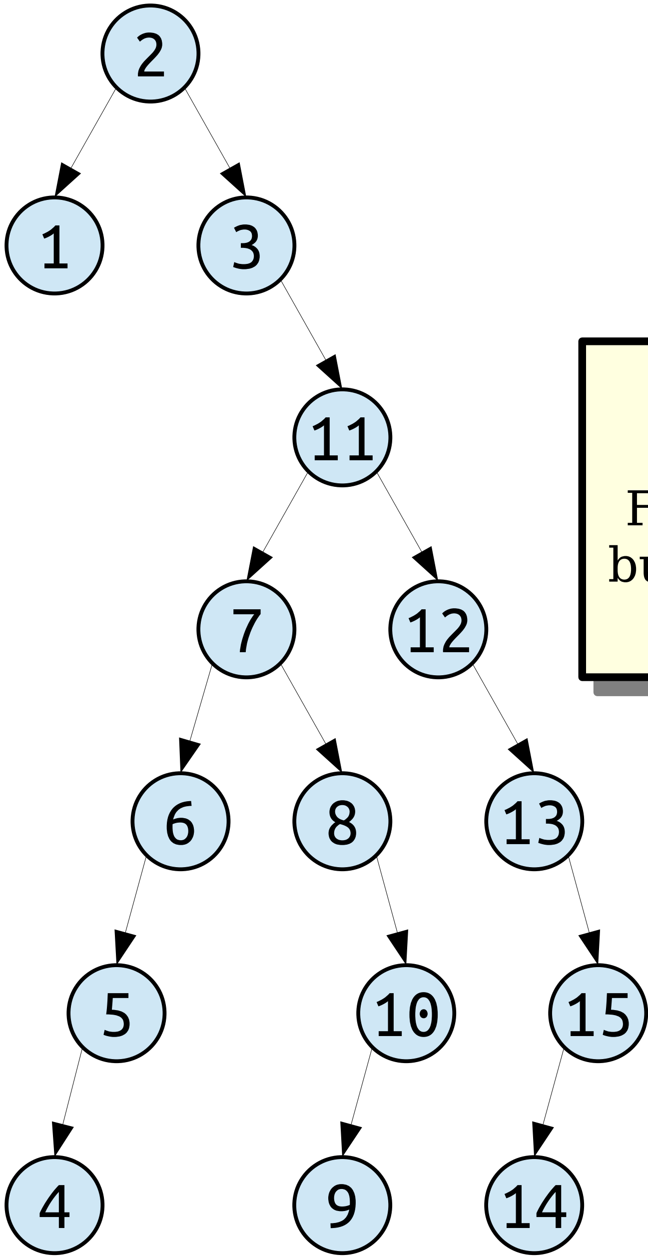
Model 1: Queries are chosen maliciously.



Claim: The worst-case lookup cost of a lookup in *any* BST with n nodes is at least $\Omega(\log n)$.



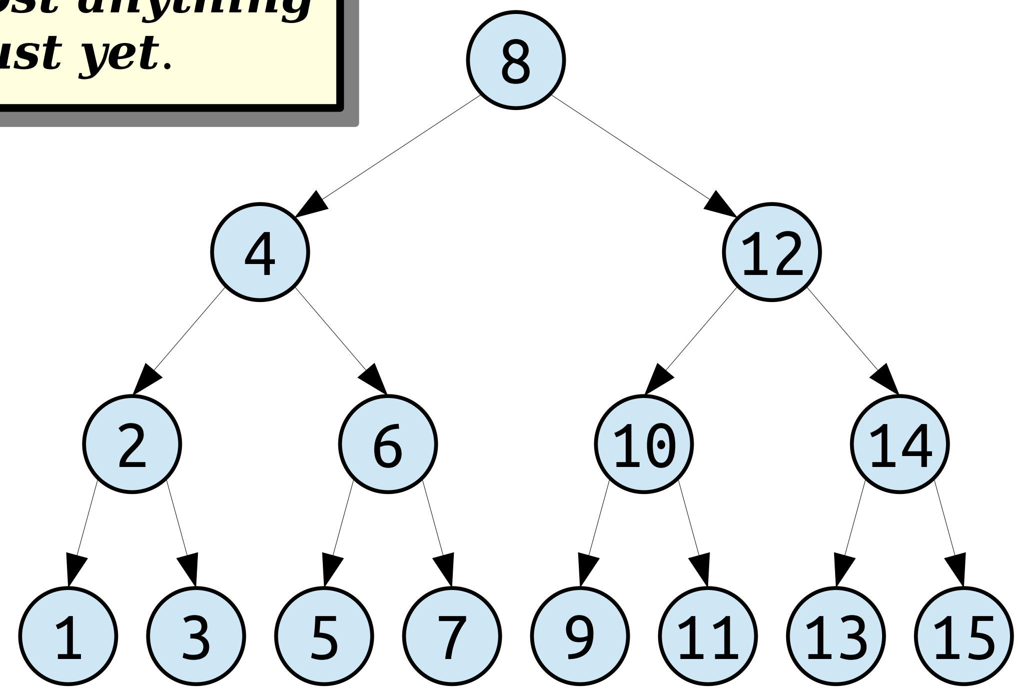
Model 1: Queries are chosen maliciously.



Claim: The worst-case lookup cost of a lookup in *any* BST with n nodes is at least $\Omega(\log n)$.

Why?

Formulate a hypothesis, but ***don't post anything in chat just yet***.

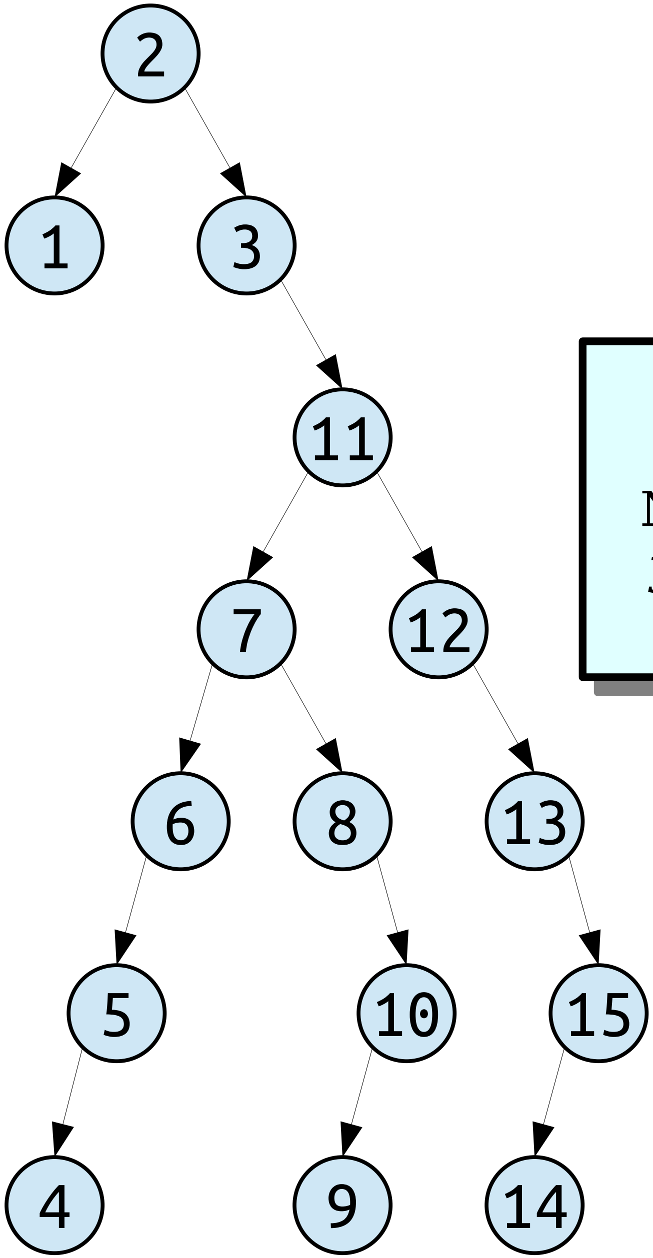


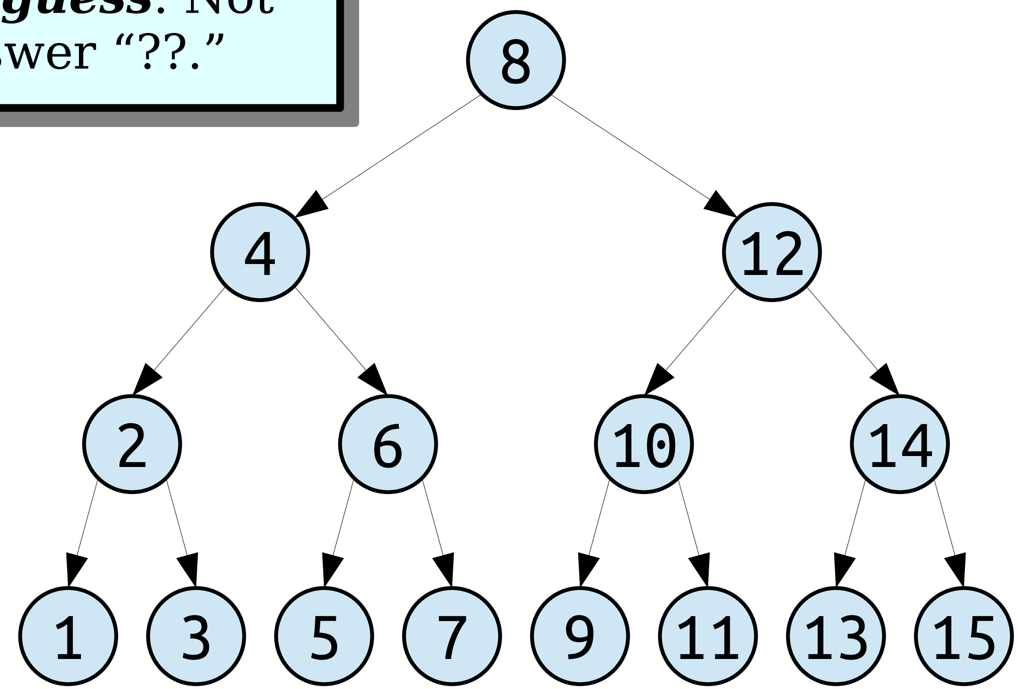
Model 1: Queries are chosen maliciously.

Claim: The worst-case lookup cost of a lookup in *any* BST with n nodes is at least $\Omega(\log n)$.

Why?

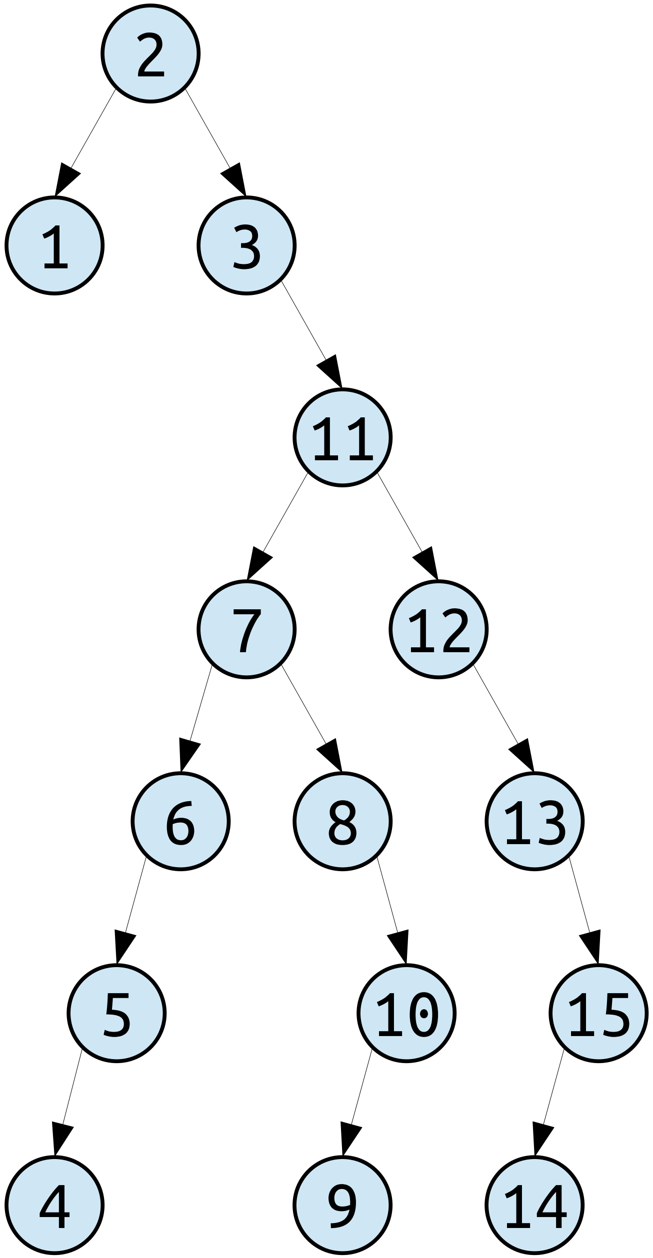
Now, ***private chat me your best guess***. Not sure? Answer “??.”

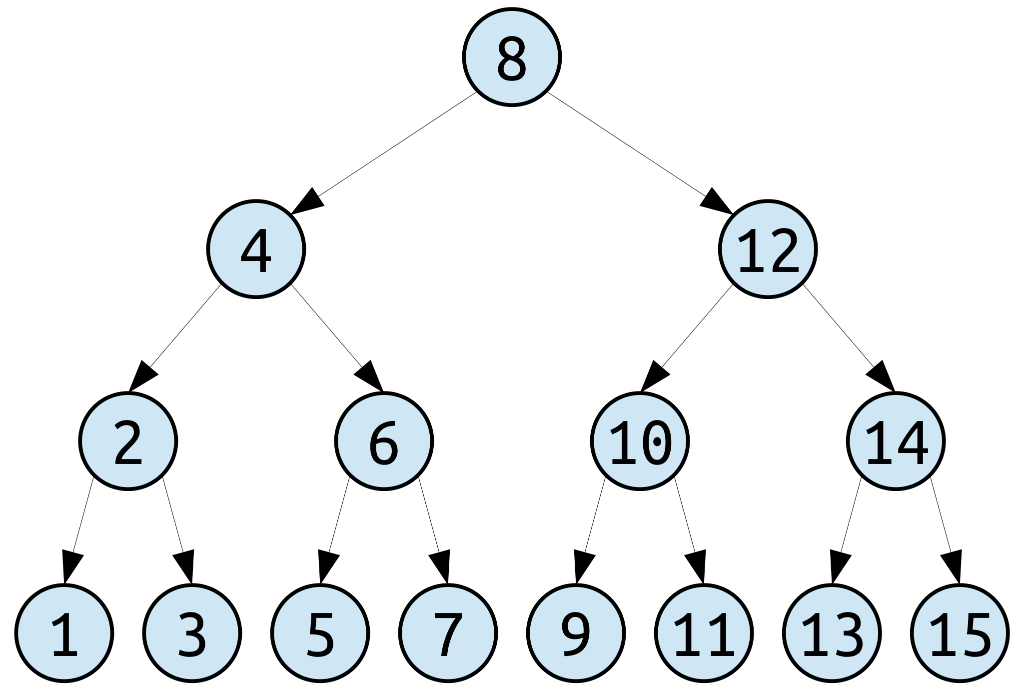




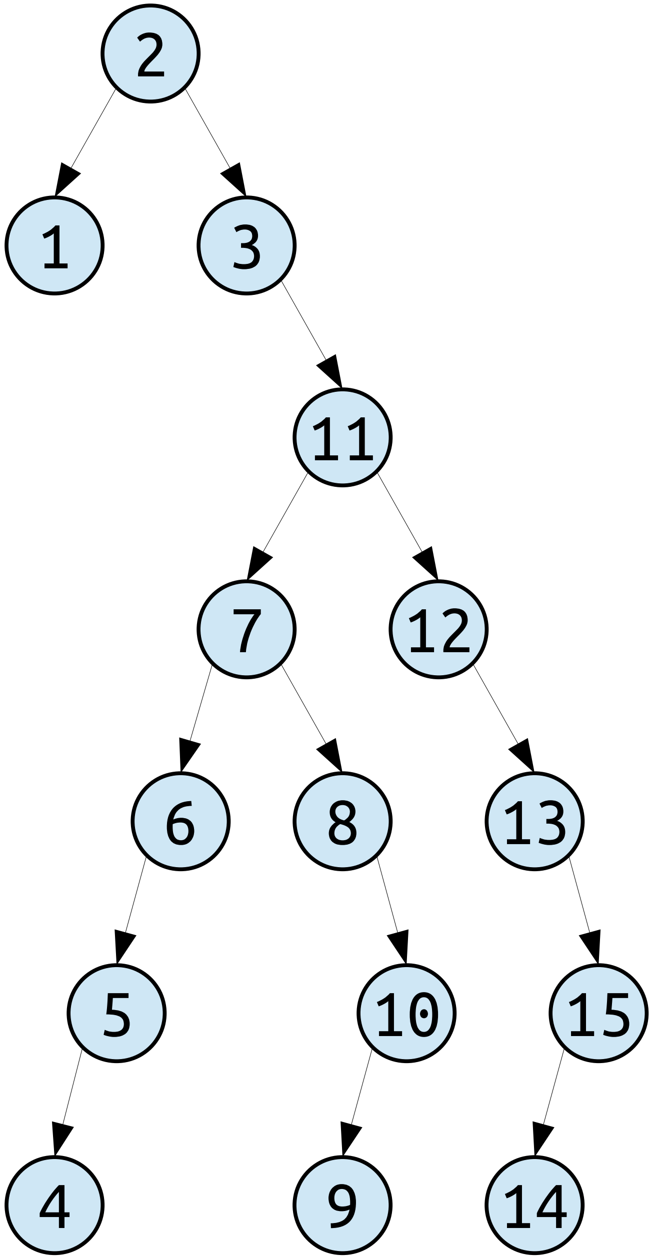
Model 1: Queries are chosen maliciously.

Claim: The worst-case lookup cost of a lookup in *any* BST with n nodes is at least $\Omega(\log n)$.



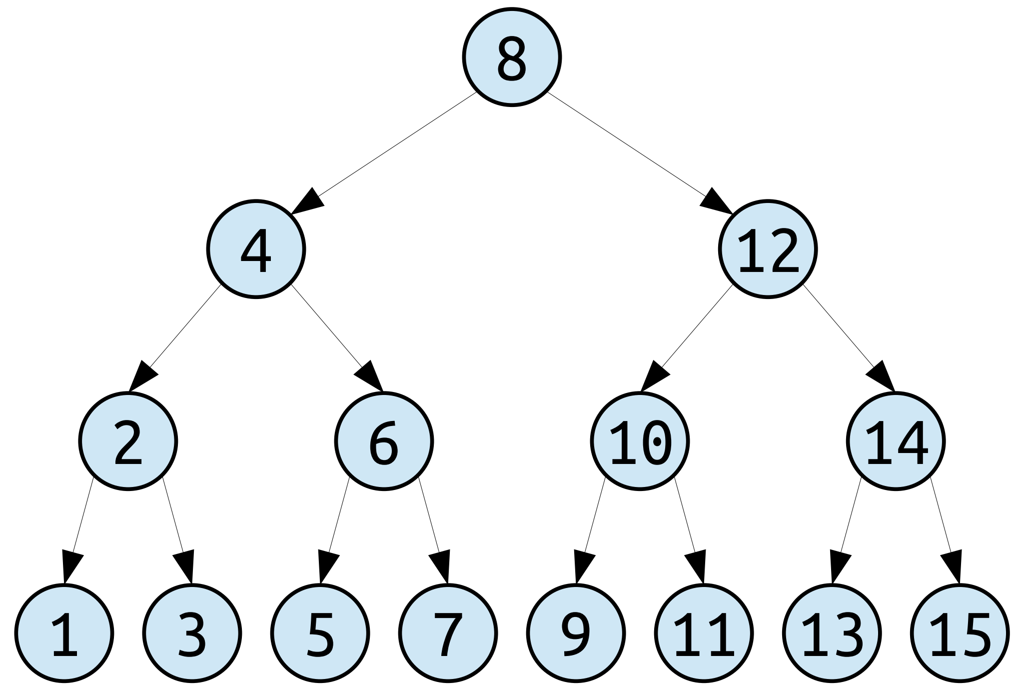


Model 1: Queries are chosen maliciously.

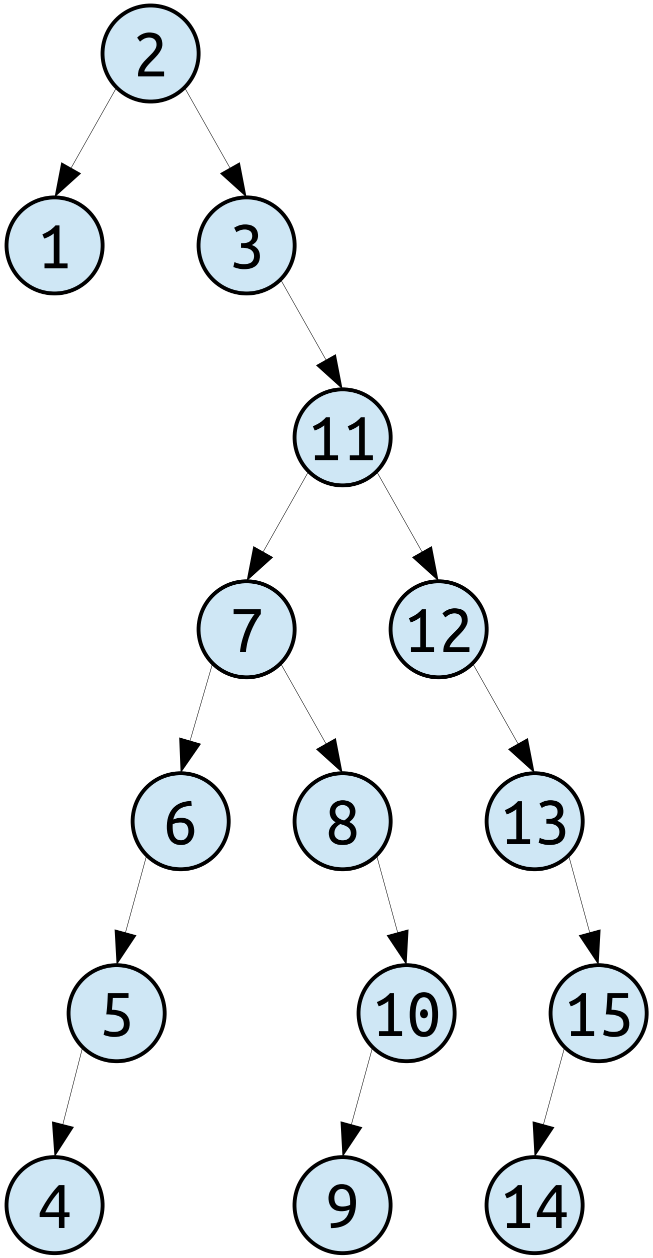


Claim: The worst-case lookup cost of a lookup in *any* BST with n nodes is at least $\Omega(\log n)$.

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



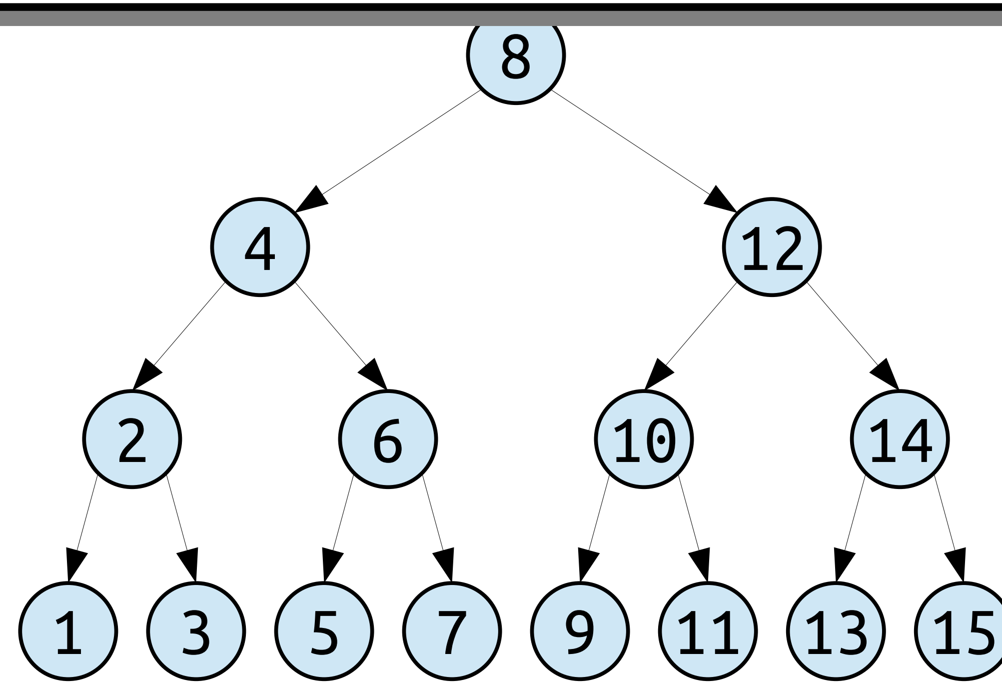
Model 1: Queries are chosen maliciously.



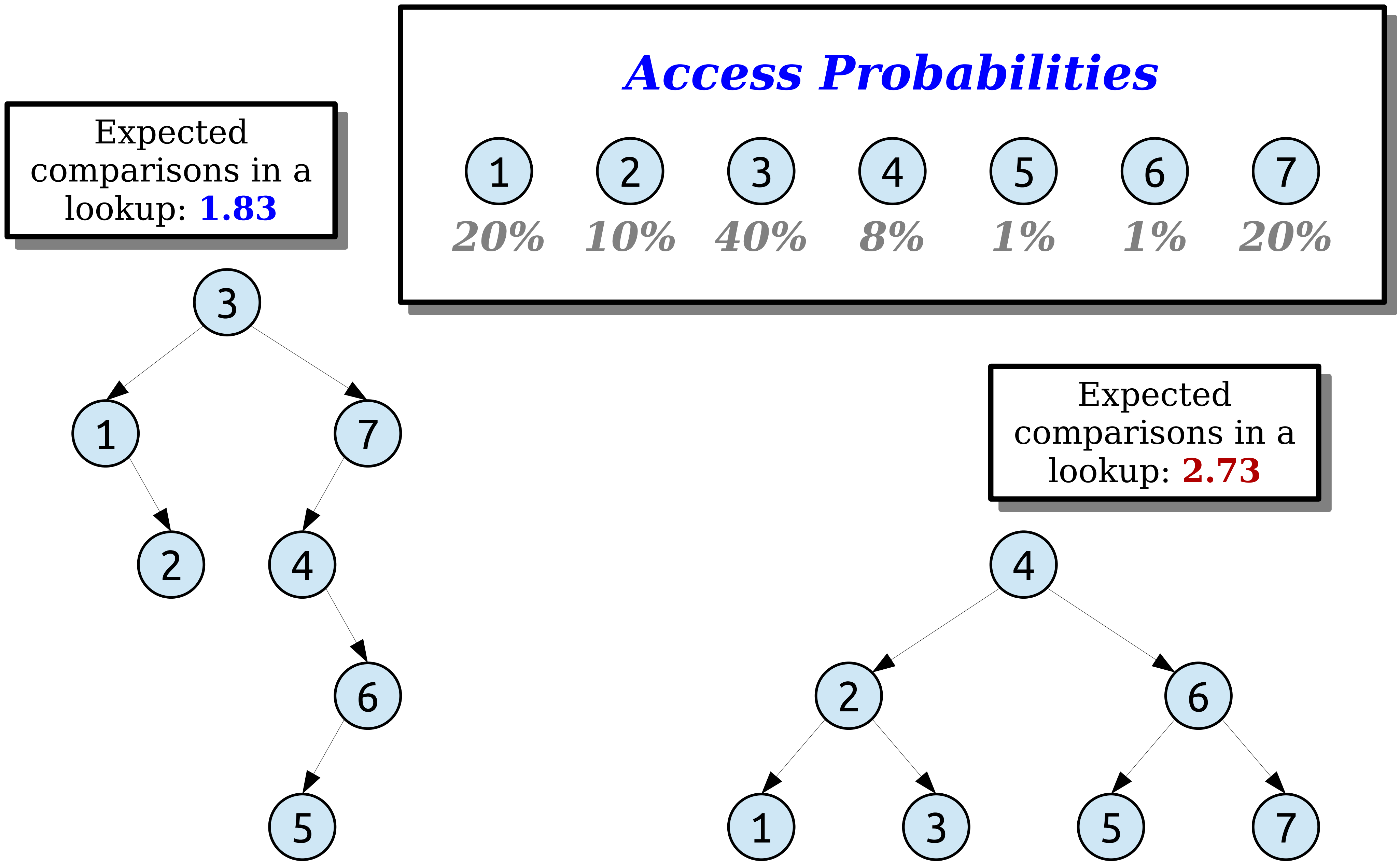
A binary search tree satisfies the ***balance property*** if the (amortized) cost of any lookup in that tree is O(log *n*).

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.

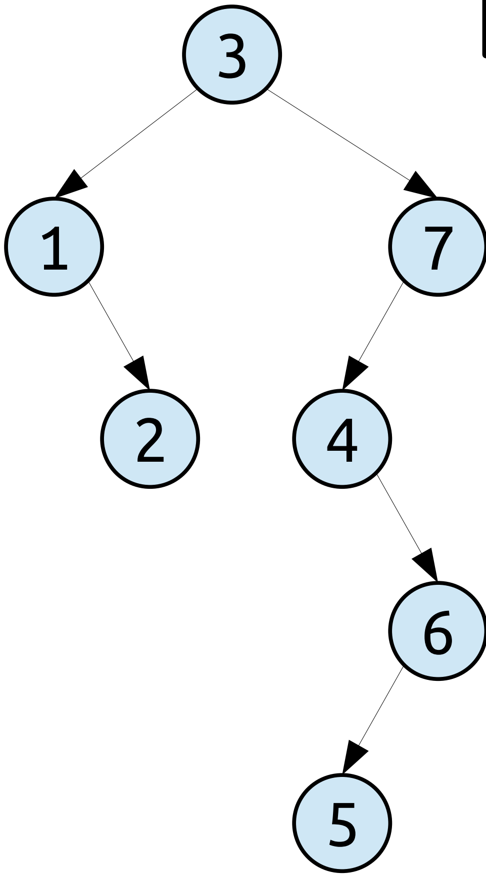


Model 2: Queries are sampled from a fixed, known probability distribution.

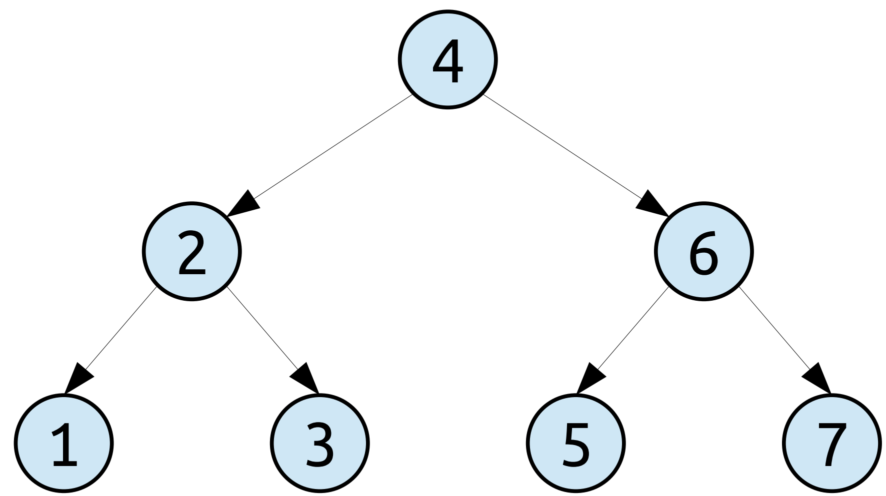
Access Probabilities

1	2	3	4	5	6	7
14%	*15%*	*14%*	*14%*	*14%*	*15%*	*14%*

Expected comparisons in a lookup: **2.87**

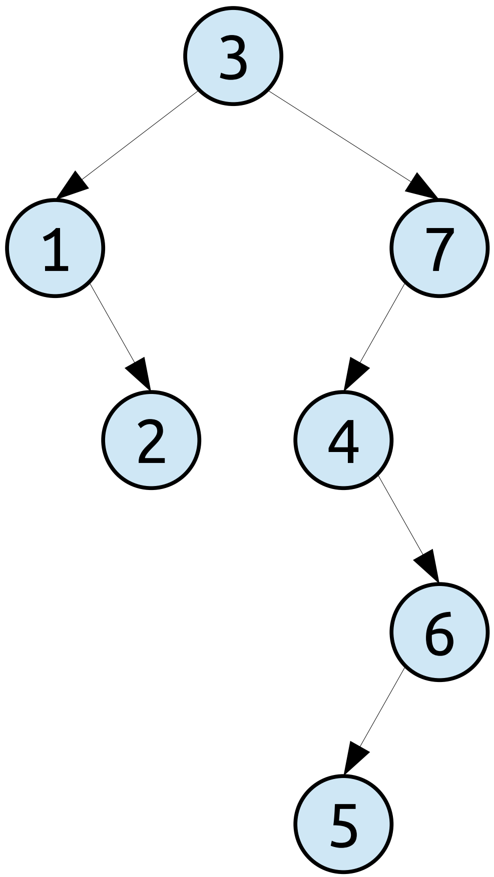


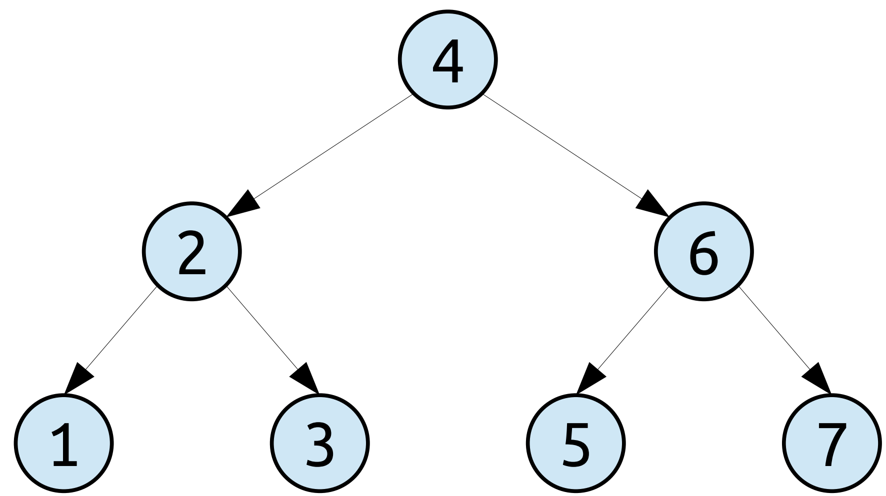
Expected comparisons in a lookup: **2.42**



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

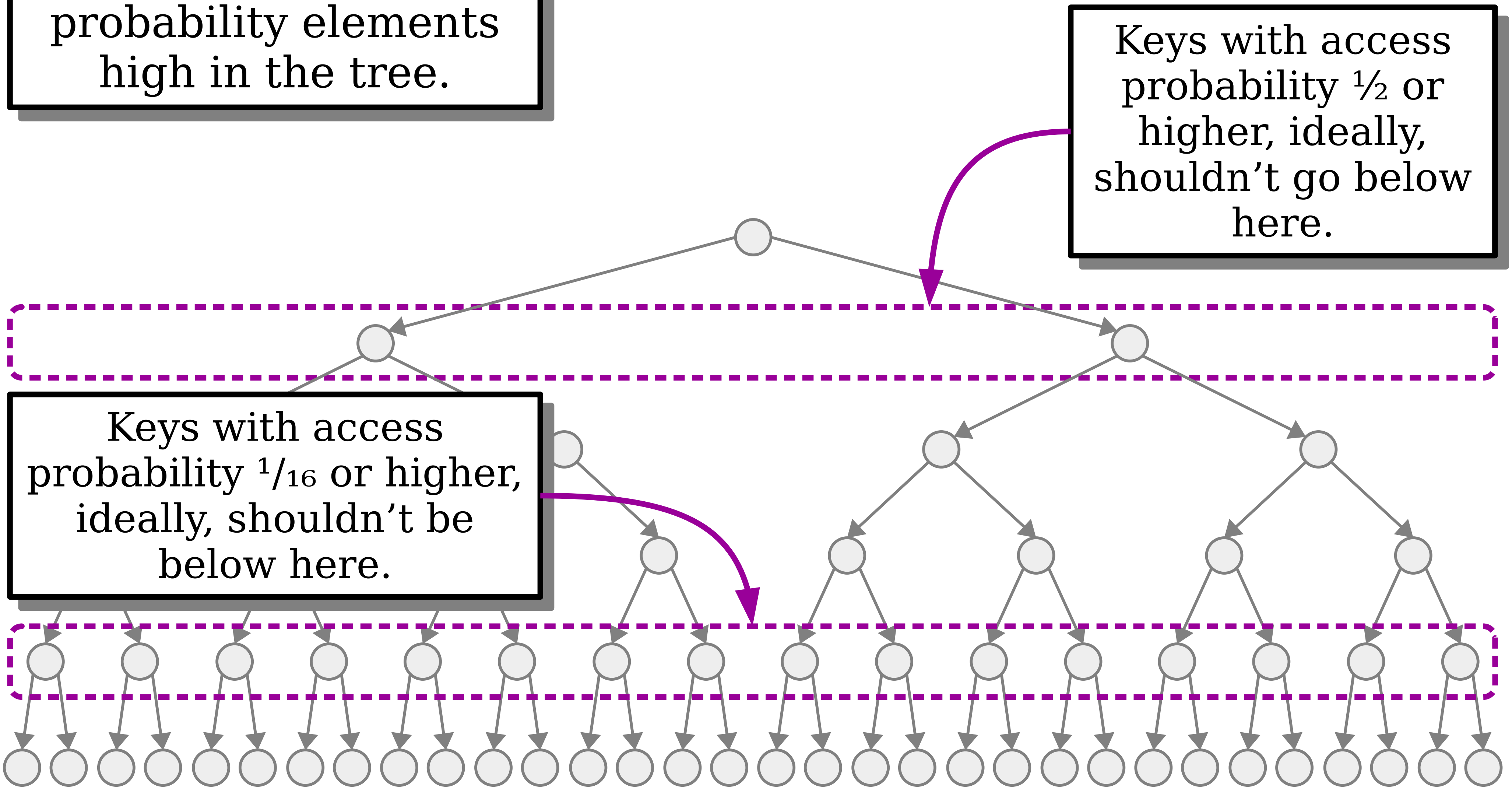
How do we know when we have a BST that's optimal with respect to *expected* lookup costs?





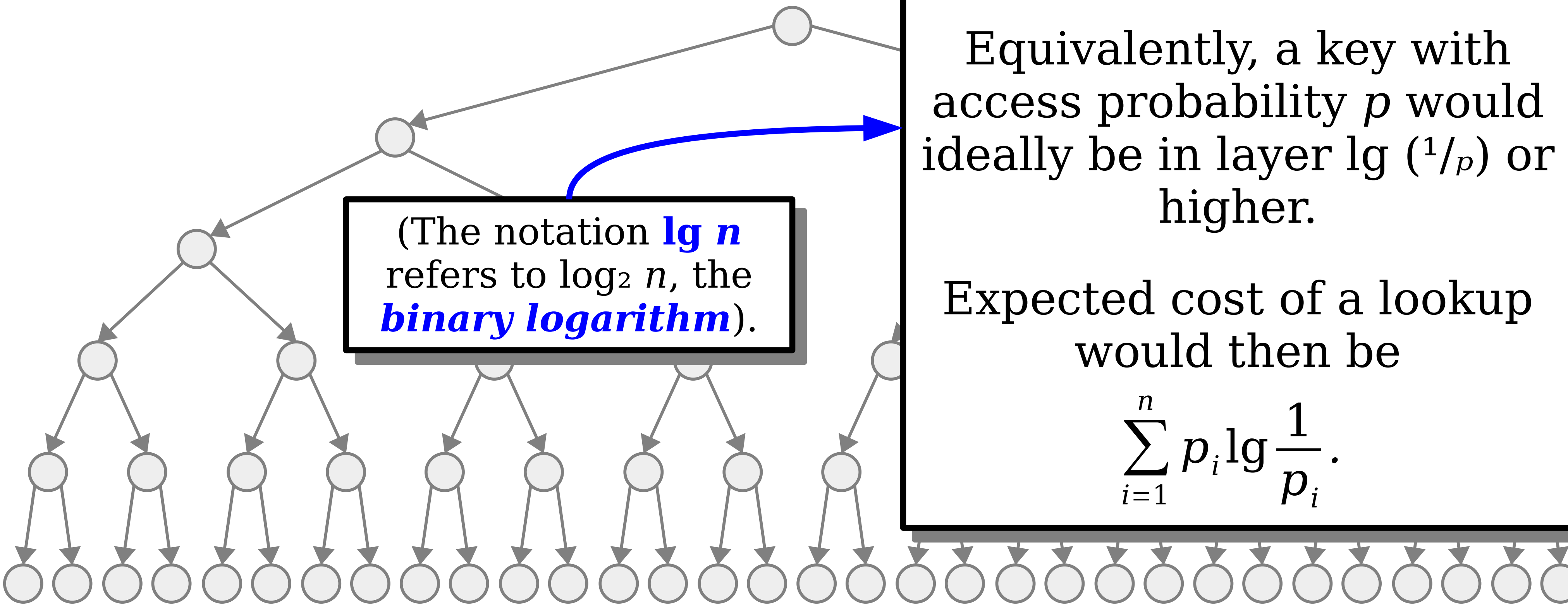
Model 2: Queries are sampled from a fixed, known probability distribution.

Intuition: Place high probability elements high in the tree.



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

Intuition: Place high probability elements high in the tree.



(The notation **lg *n*** refers to $\log_2 n$, the ***binary logarithm***).

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 $\lg (1/p)$ or higher.

Expected cost of a lookup would then be

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

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

$$\boldsymbol{H_p = \sum_{i=1}^{n} p_i \lg \frac{1}{p_i}.}$$

If you have n keys $x_1, \ldots, x_n$, what is the maximum possible value of H? What is the minimum possible value?

Formulate a hypothesis, but ***don't post anything in chat just yet***.

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

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

If you have n keys $x_1, \ldots, x_n$, what is the maximum possible value of H? What is the minimum possible value?

Now, ***private chat me your best guess***. Not sure? Just answer "??."

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

$$\boldsymbol{H_p = \sum_{i=1}^{n} p_i \lg \frac{1}{p_i}.}$$

If all elements have equal access probability ($p_i = {}^1/_n$):

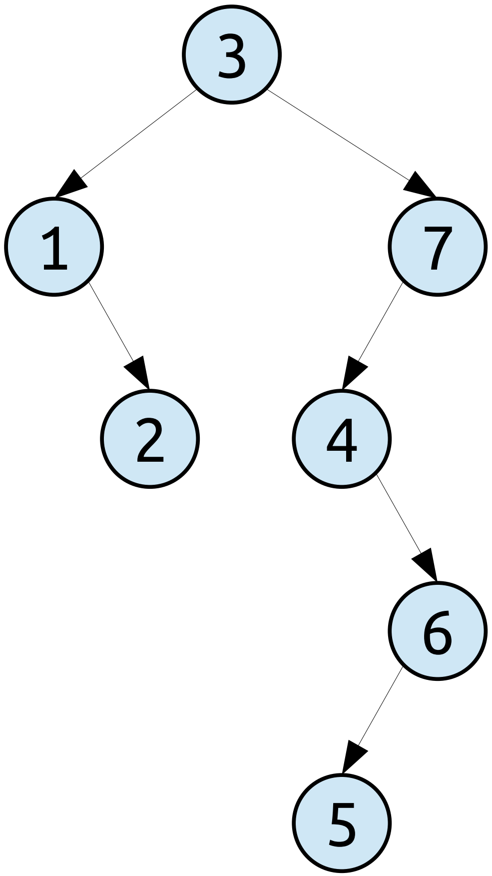
$$\begin{aligned} H_p &= \sum_{i=1}^{n} p_i \lg \frac{1}{p_i} \\ &= \sum_{i=1}^{n} \frac{1}{n} \lg n \\ &= \mathbf{lg}\, \boldsymbol{n} \end{aligned}$$

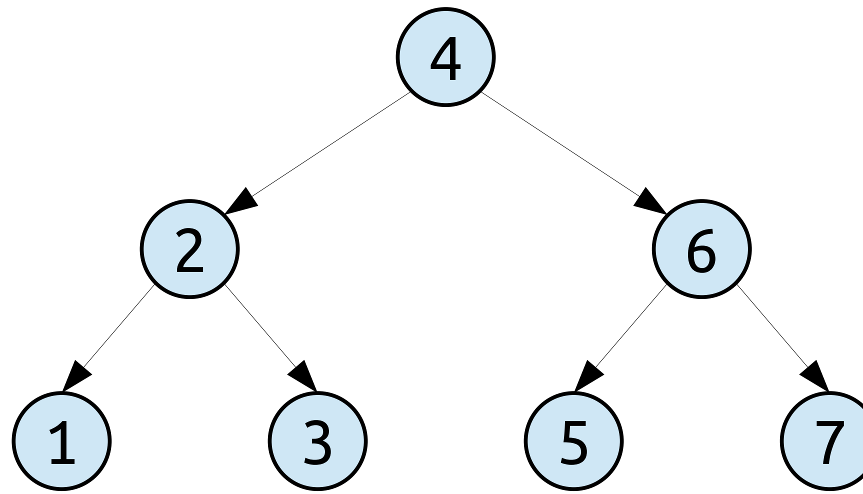
If only one element is ever accessed ($p_1 = 1$, $p_i = 0$), then

$$\begin{aligned} H_p &= \sum_{i=1}^{n} p_i \lg \frac{1}{p_i} \\ &= \lg 1 + \sum_{i=2}^{n} 0 \lg \frac{1}{0} \\ &= \mathbf{0} \end{aligned}$$

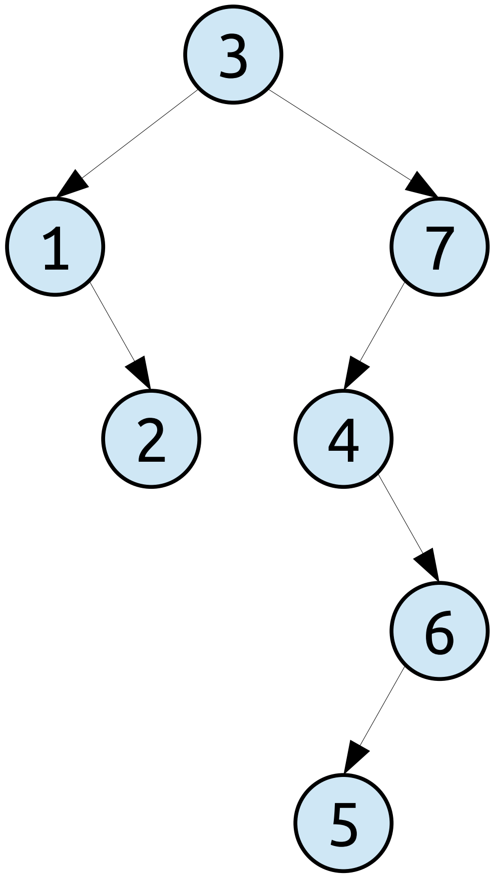
Model 2: Queries are sampled from a fixed, known probability distribution.

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.



Theorem: If accesses are sampled over a fixed 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.

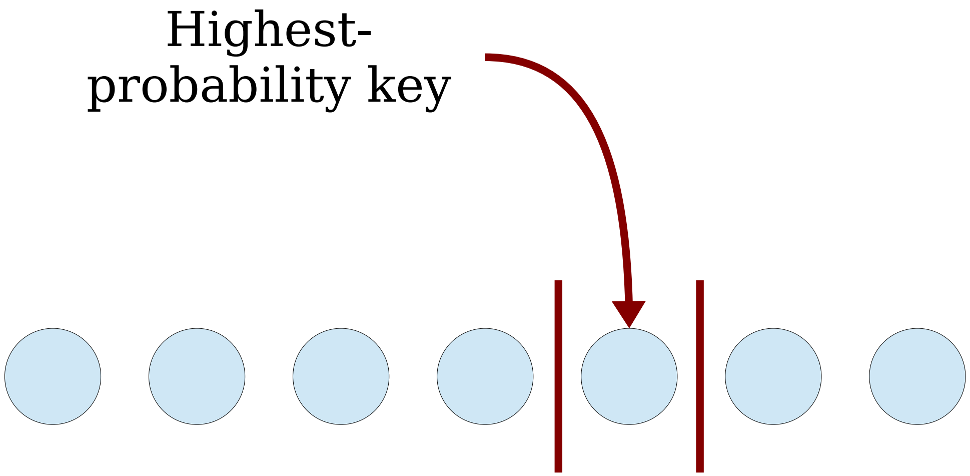
A binary search tree has the ***entropy property*** if the (amortized) expected cost of any lookup on that BST is $O(1 + H)$.

(Any BST with this property is optimal from a *expected-case* perspective, assuming a fixed probability distribution.)

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.

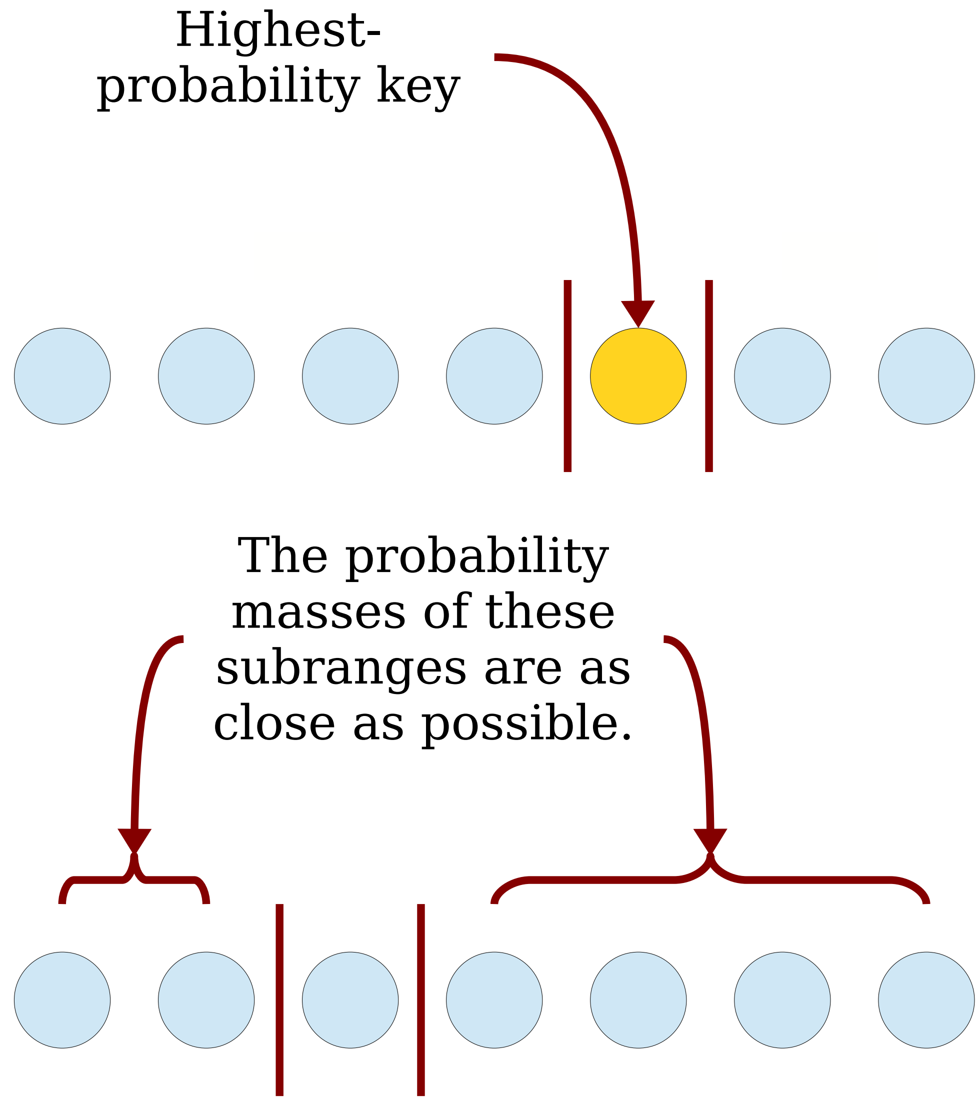


Model 2: Queries are sampled from a fixed, known probability distribution.

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.

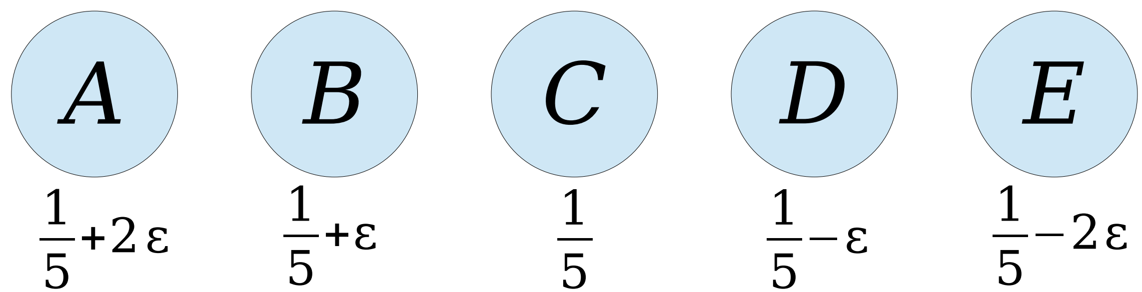
Idea 2: Pick the root to balance the probabilities of the smaller elements and the bigger elements. Then, recursively build the subtrees.

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

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.

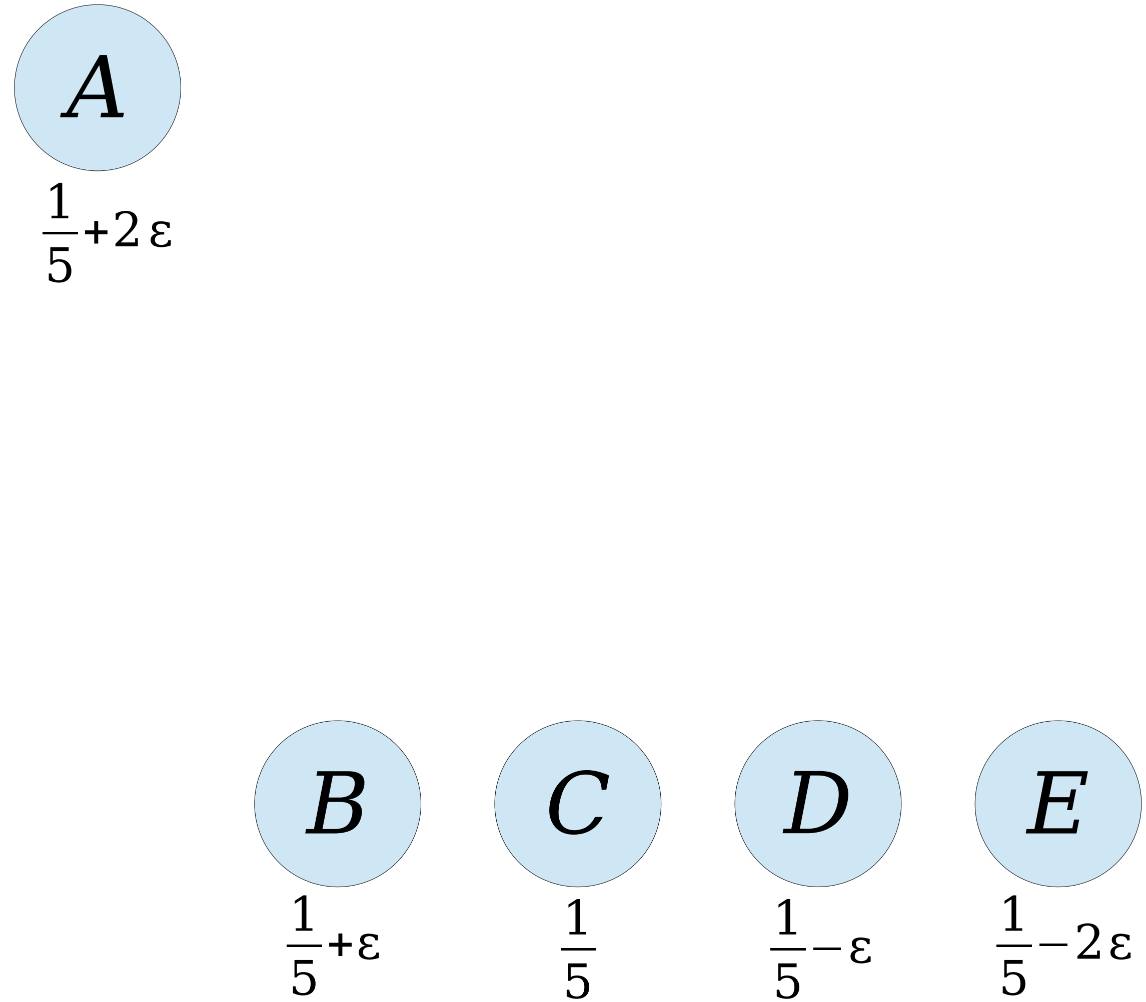


Model 2: Queries are sampled from a fixed, known probability distribution.

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.

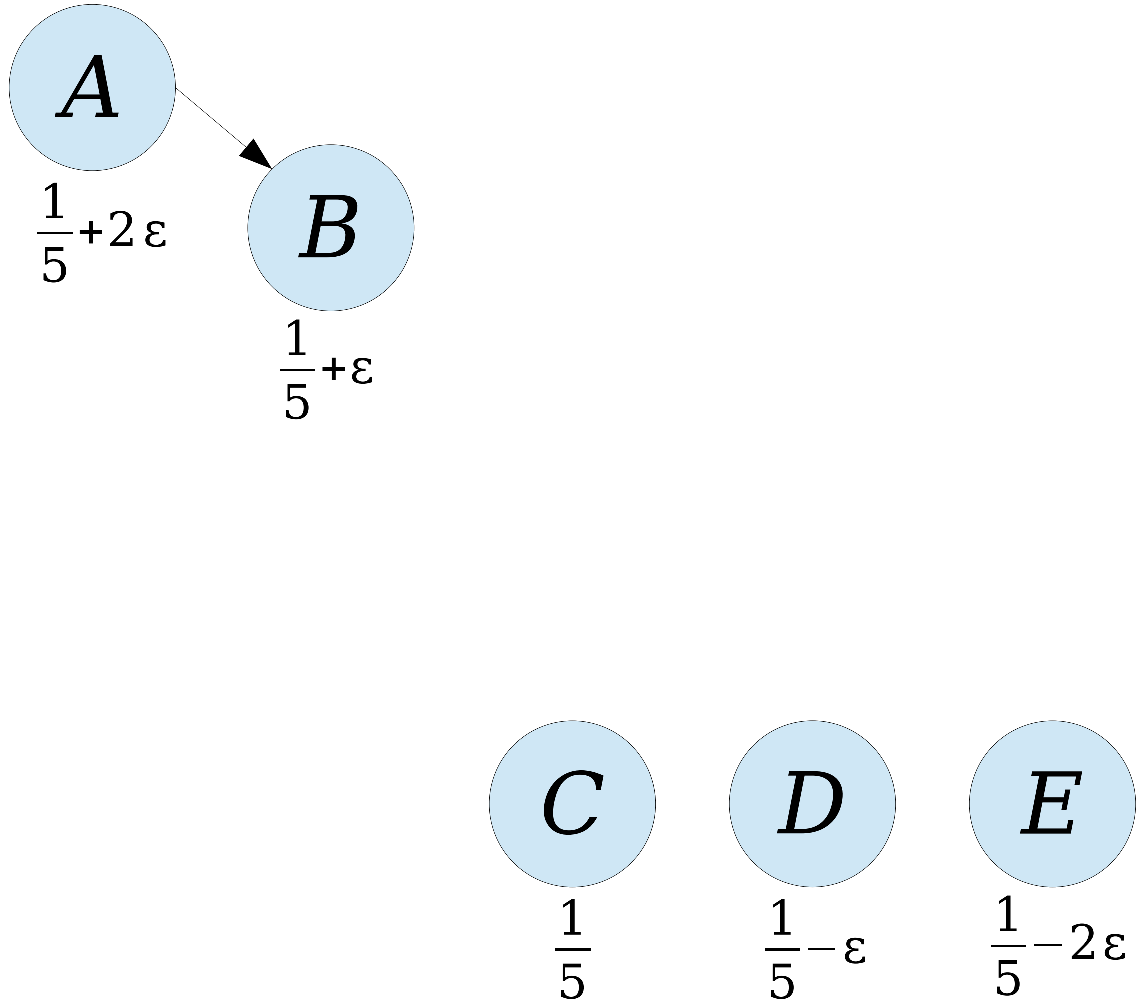


Model 2: Queries are sampled from a fixed, known probability distribution.

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.

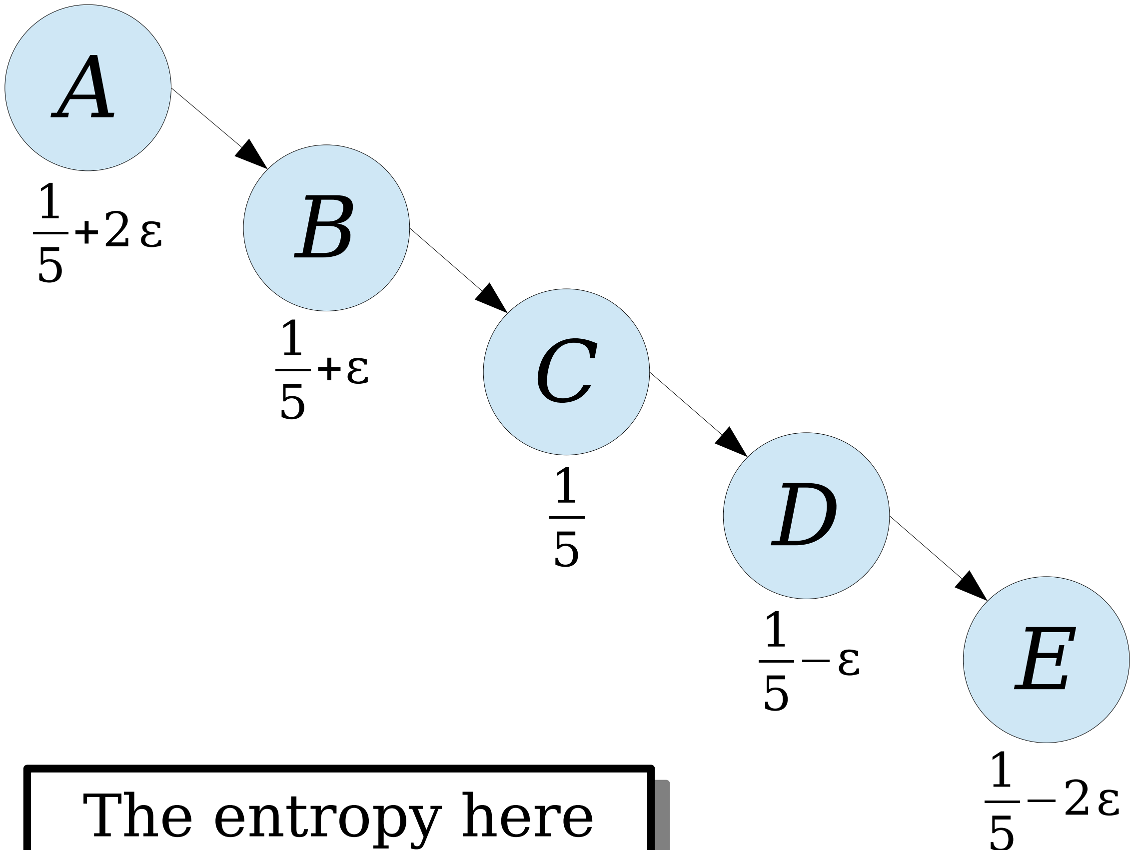


Model 2: Queries are sampled from a fixed, known probability distribution.

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.

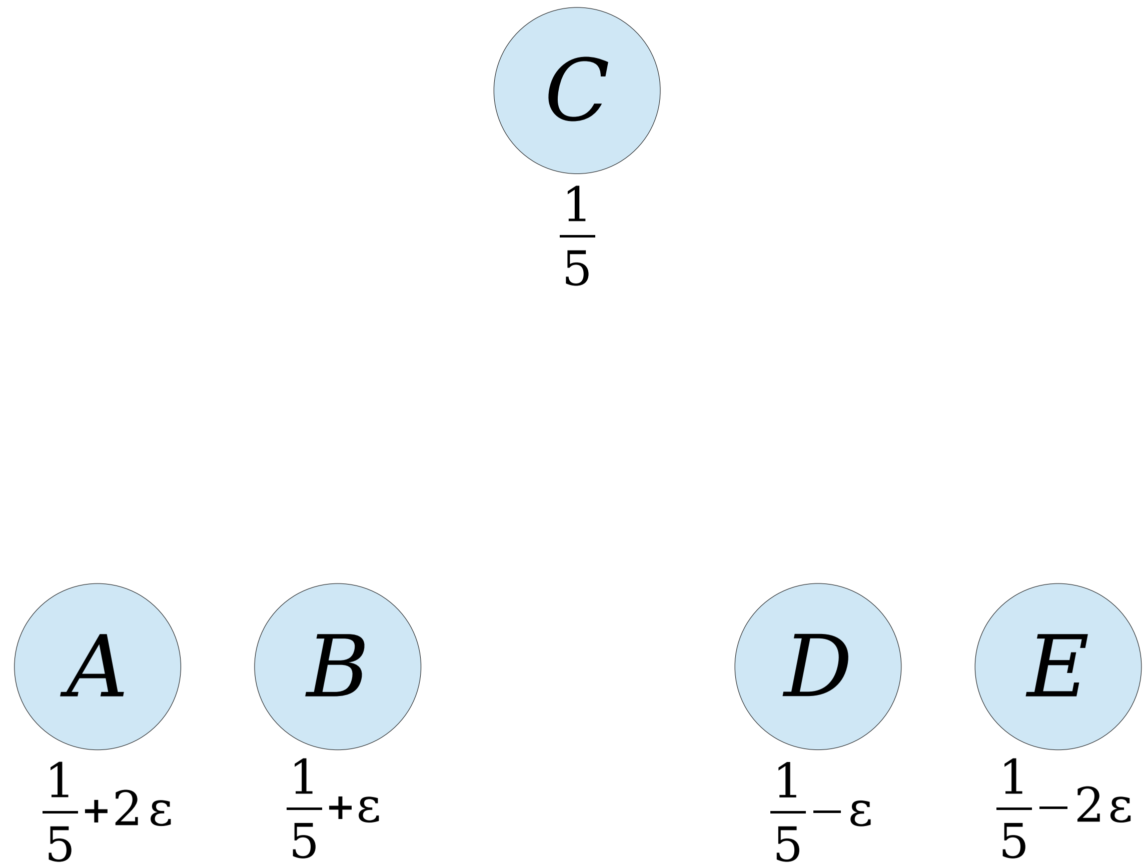
A	B	C	D	E
$\frac{1}{5}+2\varepsilon$	$\frac{1}{5}+\varepsilon$	$\frac{1}{5}$	$\frac{1}{5}-\varepsilon$	$\frac{1}{5}-2\varepsilon$

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.

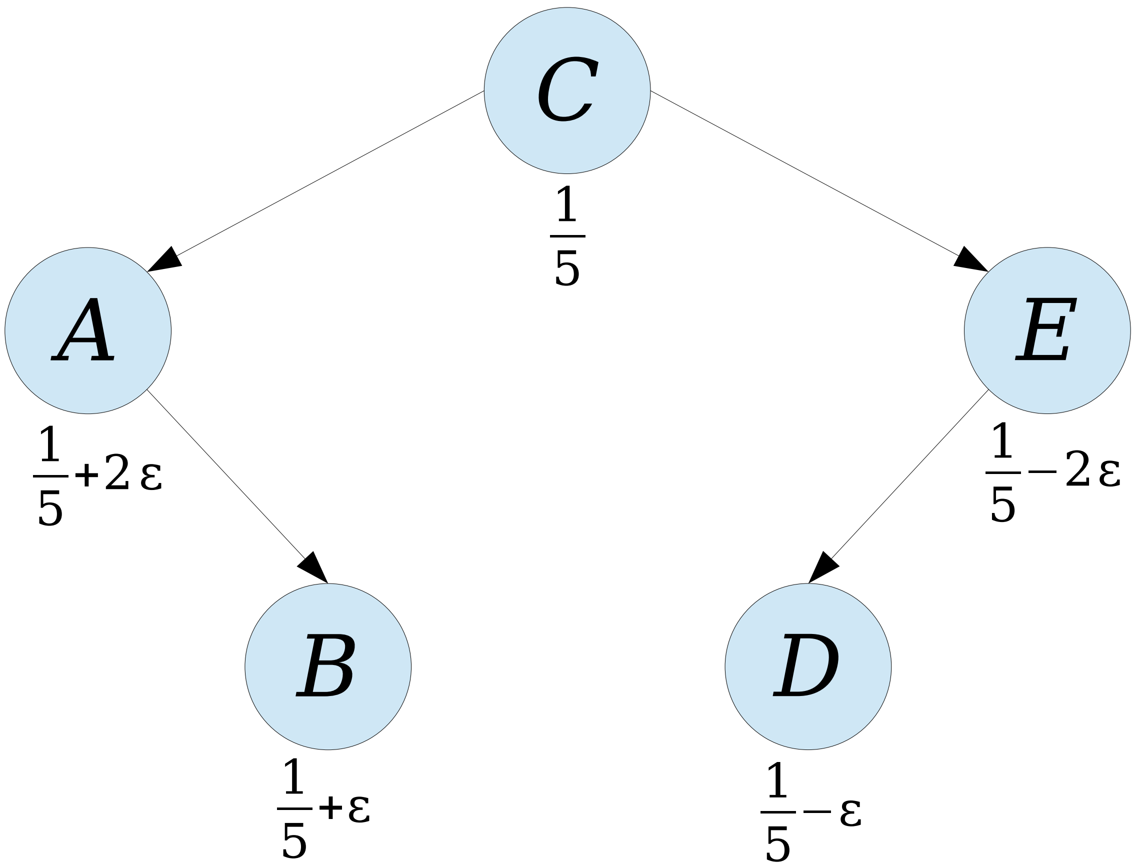


Model 2: Queries are sampled from a fixed, known probability distribution.

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-equalized tree*** is a BST where the root is chosen to minimize the weight difference between the left and right subtrees.

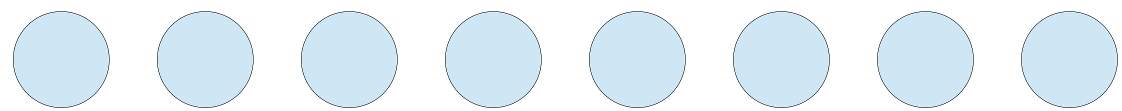
Theorem: In a weight-equalized tree with total weight W, the left and right subtrees each have weight at most $2W / 3$.

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

A ***weight-equalized tree*** is a BST where the root is chosen to minimize the weight difference between the left and right subtrees.

Theorem: In a weight-equalized 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 $W / 3$.

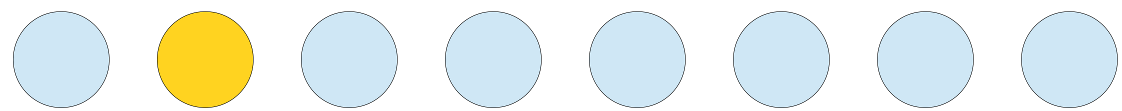


Model 2: Queries are sampled from a fixed, known probability distribution.

A ***weight-equalized tree*** is a BST where the root is chosen to minimize the weight difference between the left and right subtrees.

Theorem: In a weight-equalized 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 $W / 3$.



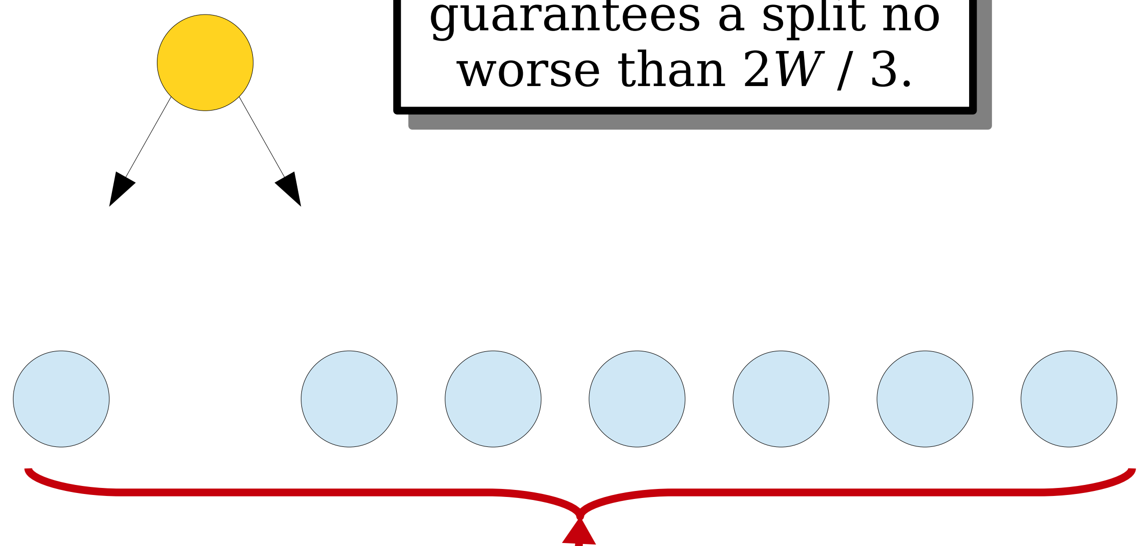
Model 2: Queries are sampled from a fixed, known probability distribution.

A ***weight-equalized tree*** is a BST where the root is chosen to minimize the weight difference between the left and right subtrees.

Theorem: In a weight-equalized 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 $2W / 3$.

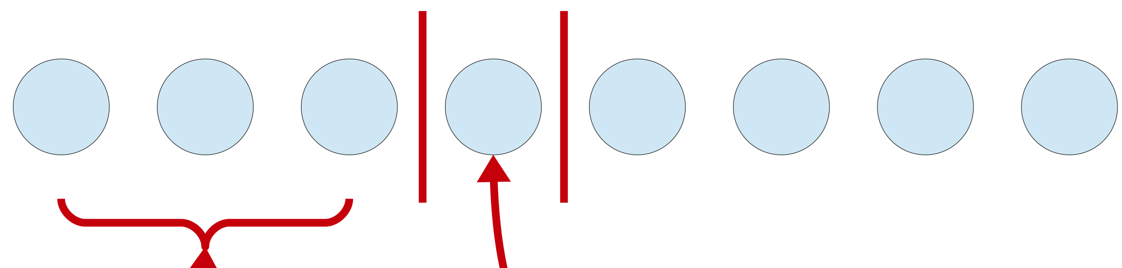
Model 2: Queries are sampled from a fixed, known probability distribution.

A ***weight-equalized tree*** is a BST where the root is chosen to minimize the weight difference between the left and right subtrees.

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

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

A ***weight-equalized tree*** is a BST where the root is chosen to minimize the weight difference between the left and right subtrees.

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

Theorem: The above bound is the tightest possible bound on the sizes of a node's two subtrees in a weight-equalized tree. (Prove this!)

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

Theorem: Weight-equalized trees have the entropy property.

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

Theorem: Weight-equalized trees have the entropy property.

Proof: The expected cost of a lookup in a weight-equalized 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-equalized tree containing x_i.

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

Theorem: Weight-equalized trees have the entropy property.

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

$$\sum_{i=1}^{n} \underbrace{p_i}_{\text{Probability of looking up } x_i} \cdot (1 + l_i)$$

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

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

Theorem: Weight-equalized trees have the entropy property.

Proof: The expected cost of a lookup in a weight-equalized 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-equalized tree containing x_i.

Probability of looking up x_i

Cost of looking up x_i: one comparison per layer descended, plus one final comparison confirming we have x_i.

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

Theorem: Weight-equalized trees have the entropy property.

Proof: The expected cost of a lookup in a weight-equalized 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-equalized tree containing x_i.

Focus on some key x_i and its depth l_i.

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

Theorem: Weight-equalized trees have the entropy property.

Proof: The expected cost of a lookup in a weight-equalized 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-equalized 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 $(2/3)^{l_i}$.

Theorem: In a weight-equalized 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.

Theorem: Weight-equalized trees have the entropy property.

Proof: The expected cost of a lookup in a weight-equalized 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-equalized 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 $(2/3)^{l_i}$. This means that $(2/3)^{l_i} \geq p_i$, so $l_i \leq \log_{3/2} (1/p_i)$.

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

Theorem: Weight-equalized trees have the entropy property.

Proof: The expected cost of a lookup in a weight-equalized 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-equalized 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 $(2/3)^{l_i}$. This means that $(2/3)^{l_i} \geq p_i$, so $l_i \leq \log_{3/2} (1/p_i)$. Therefore, we see that

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

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

Theorem: Weight-equalized trees have the entropy property.

Proof: The expected cost of a lookup in a weight-equalized 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-equalized 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 $(2/3)^{l_i}$. This means that $(2/3)^{l_i} \geq p_i$, so $l_i \leq \log_{3/2} (1/p_i)$. Therefore, we see that

$$\begin{aligned}\sum_{i=1}^{n} p_i \cdot (1 + l_i) &\leq \sum_{i=1}^{n} p_i \cdot \left(1 + \log_{3/2} \frac{1}{p_i}\right) \\ &= 1 + \sum_{i=1}^{n} \left(p_i \log_{3/2} \frac{1}{p_i}\right)\end{aligned}$$

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

Theorem: Weight-equalized trees have the entropy property.

Proof: The expected cost of a lookup in a weight-equalized 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-equalized 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 $(2/3)^{l_i}$. This means that $(2/3)^{l_i} \geq p_i$, so $l_i \leq \log_{3/2} (1/p_i)$. Therefore, we see that

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

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

Adding up p_i over a probability distribution.

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

Theorem: Weight-equalized trees have the entropy property.

Proof: The expected cost of a lookup in a weight-equalized 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-equalized tree containing x_i.

Focus on some key x_i and its depth l_i. After ta
the tree, the remaining probability mass is a
means that $(2/3)^{l_i} \geq p_i$, so $l_i \leq \log_{3/2} (1/p_i)$. There

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

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

Shannon entropy in a different log base; a constant multiple of Shannon entropy.

Adding up p_i over a probability distribution.

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

Theorem: Weight-equalized trees have the entropy property.

Proof: The expected cost of a lookup in a weight-equalized 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-equalized 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 $(2/3)^{l_i}$. This means that $(2/3)^{l_i} \geq p_i$, so $l_i \leq \log_{3/2} (1/p_i)$. Therefore, we see that

$$\begin{aligned}\sum_{i=1}^{n} p_i \cdot (1 + l_i) &\leq \sum_{i=1}^{n} p_i \cdot \left(1 + \log_{3/2} \frac{1}{p_i}\right) \\ &= 1 + \sum_{i=1}^{n} \left(p_i \log_{3/2} \frac{1}{p_i}\right) \\ &= O(1 + H). \blacksquare\end{aligned}$$

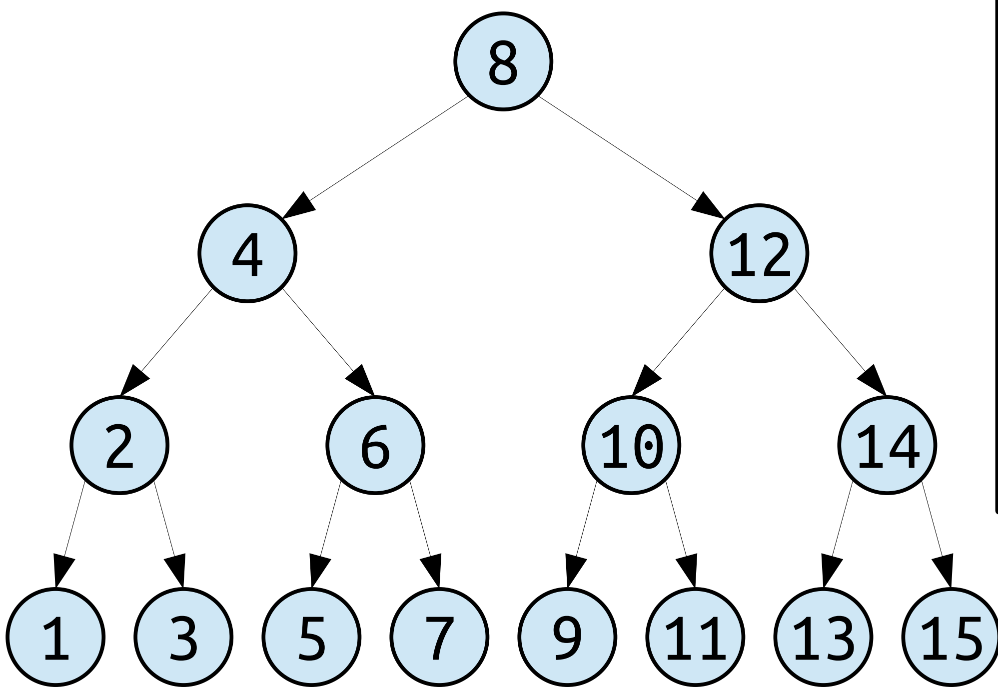
Model 2: Queries are sampled from a fixed, known probability distribution.

Theorem: Weight-equalized trees have the entropy property.

Fredman (1975): Weight-equalized 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.

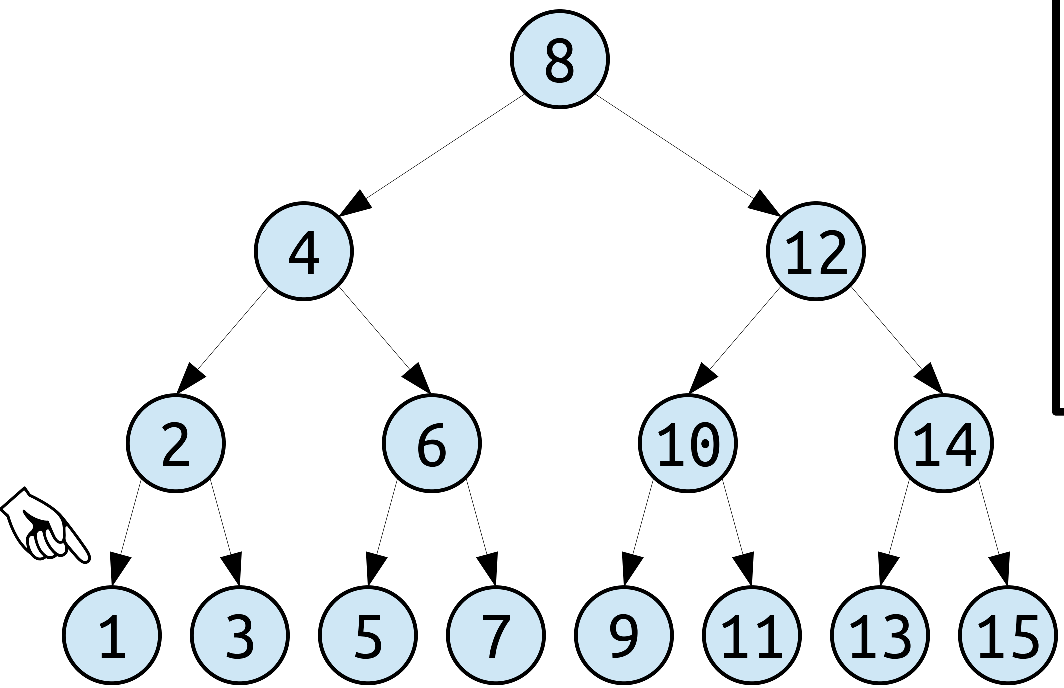


It's possible to visit all the nodes in *any* BST in sorted order in time O(n) via an inorder traversal, for an average lookup cost of O(1).

The balance property says the average cost of a lookup, across all nodes, is Ω(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 Ω(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.

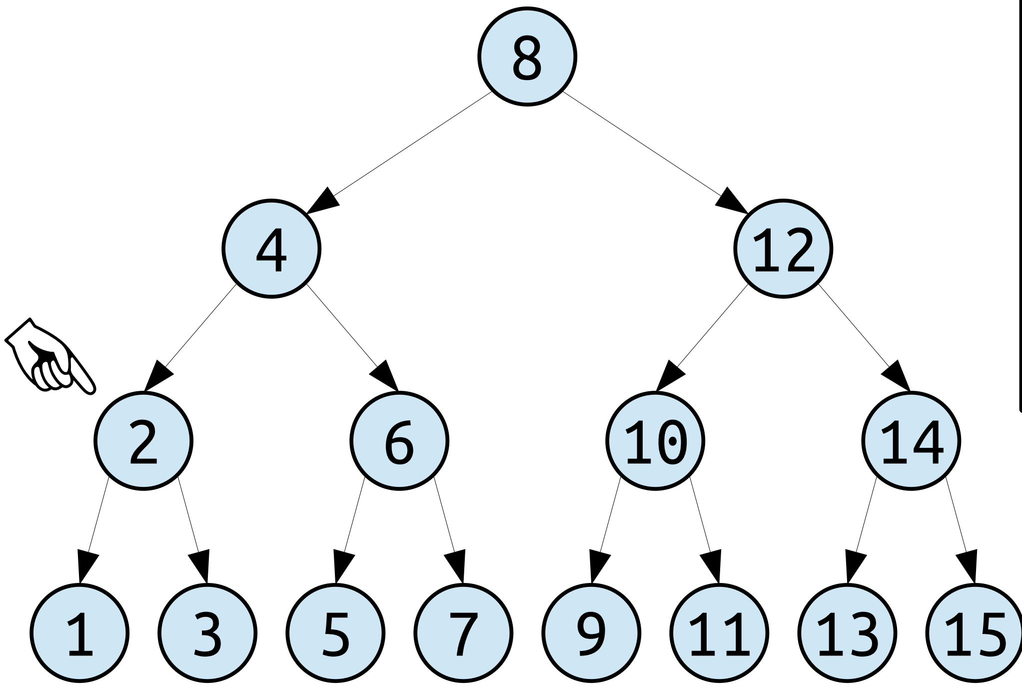


The balance and entropy properties assume our searches start at the top of the tree.

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.

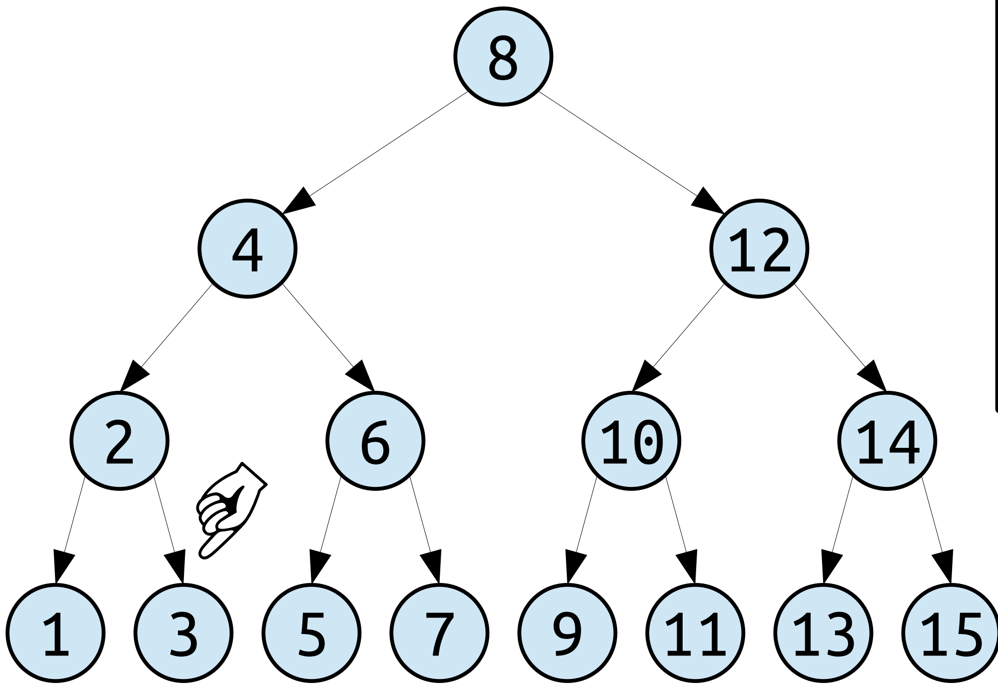


The balance and entropy properties assume our searches start at the top of the tree.

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.

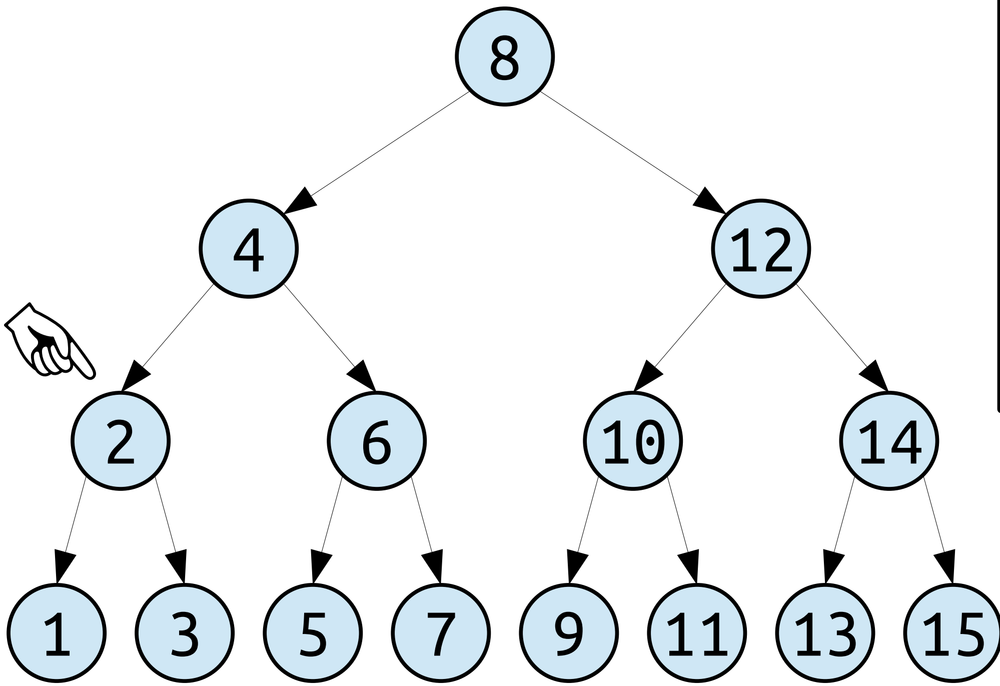


The balance and entropy properties assume our searches start at the top of the tree.

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.

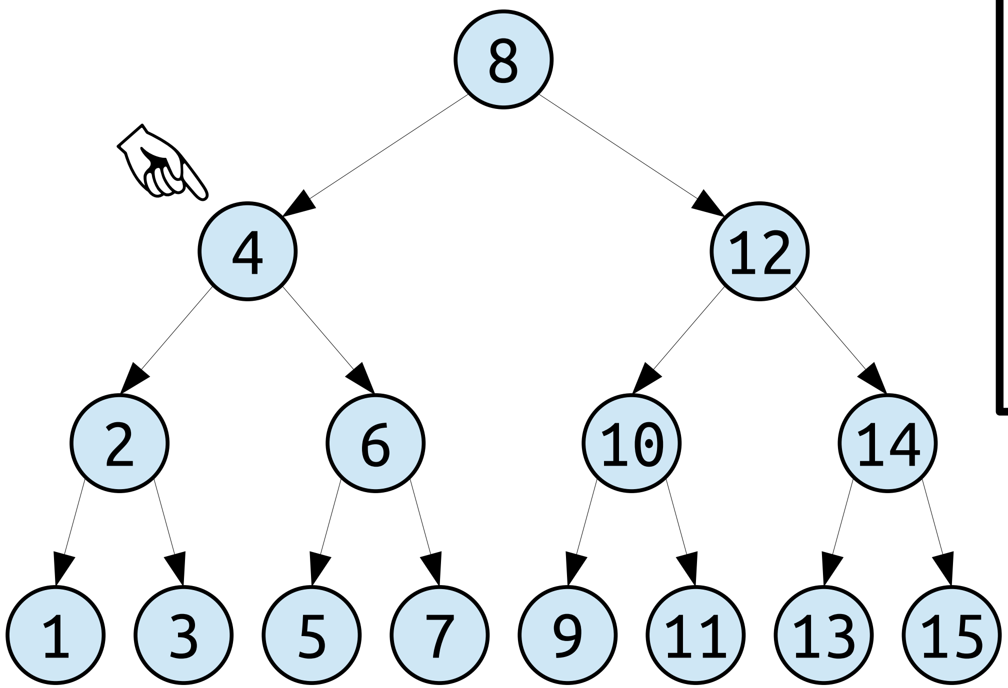


The balance and entropy properties assume our searches start at the top of the tree.

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.

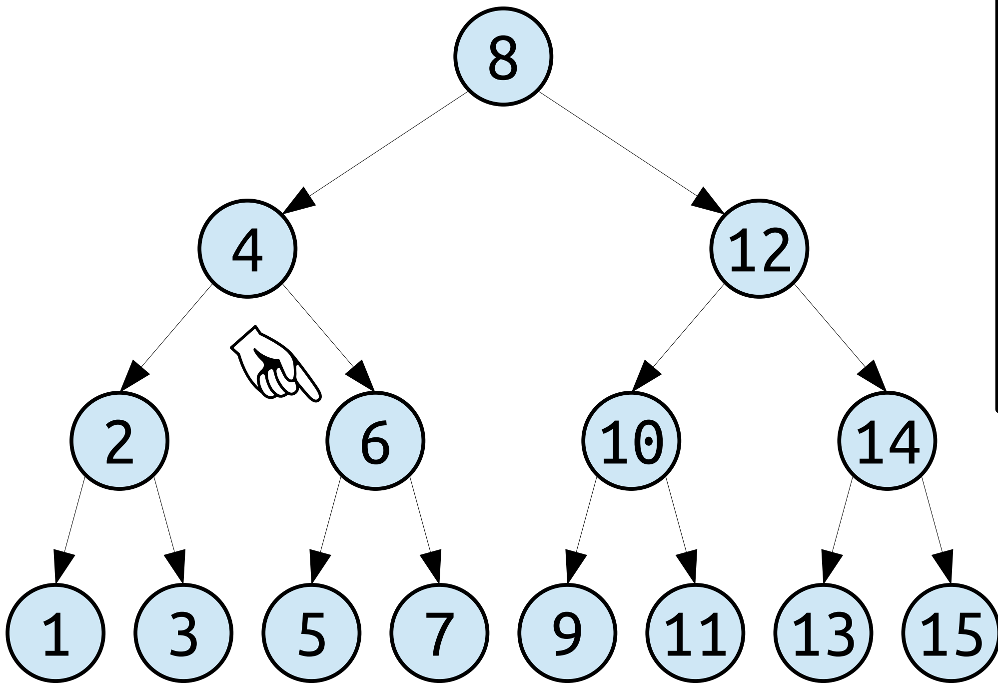


The balance and entropy properties assume our searches start at the top of the tree.

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.

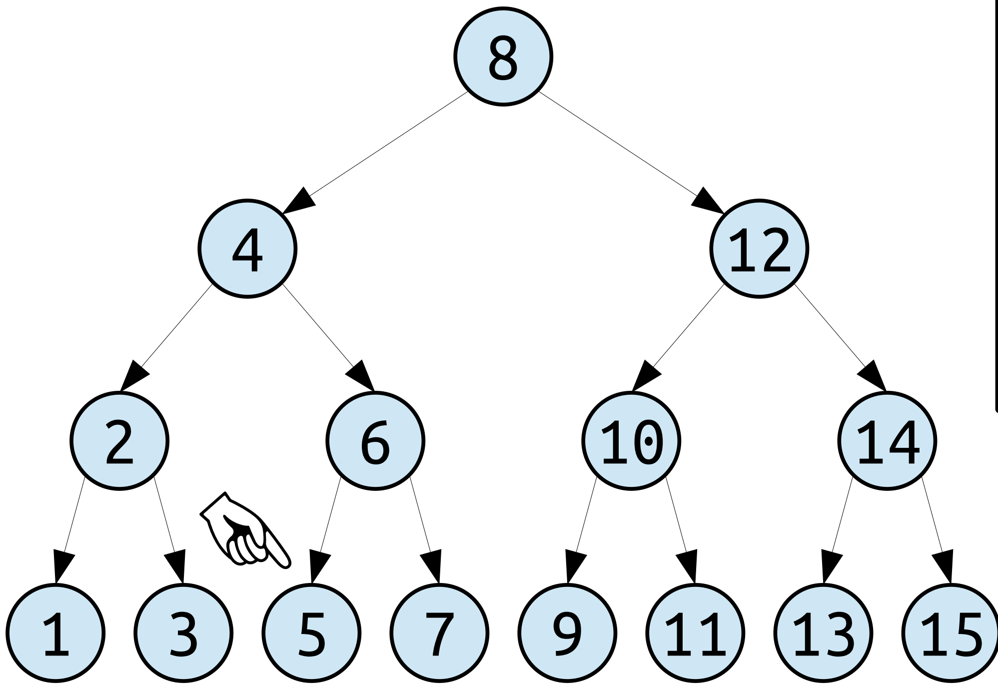


The balance and entropy properties assume our searches start at the top of the tree.

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.



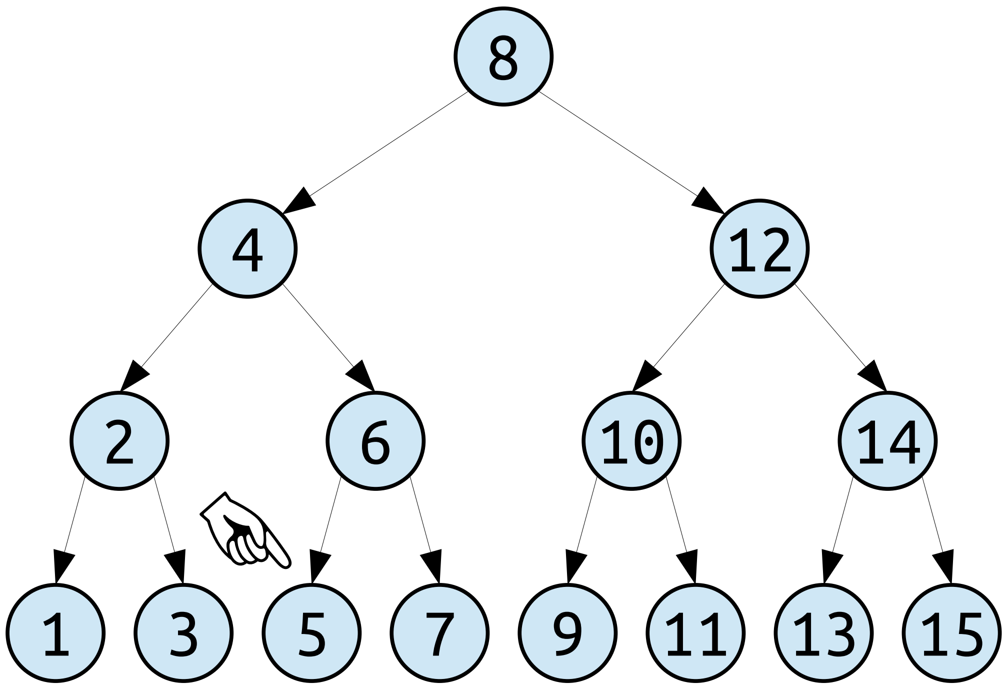
The balance and entropy properties assume our searches start at the top of the tree.

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.

Question: Can you build a binary search tree where the cost of a lookup depends on how similar the item looked up is to the most-recently-visited item?



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

11	13	17	19	21	23	29	31	37	41	43	47	53	59	61	67	71	73	79	83	89

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.

11	13	17	19	21	23	29	31	37	41	43	47	53	59	61	67	71	73	79	83	89

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.

11	13	17	19	21	23	29	31	37	41	43	47	53	59	61	67	71	73	79	83	89

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.

11	13	17	19	21	23	29	31	37	41	43	47	53	59	61	67	71	73	79	83	89

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.

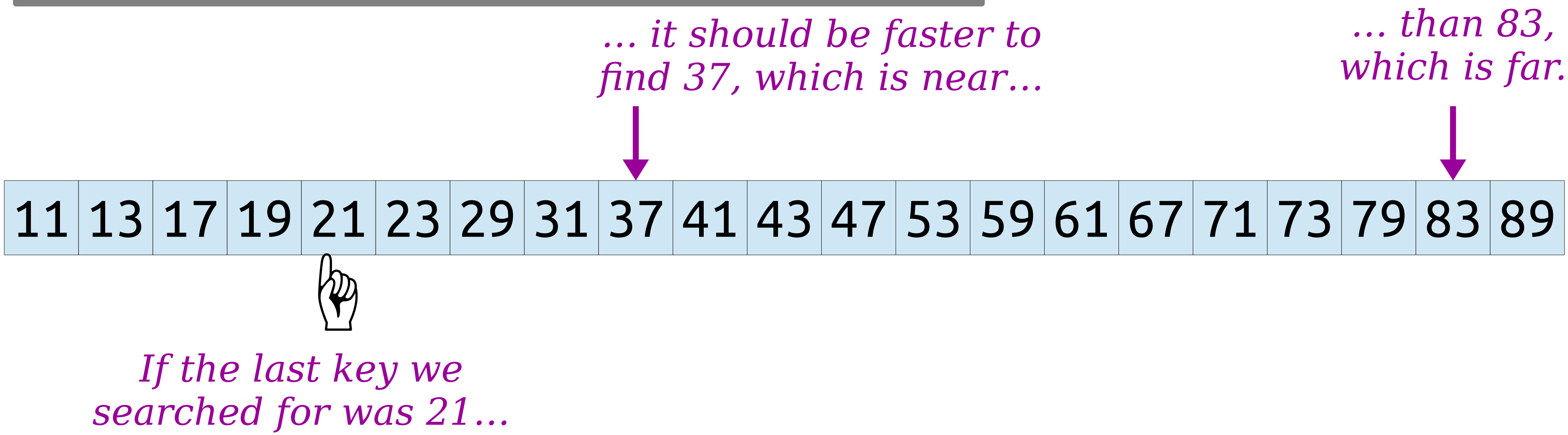
… it should be faster to find 37, which is near…

11	13	17	19	21	23	29	31	37	41	43	47	53	59	61	67	71	73	79	83	89

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.



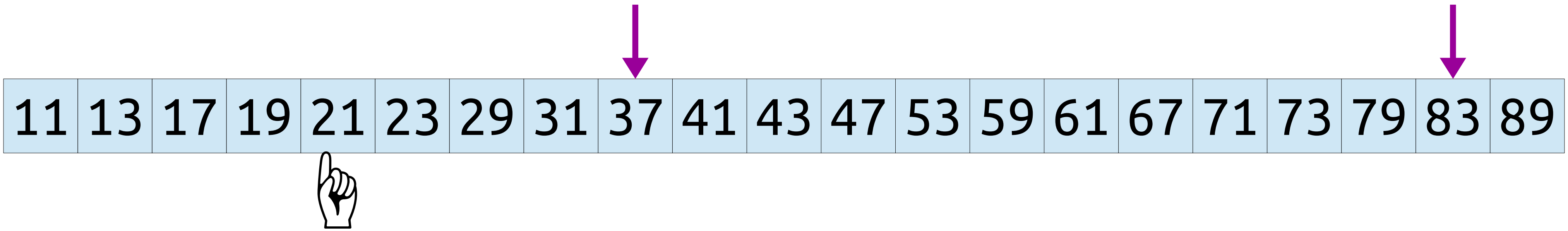
Model 3: Queries have ***spatial locality***. If a key is queried, keys with nearby values will likely be queried.

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

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

(The number of positions away the two elements are in the sorted sequence.)

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

… than 83, which is far.

11	13	17	19	21	23	29	31	37	41	43	47	53	59	61	67	71	73	79	83	89

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

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

(The number of positions away the two elements are in the sorted sequence.)

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

… than 83, which is far.

11	13	17	19	21	23	29	31	37	41	43	47	53	59	61	67	71	73	79	83	89

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

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

(The number of positions away the two elements are in the sorted sequence.)

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

… than 83, which is far.

11	13	17	19	21	23	29	31	37	41	43	47	53	59	61	67	71	73	79	83	89

If the last key we searched for was 21…

Formulate a hypothesis, but ***don't post anything in chat just yet***.

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 $O(\log \log \Delta)$?

(The number of positions away the two elements are in the sorted sequence.)

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

… than 83, which is far.

11	13	17	19	21	23	29	31	37	41	43	47	53	59	61	67	71	73	79	83	89

If the last key we searched for was 21…

Now, ***private chat me your best guess***. Not sure? Just answer "??."

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

(The number of positions away the two elements are in the sorted sequence.)

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

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

… than 83, which is far.

11	13	17	19	21	23	29	31	37	41	43	47	53	59	61	67	71	73	79	83	89

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

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

Idea: Just do a simple linear scan.

11	13	17	19	21	23	29	31	37	41	43	47	53	59	61	67	71	73	79	83	89

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 $O(\log \log \Delta)$?

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.

11	13	17	19	21	23	29	31	37	41	43	47	53	59	61	67	71	73	79	83	89
					1	2		4				8								

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 $O(\log \log \Delta)$?

11	13	17	19	21	23	29	31	37	41	43	47	53	59	61	67	71	73	79	83	89

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 Δ)?~~

$\Delta = O(n)$.

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

(***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.)

11	13	17	19	21	23	29	31	37	41	43	47	53	59	61	67	71	73	79	83	89

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.

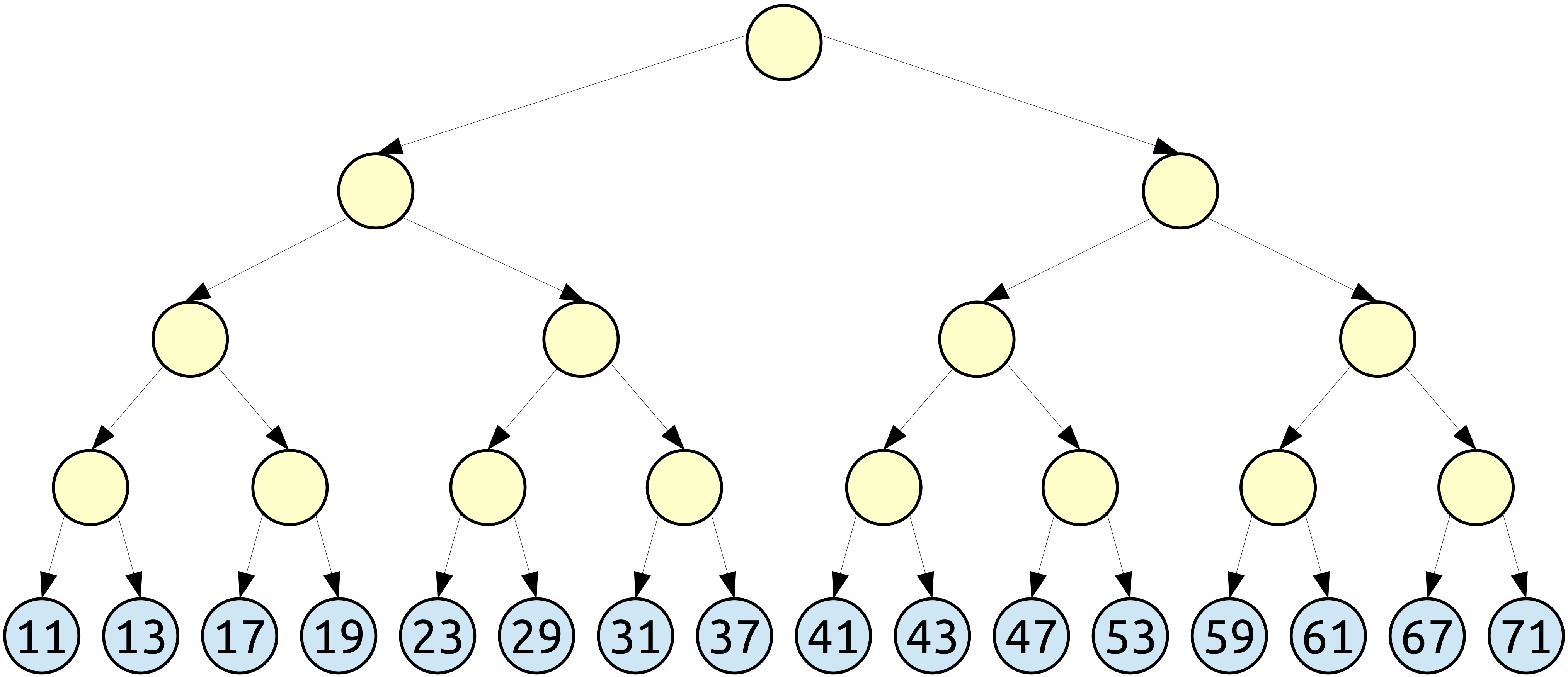
Let $\Delta = |rank(x_2) - rank(x_1)|$.

Can we do the search in time O(Δ)?
How about time O(log Δ)?
~~How about time O(log log Δ)?~~

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

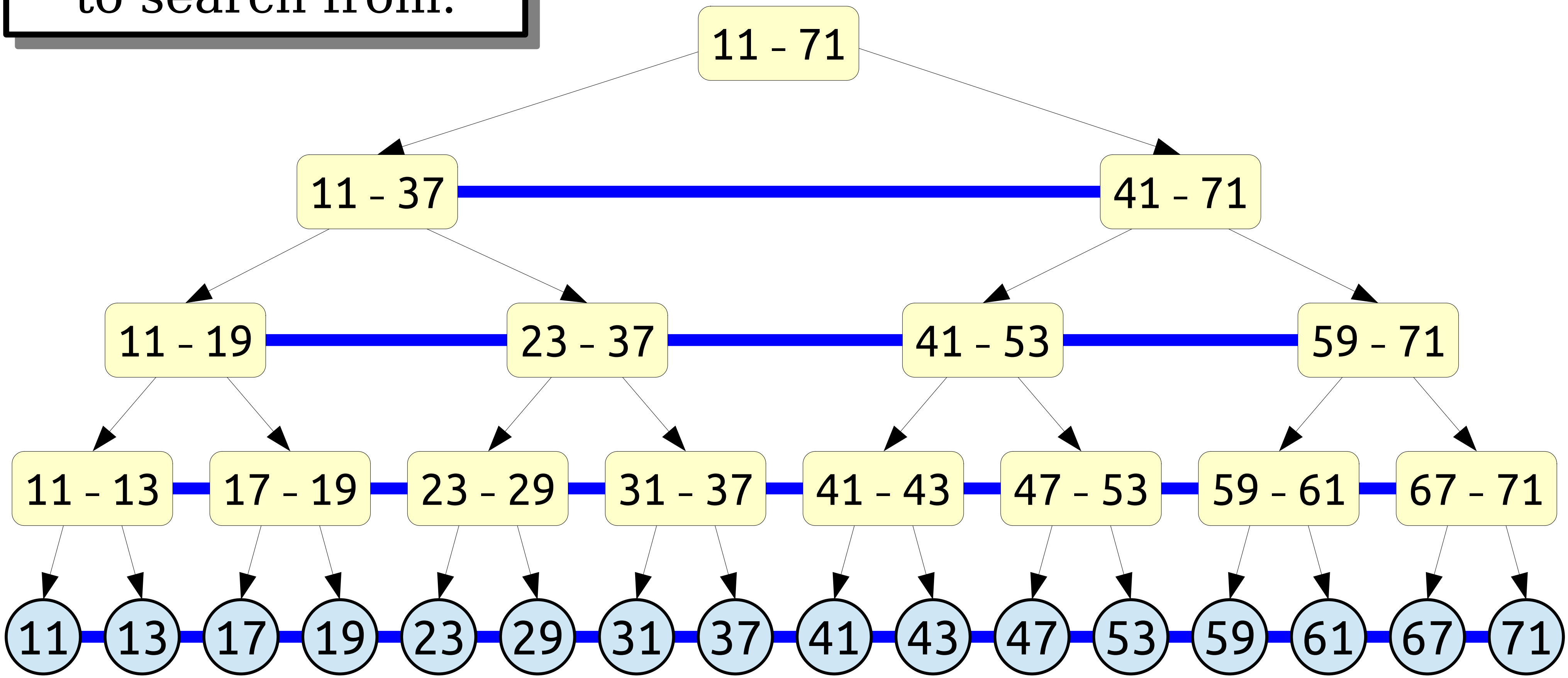
11	13	17	19	21	23	29	31	37	41	43	47	53	59	61	67	71	73	79	83	89

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

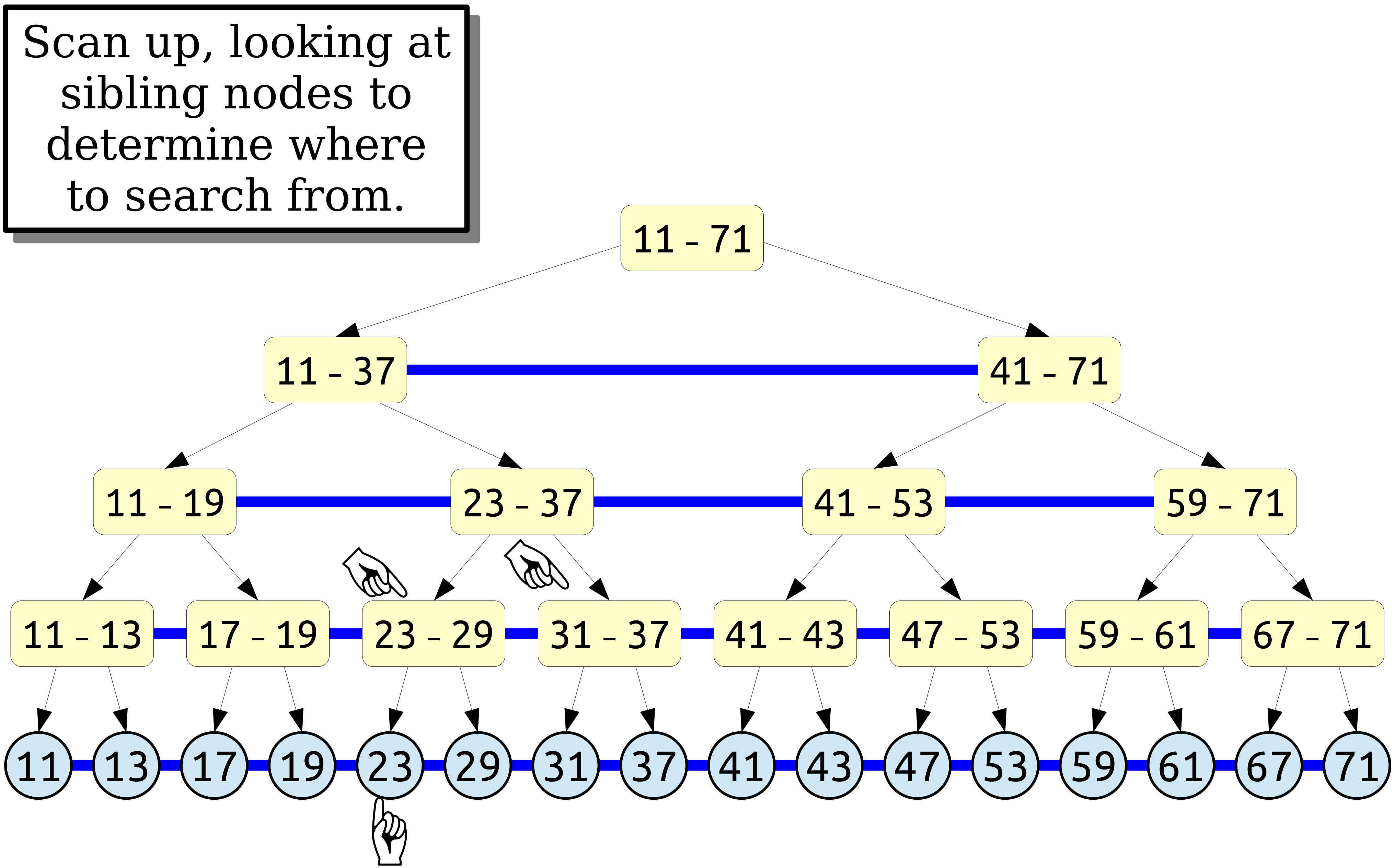


Model 3: Queries have ***spatial locality***. If a key is queried, keys with nearby values will likely be queried.

Scan up, looking at sibling nodes to determine where to search from.

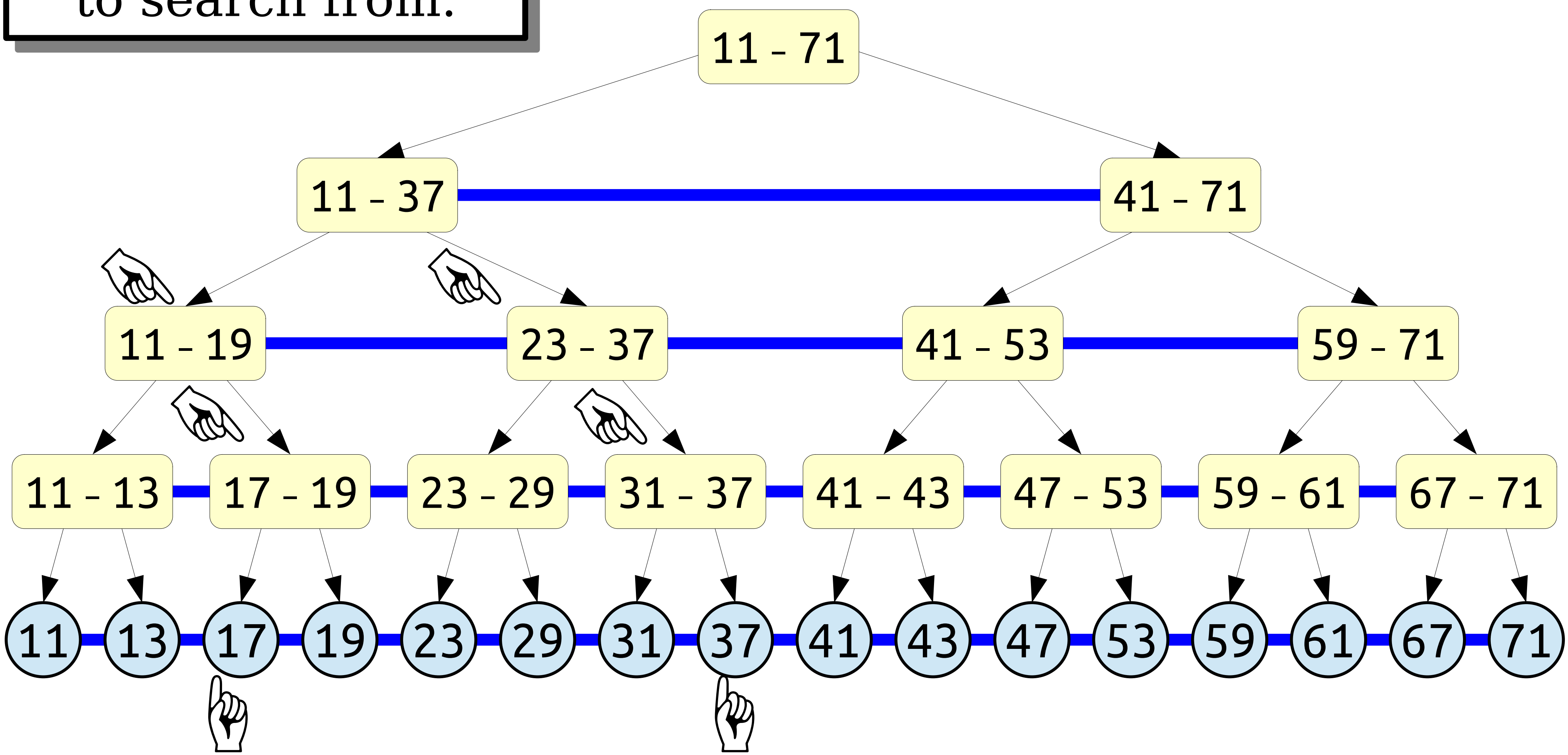


Model 3: Queries have ***spatial locality***. If a key is queried, keys with nearby values will likely be queried.



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

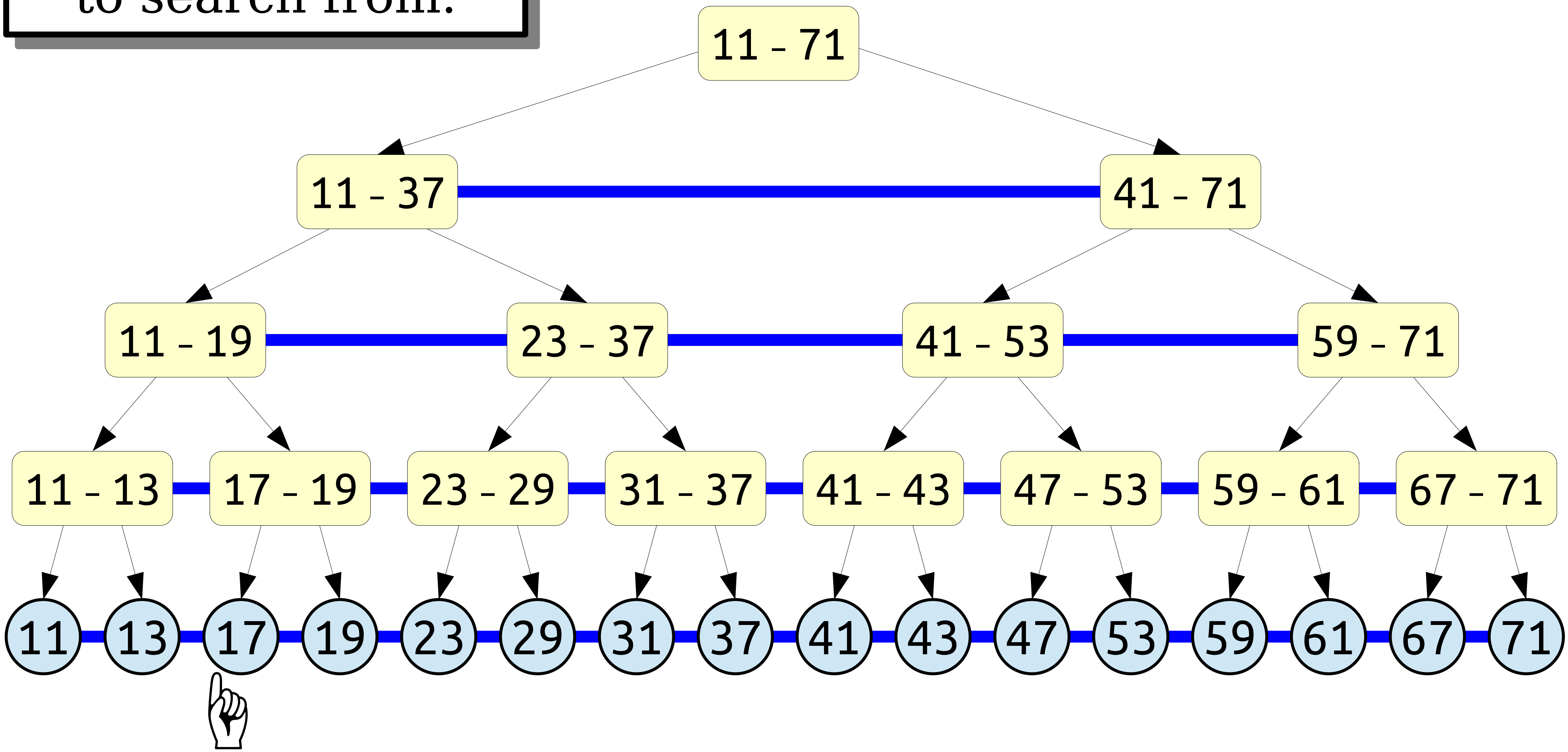
Scan up, looking at sibling nodes to determine where to search from.



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

Scan up, looking at sibling nodes to determine where to search from.

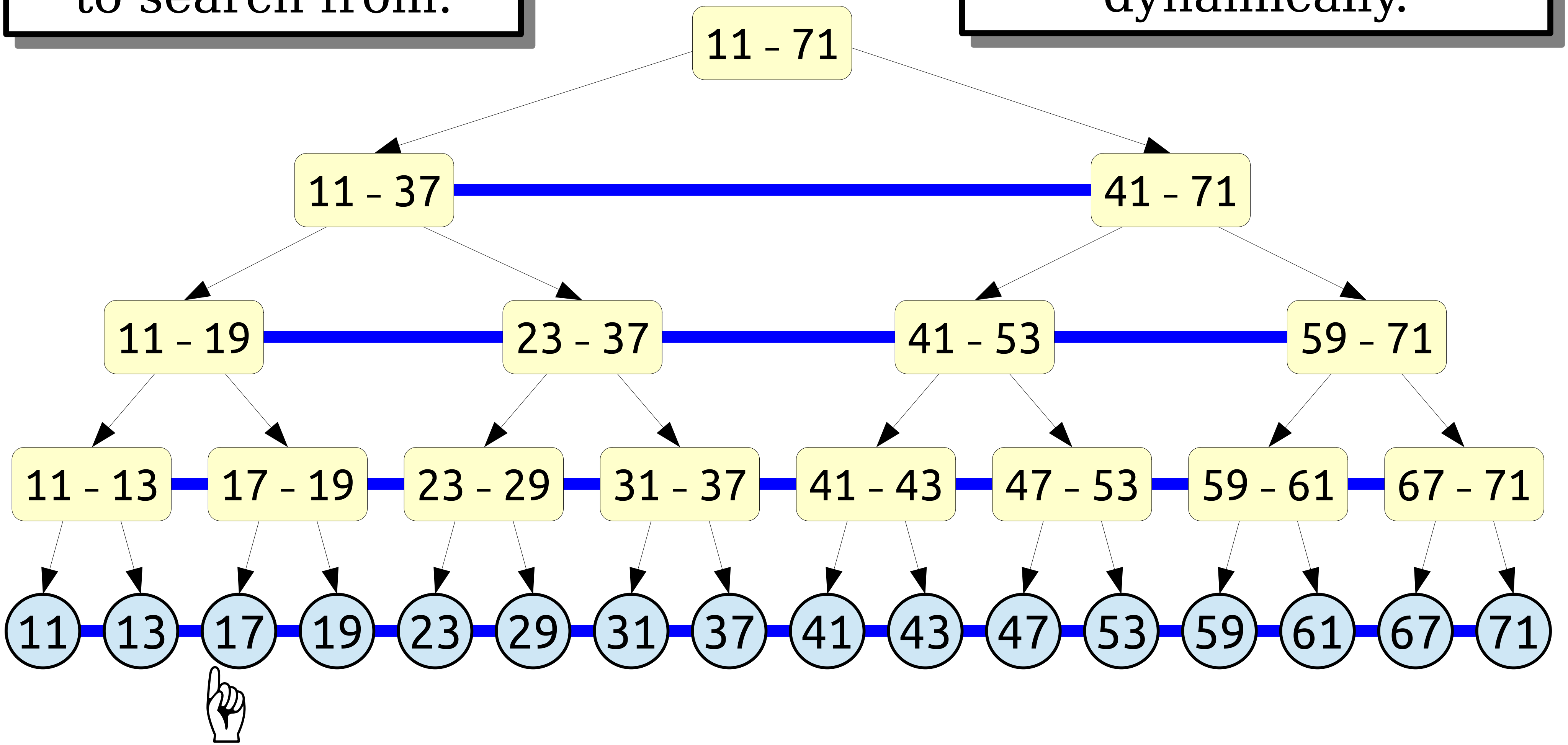
Claim: This simulates our earlier search. Runtime is $O(\log \Delta)$.



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

Scan up, looking at sibling nodes to determine where to search from.

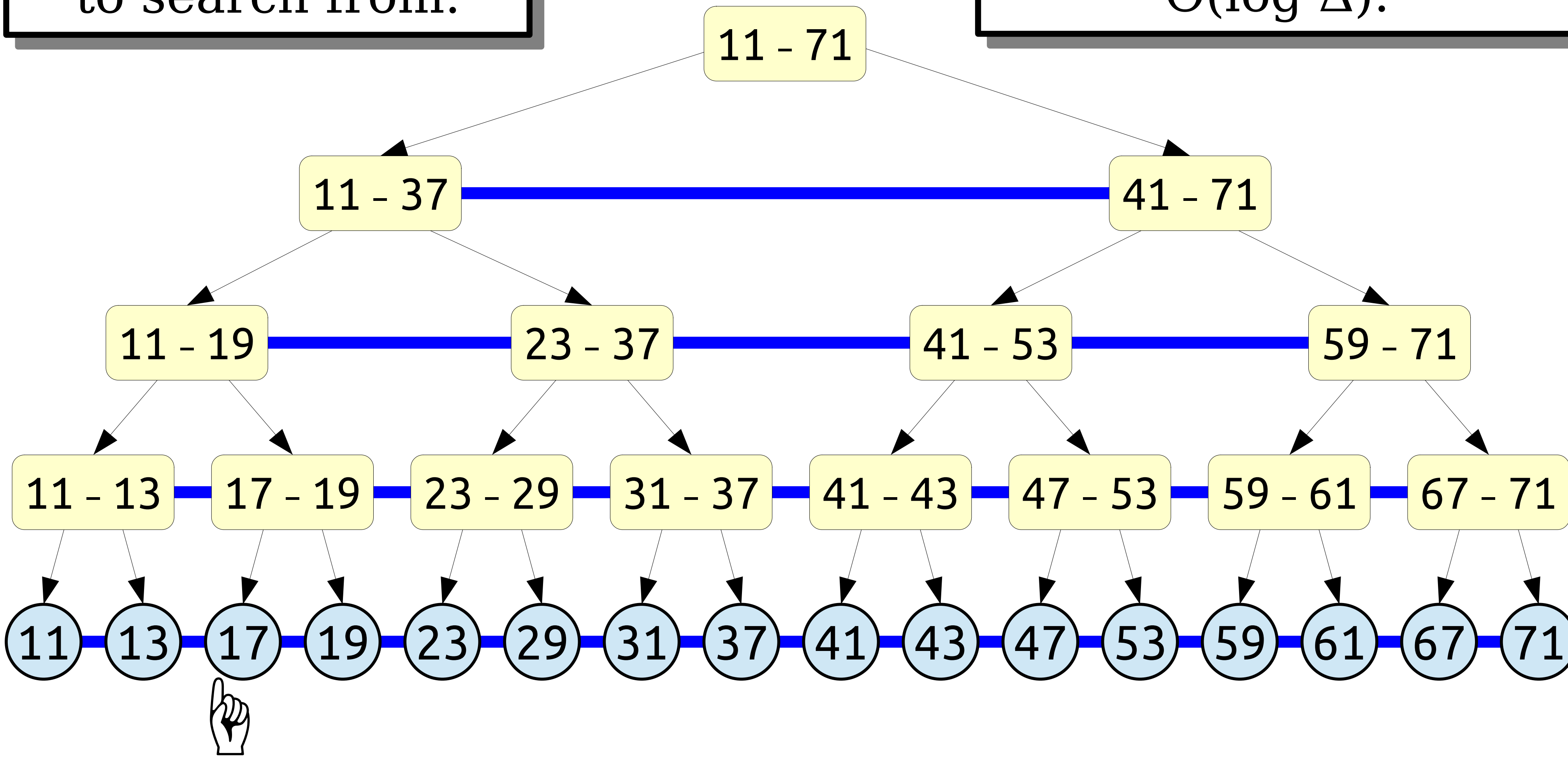
The ***level-linked red/black tree*** implements this dynamically.



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

Scan up, looking at sibling nodes to determine where to search from.

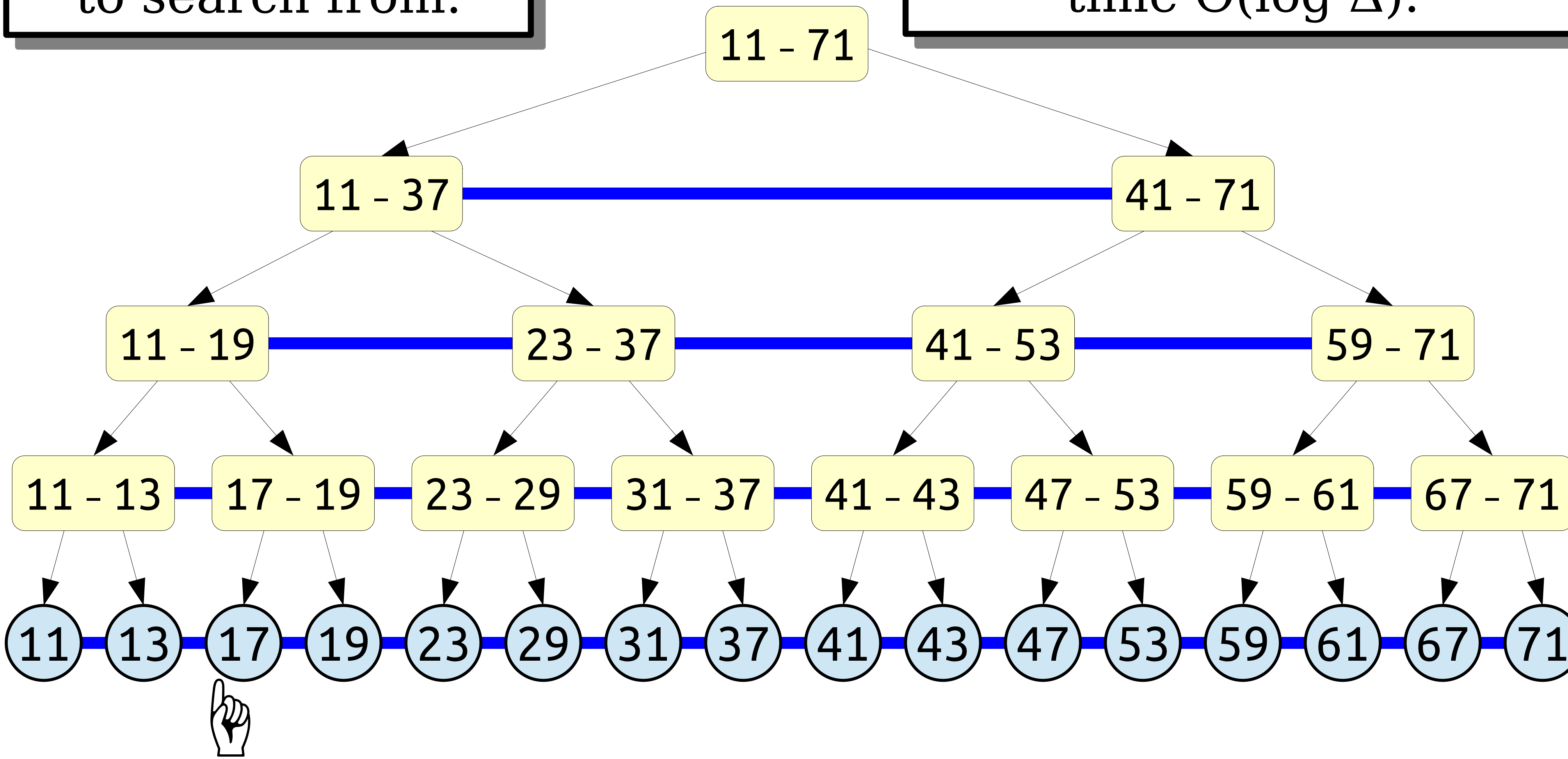
Great exercise: Do this with skiplists in (expected) time O(log Δ).



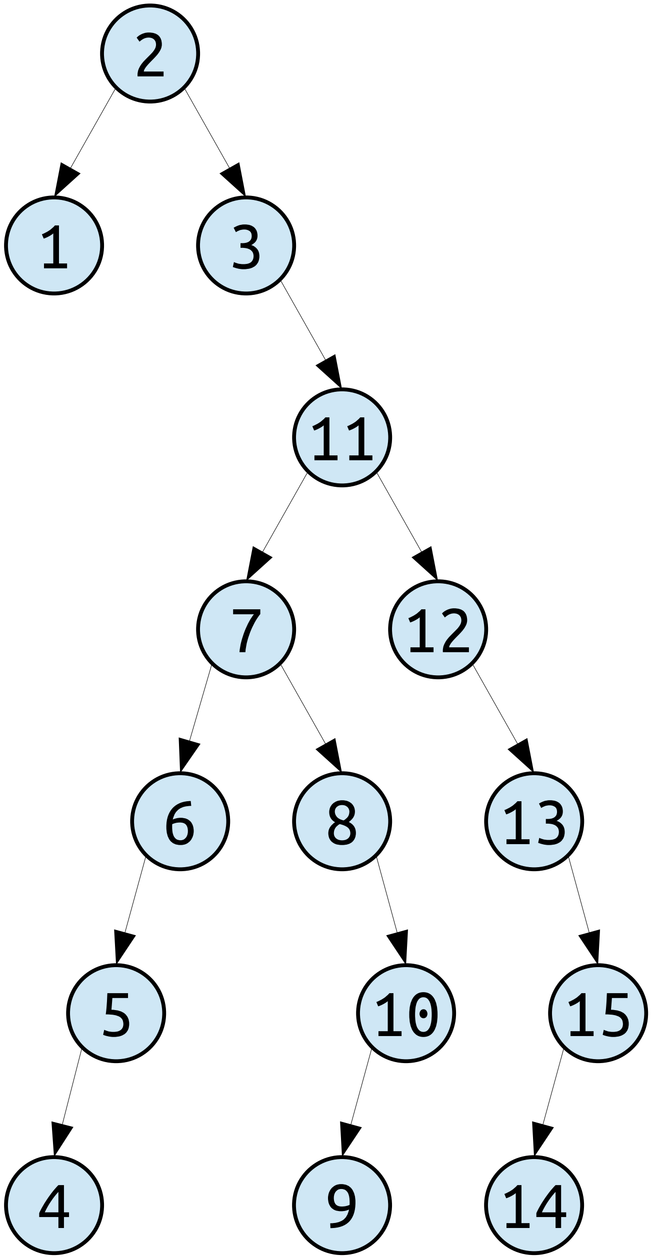
Model 3: Queries have ***spatial locality***. If a key is queried, keys with nearby values will likely be queried.

Scan up, looking at sibling nodes to determine where to search from.

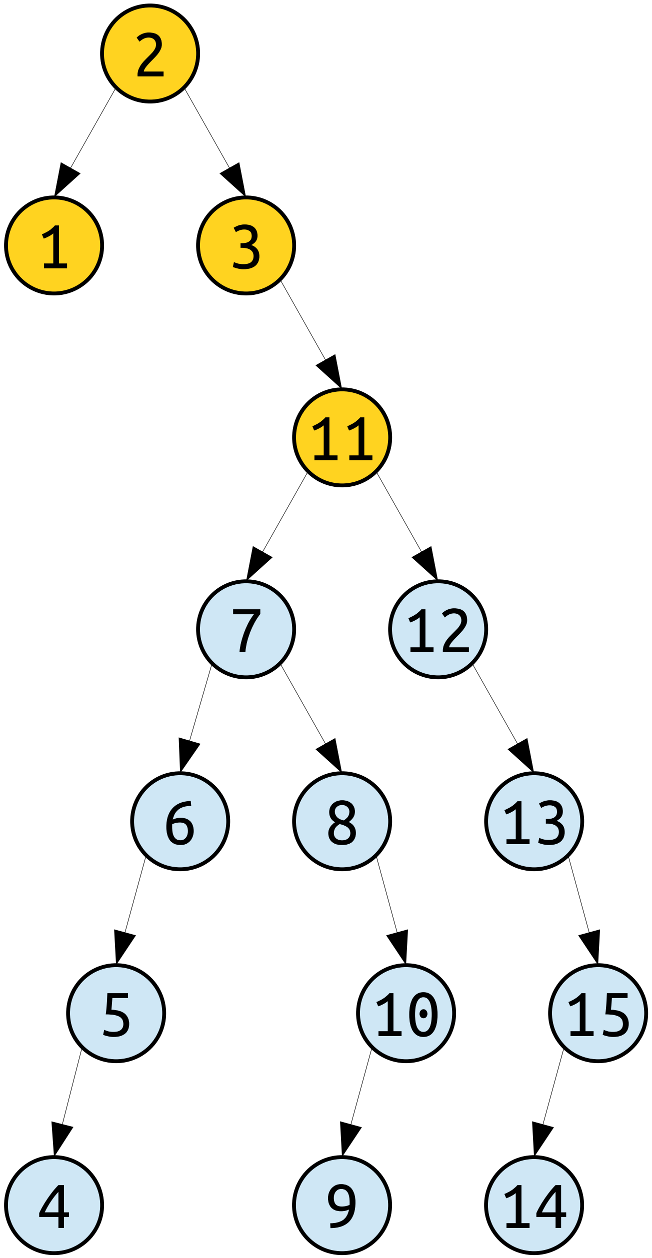
A BST has the ***dynamic finger property*** if lookups take (amortized) time O(log Δ).



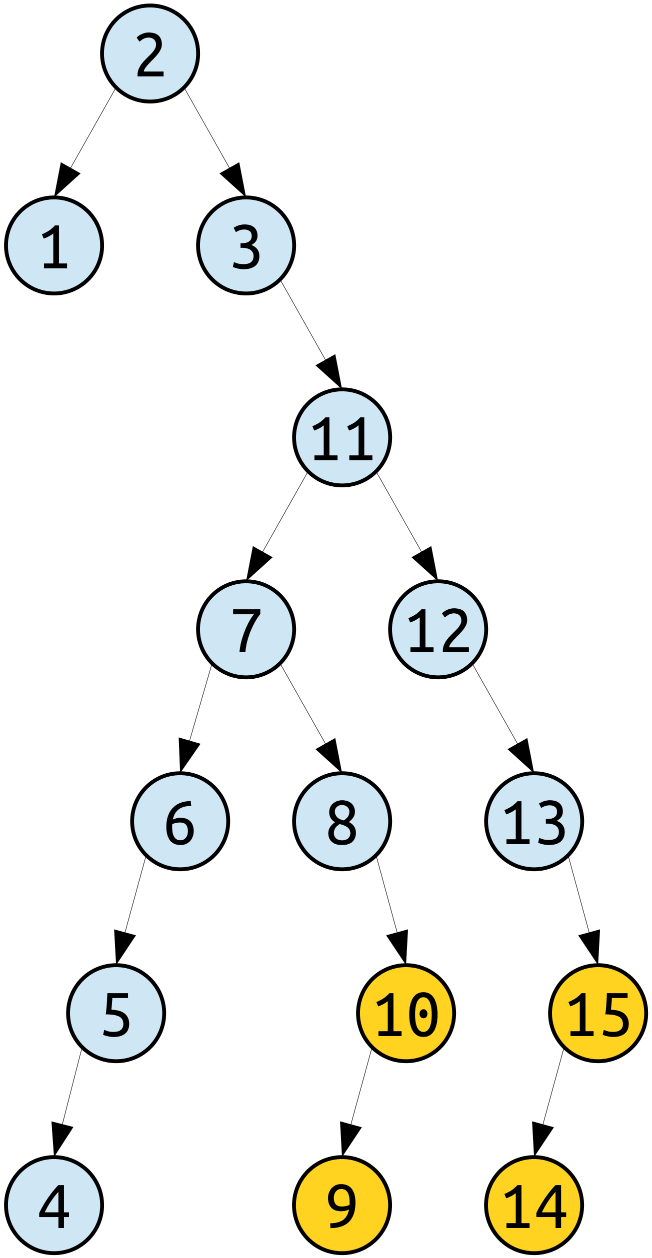
Model 3: Queries have ***spatial locality***. If a key is queried, keys with nearby values will likely be queried.



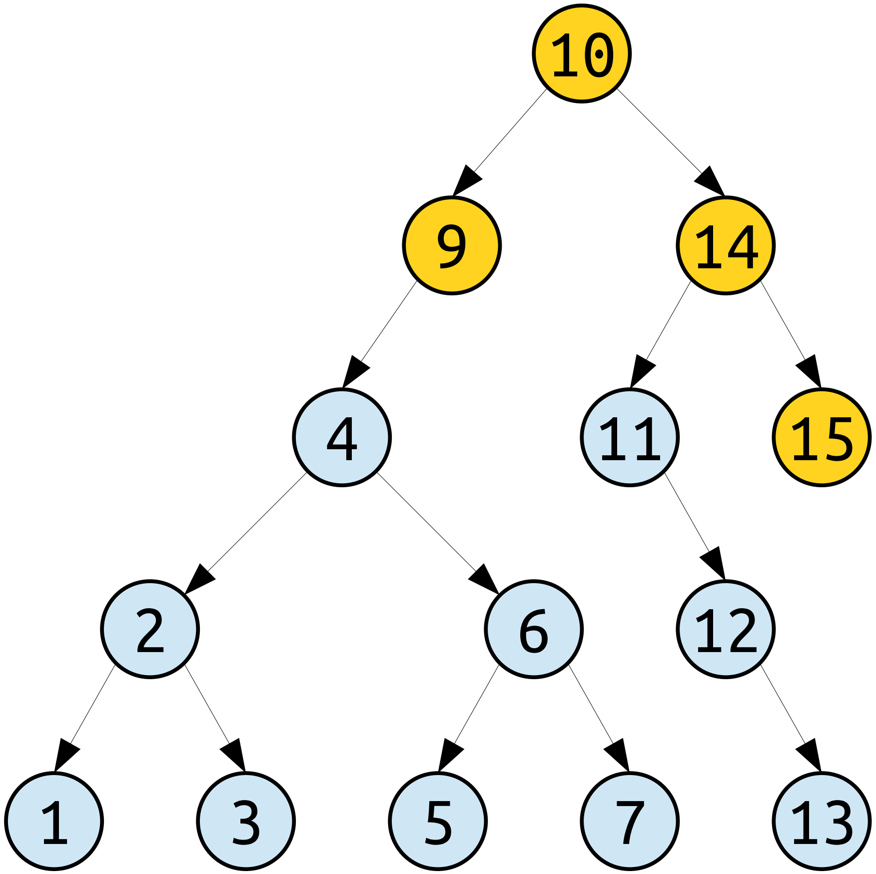
Model 4: Queries have ***temporal locality***. If a key is queried, it's likely going to be queried again soon.



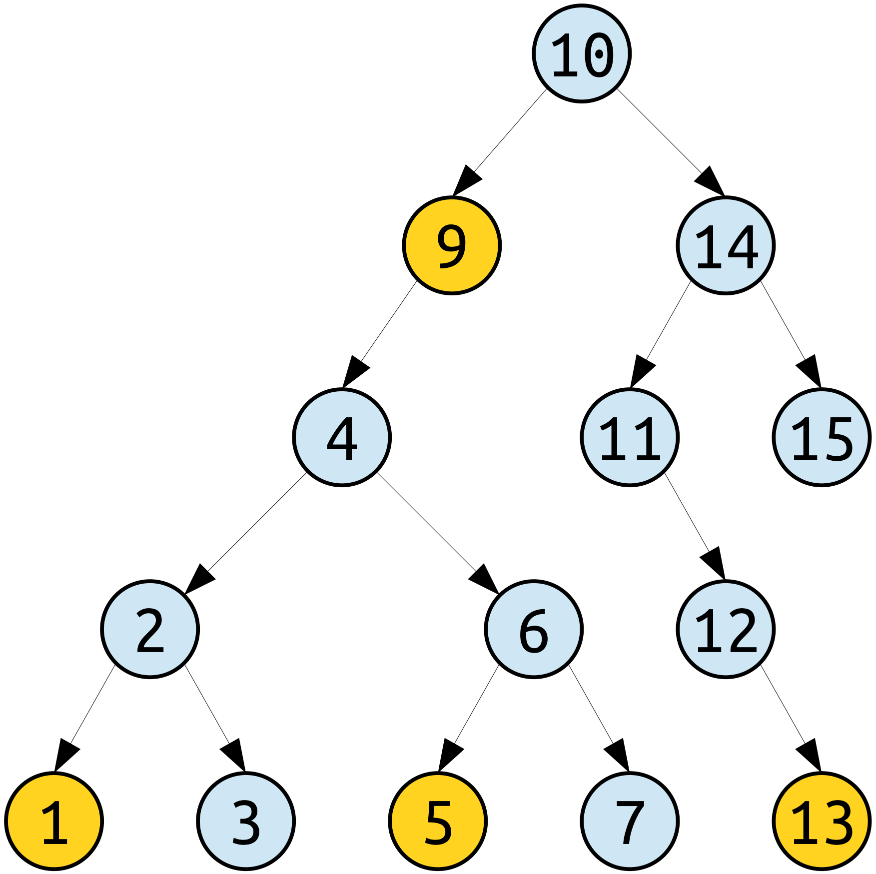
Model 4: Queries have ***temporal locality***. If a key is queried, it's likely going to be queried again soon.



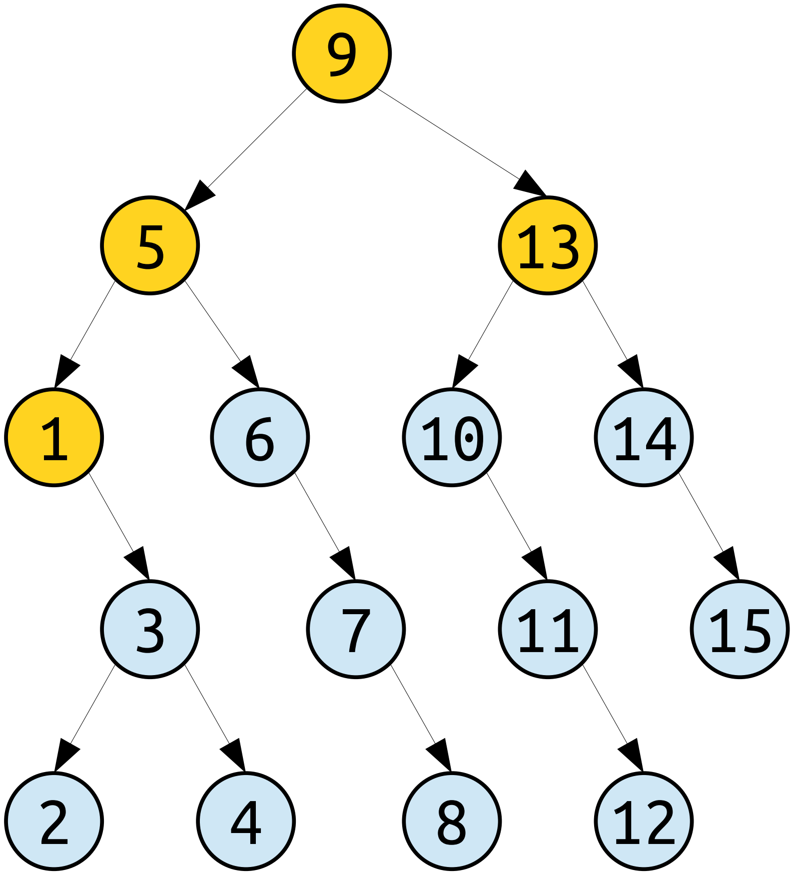
Model 4: Queries have ***temporal locality***. If a key is queried, it's likely going to be queried again soon.



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

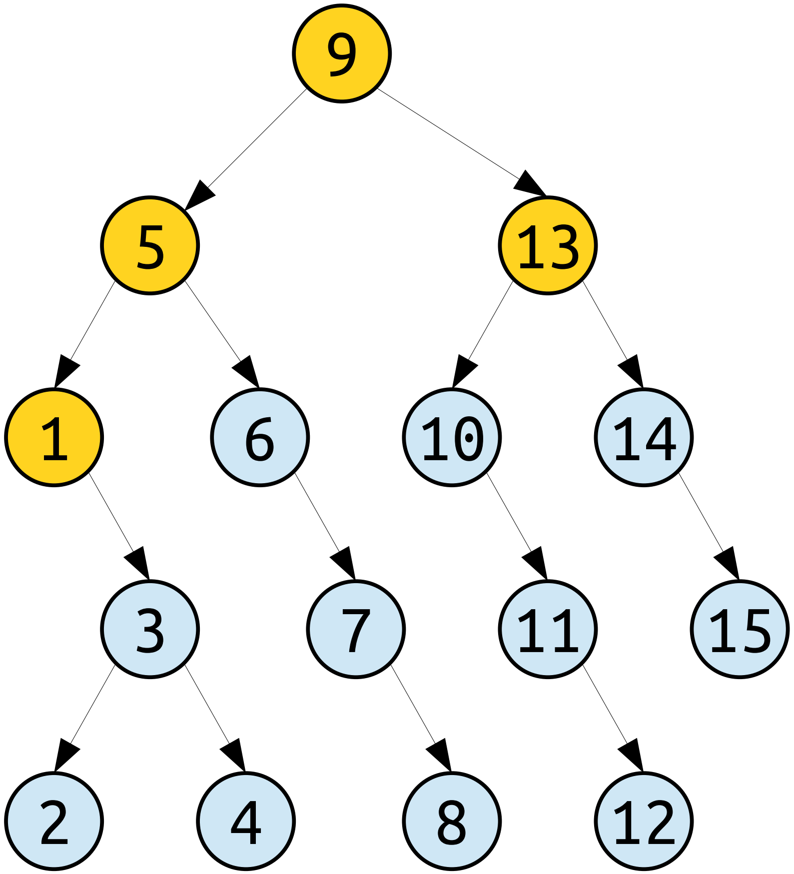


Model 4: Queries have ***temporal locality***. If a key is queried, it's likely going to be queried again soon.



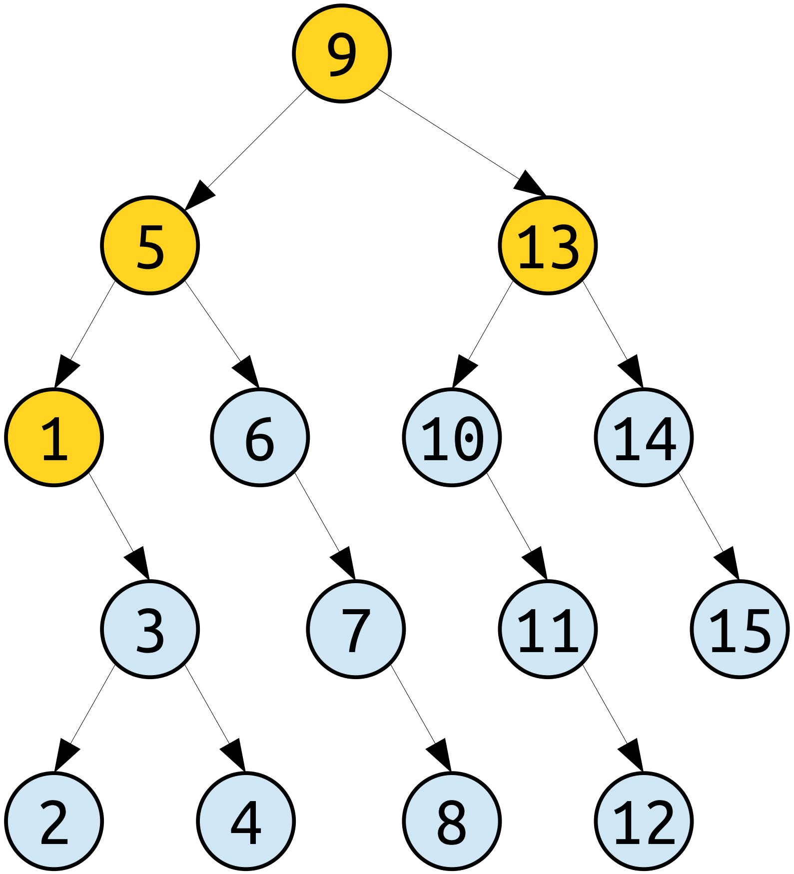
Goal: If only t elements are "hot" at a particular time, make accesses to those "hot" elements take time O(log t), not O(log n).

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.

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

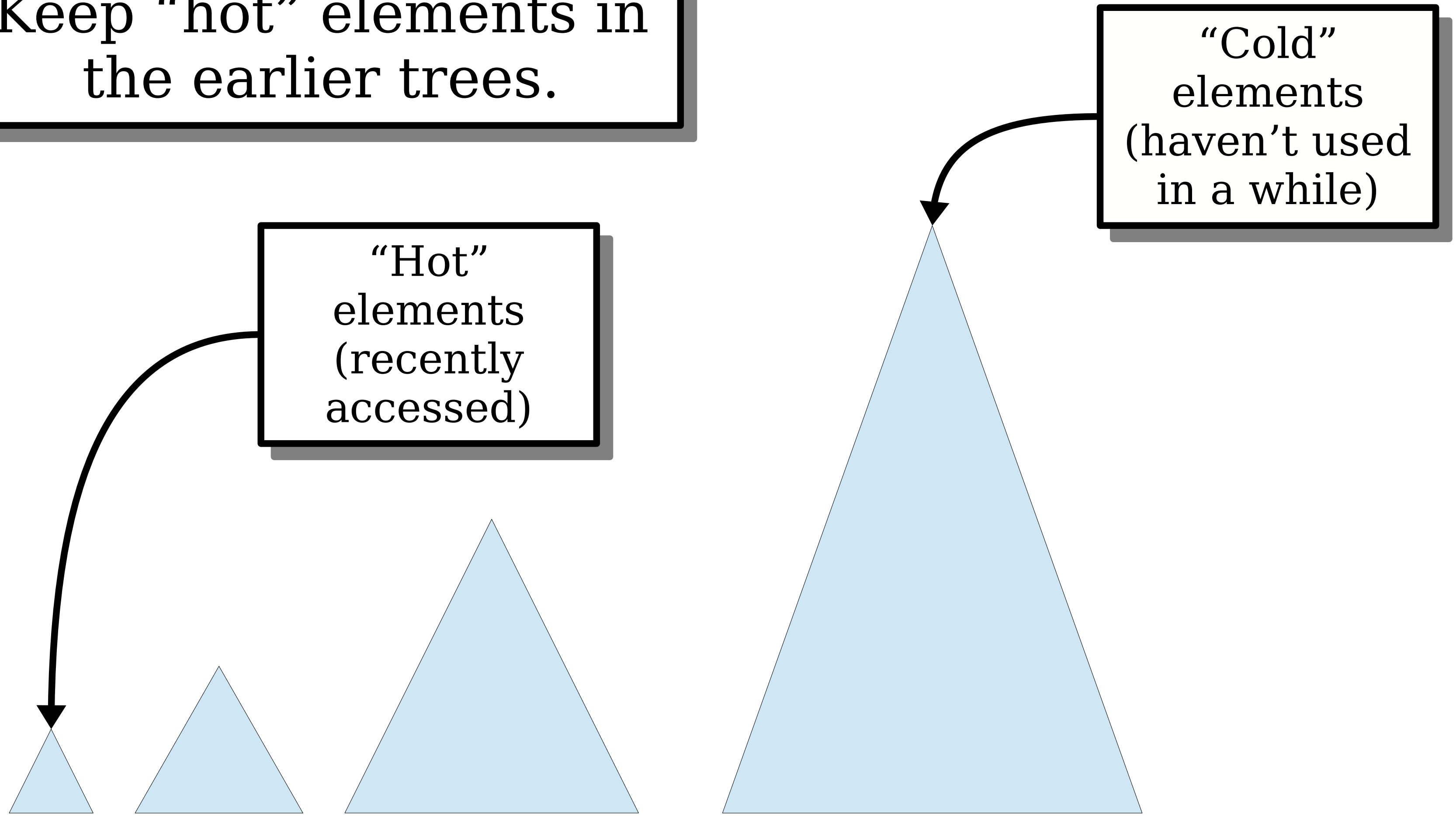


Intuition: Any tree structure with a fixed shape is going to have a hard time making these queries fast.

Idea: What if we move elements around?

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.



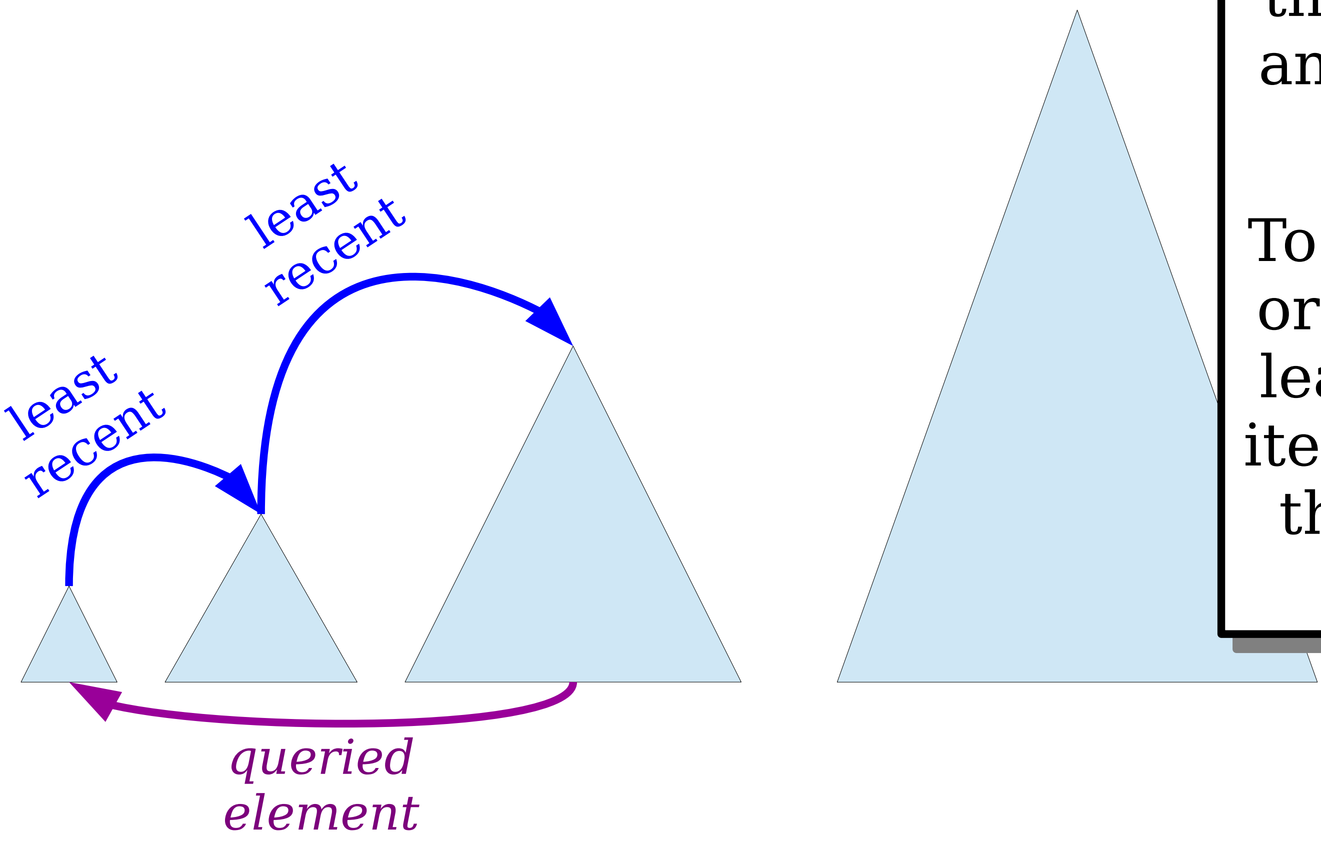
Model 4: Queries have ***temporal locality***. If a key is queried, it's likely going to be queried again soon.

Intuition: Use a sequence of trees. Keep "hot" elements in the earlier trees.

To look up an element, search each tree in order of size until you find it.

Then, remove it from the tree you found it in and insert it to the first tree.

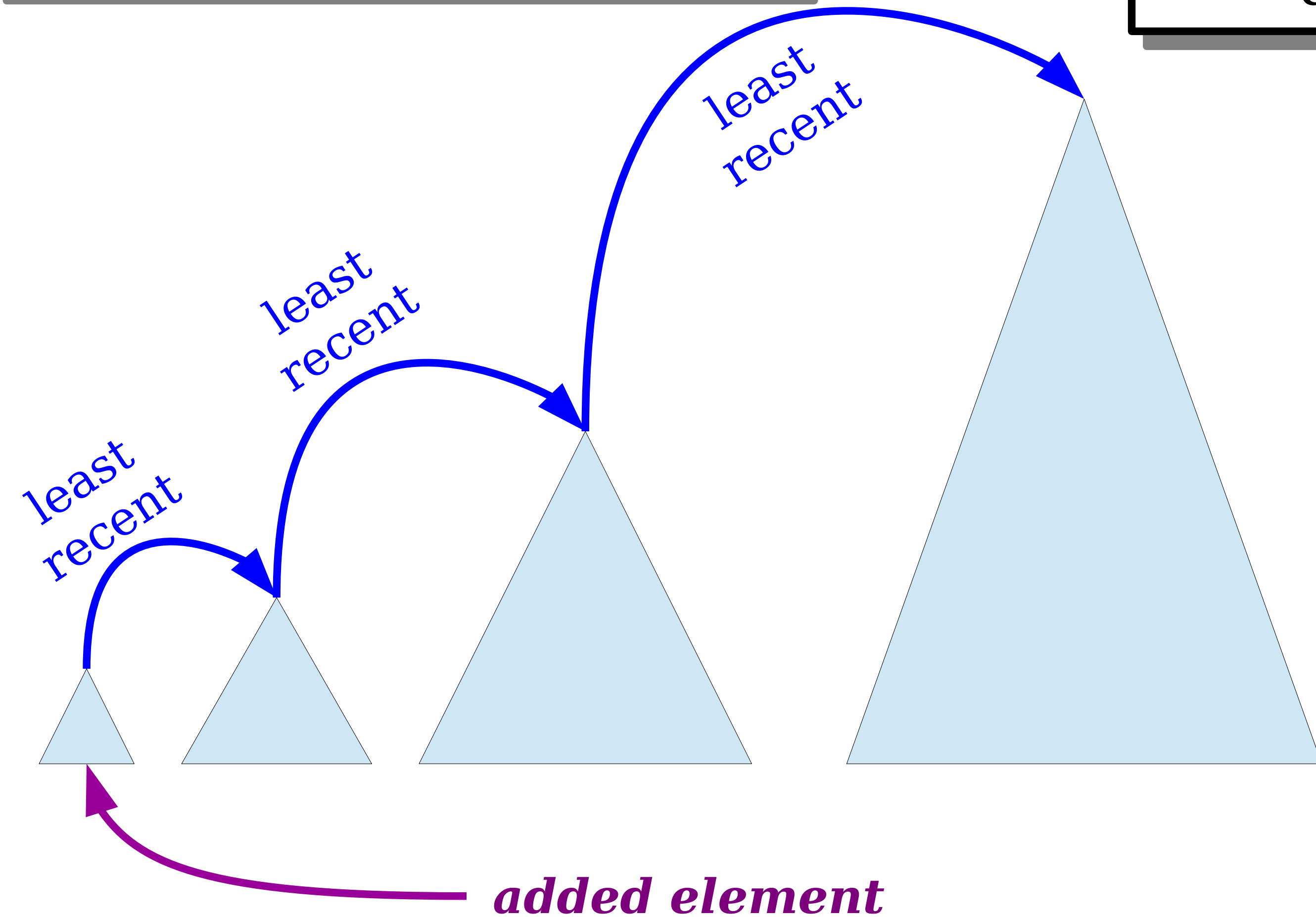
To fill the gap left in the original tree, move the least-recently-accessed item from each tree into the next tree until the gap is filled.



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.

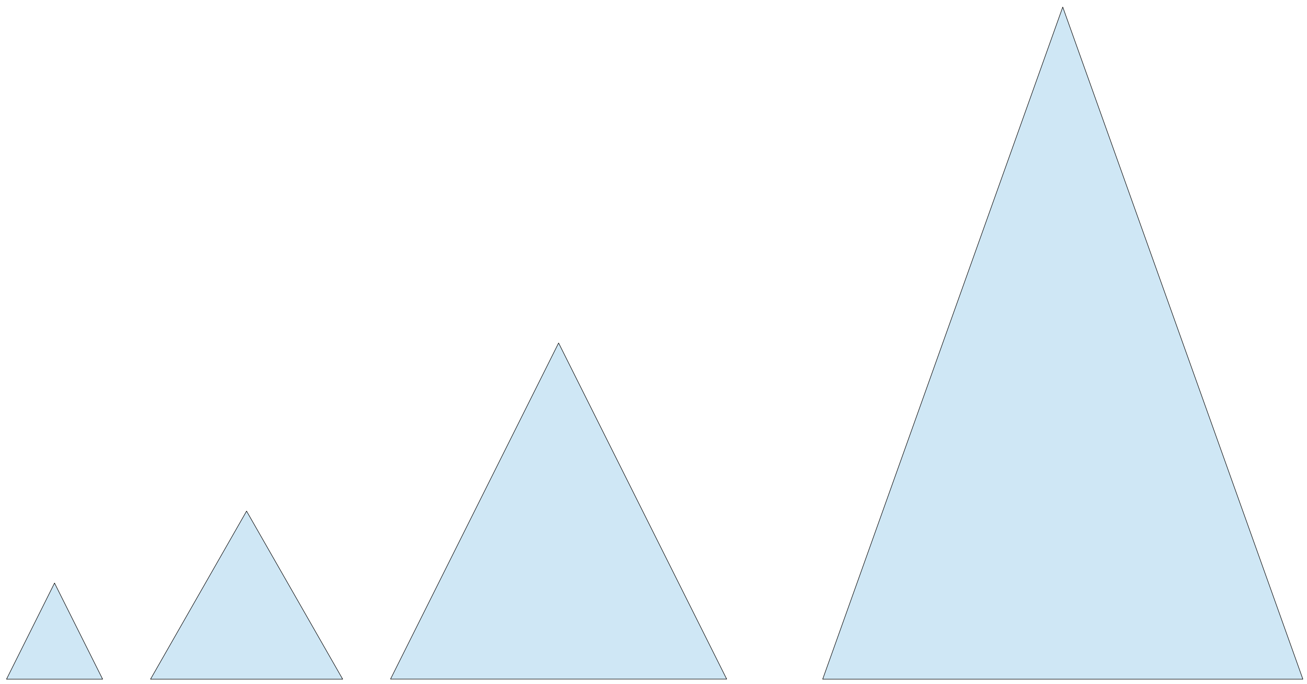
To insert an element, put it in the first tree. Then, repeatedly kick the oldest element out of each tree and into the next.



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?

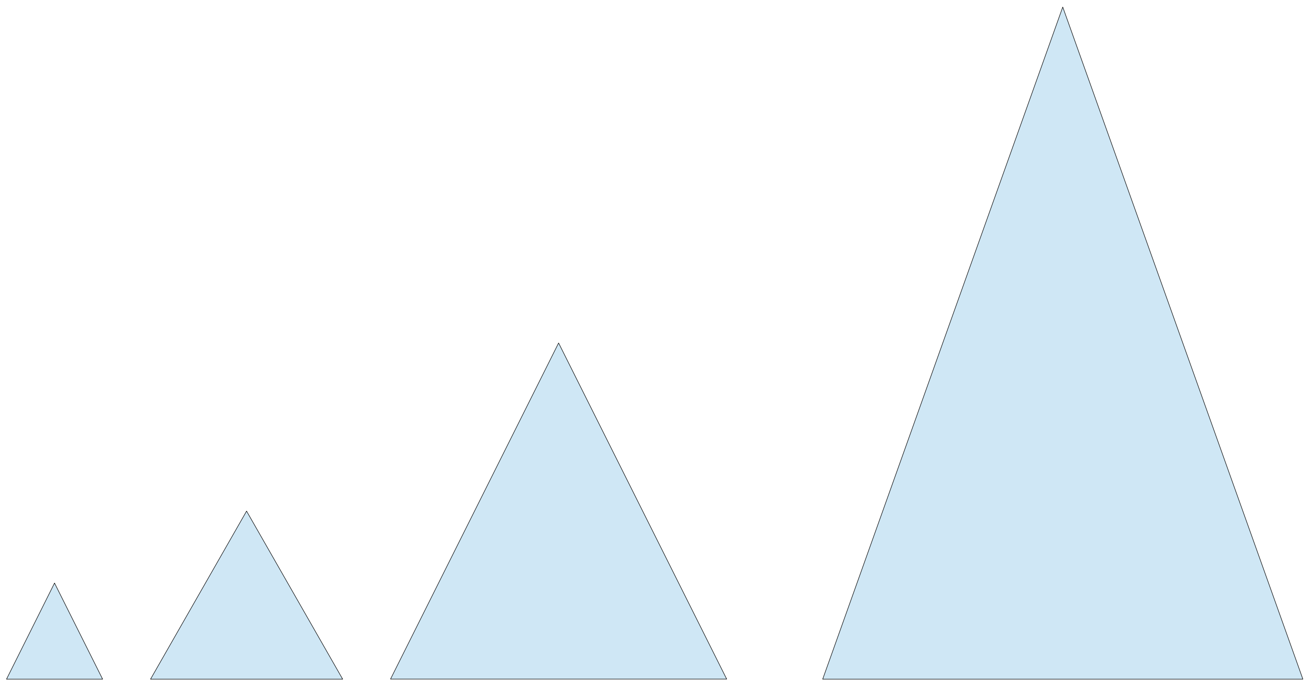


Model 4: Queries have ***temporal locality***. If a key is queried, it's likely going to be queried again soon.

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.



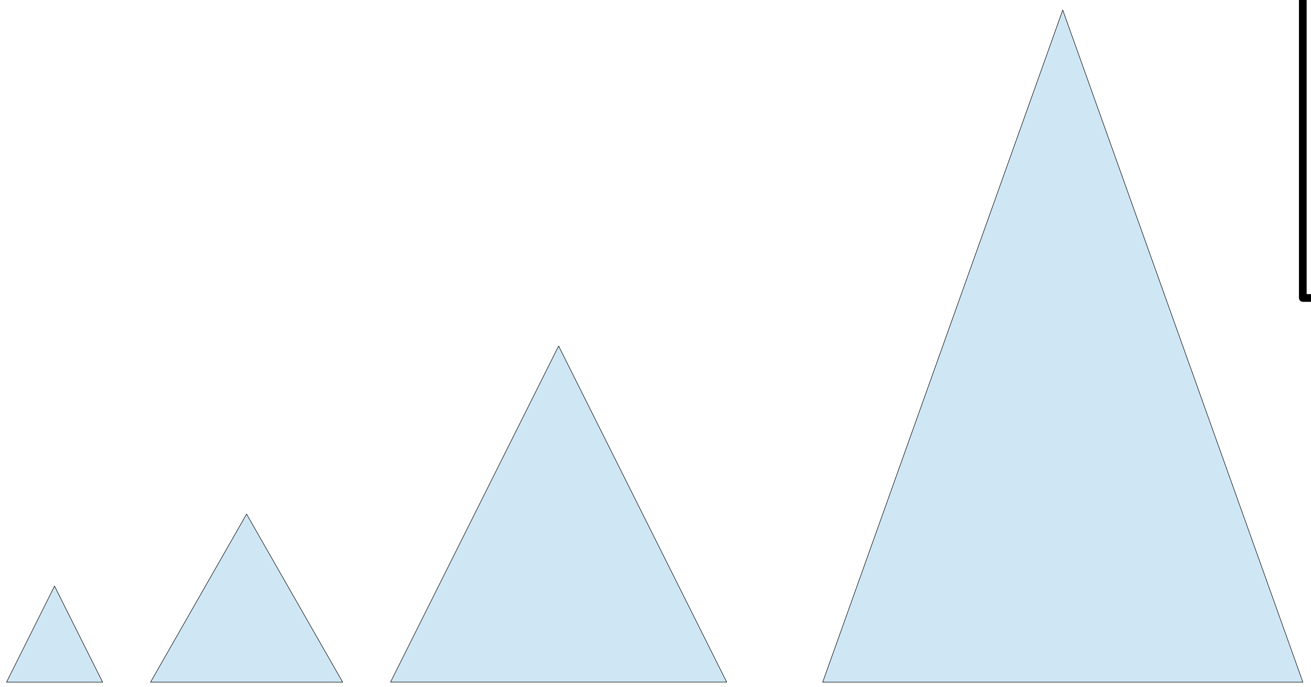
Model 4: Queries have ***temporal locality***. If a key is queried, it's likely going to be queried again soon.

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.

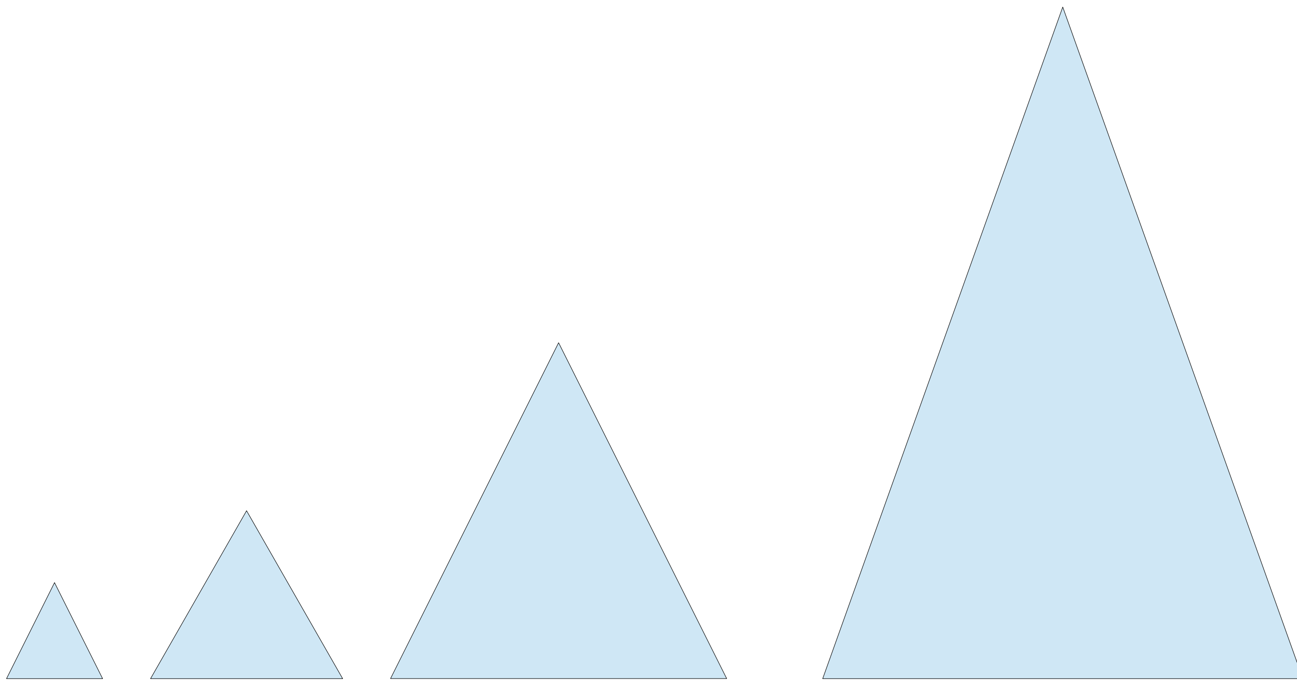
Idea: Each tree's height is double that of the previous tree.

Tree heights:

$2^0, 2^1, 2^2, 2^3, \ldots,$

Nodes per tree (roughly):

$2^{2^0}, 2^{2^1}, 2^{2^2}, 2^{2^3}, \ldots$



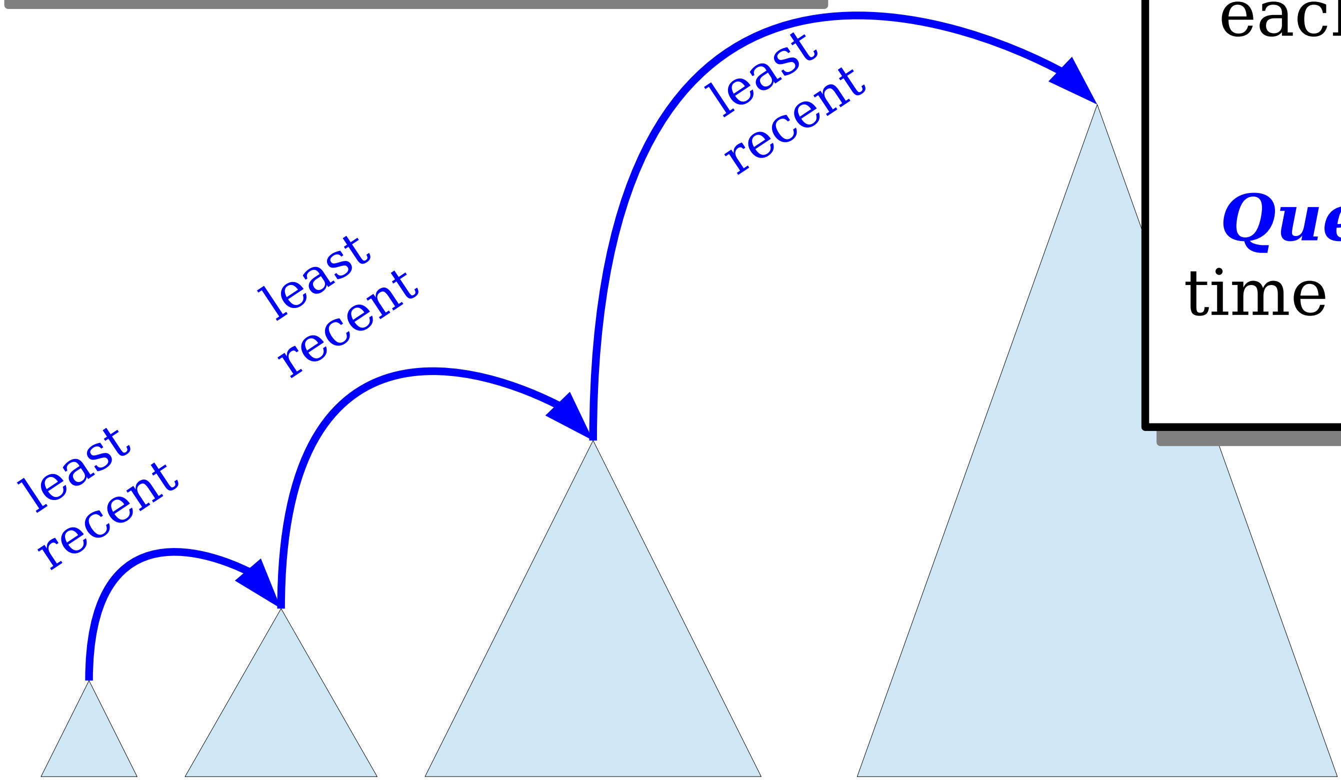
2^{2^0} 2^{2^1} 2^{2^2} $\ldots$ $2^{2^{\lg \lg n}}$

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.

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?



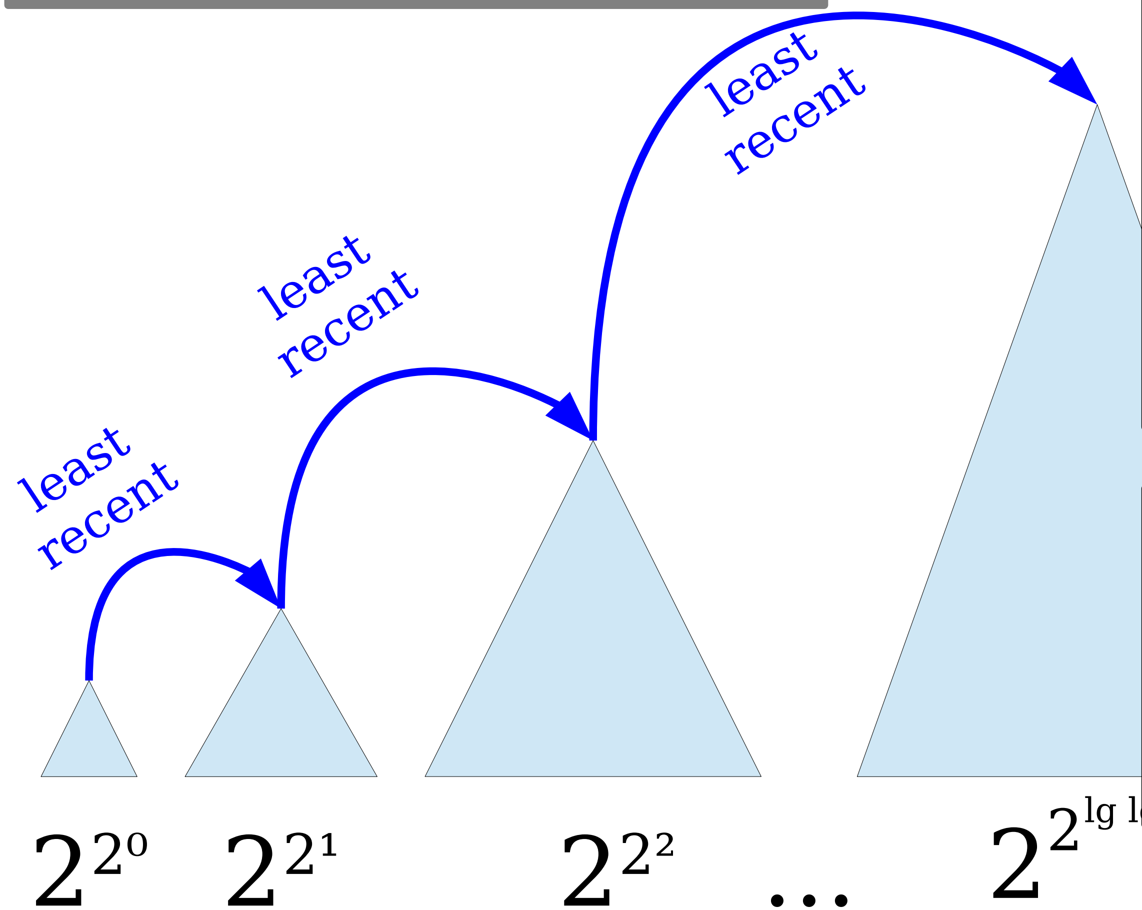
2^{2^0} 2^{2^1} 2^{2^2} … $2^{2^{\lg \lg n}}$

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.

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

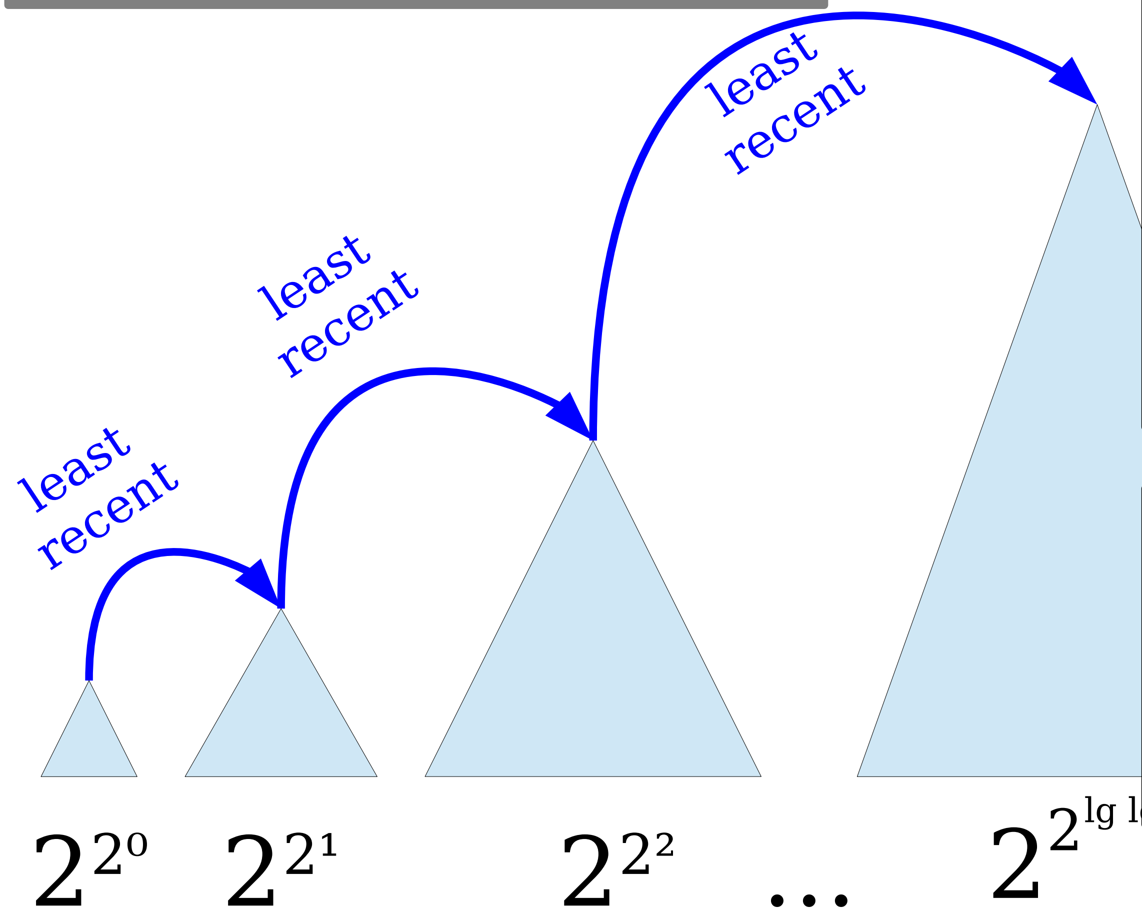
Question: How much time does this take, as a function of *n*?



Formulate a hypothesis, but ***don’t post anything in chat just yet***.

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?

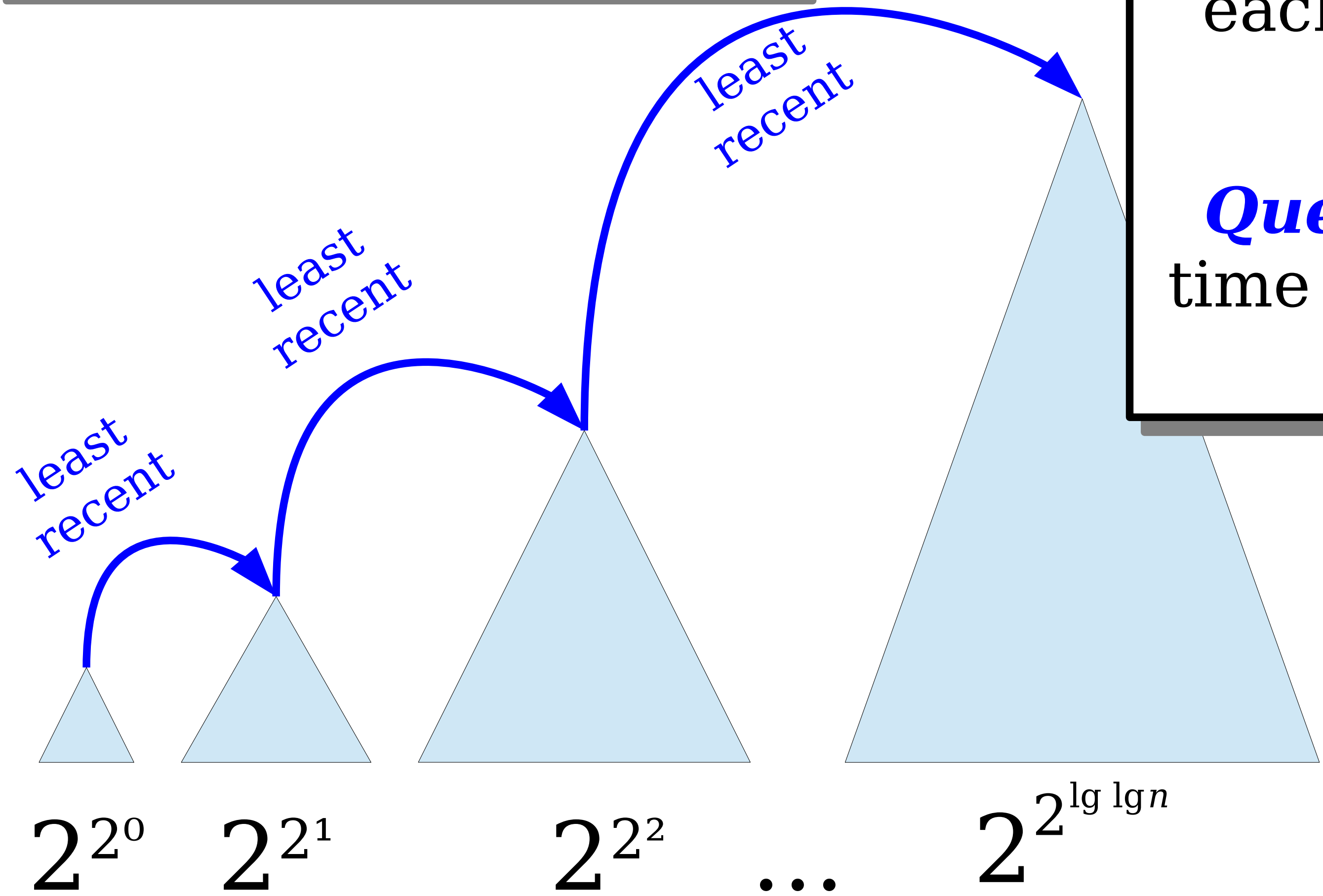
Now, ***private chat me your best guess***. Not sure? Answer "??."

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?



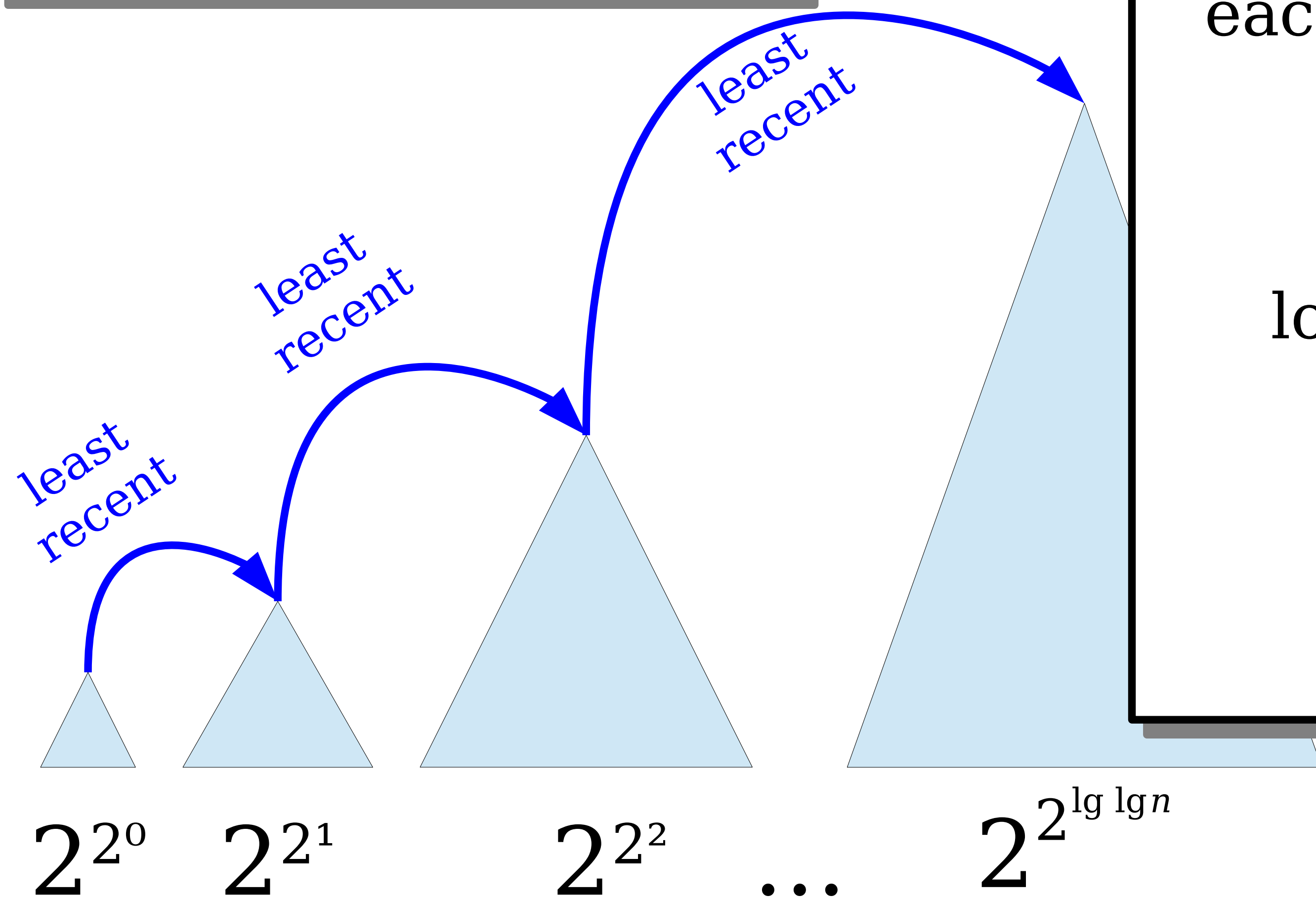
Model 4: Queries have ***temporal locality***. If a key is queried, it's likely going to be queried again soon.

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:

$$\log 2^{2^0} + \ldots + \log 2^{2^{\lg \lg n}}$$



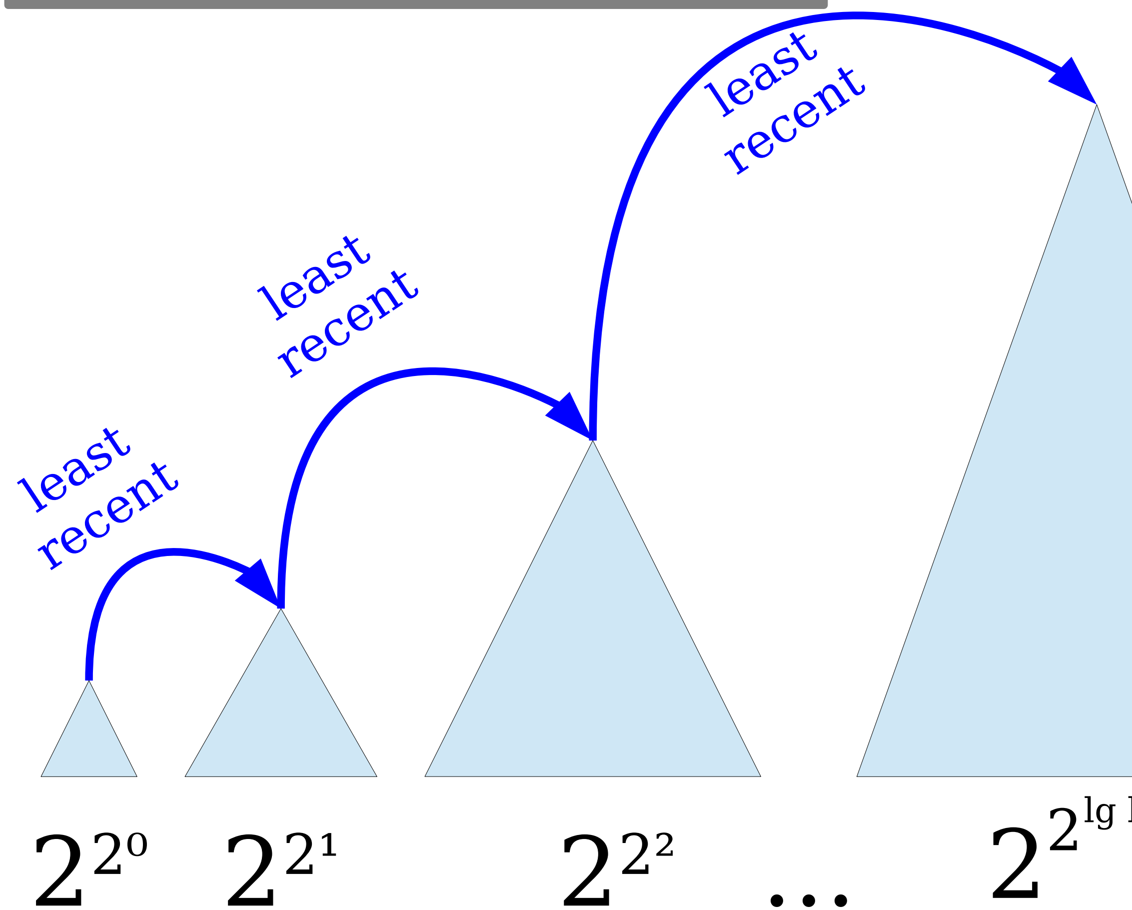
Model 4: Queries have ***temporal locality***. If a key is queried, it's likely going to be queried again soon.

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

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:

$$\log 2^{2^0} + \ldots + \log 2^{2^{\lg \lg n}}$$
$$= \quad 2^0 + 2^1 + \ldots + 2^{\lg \lg n}$$



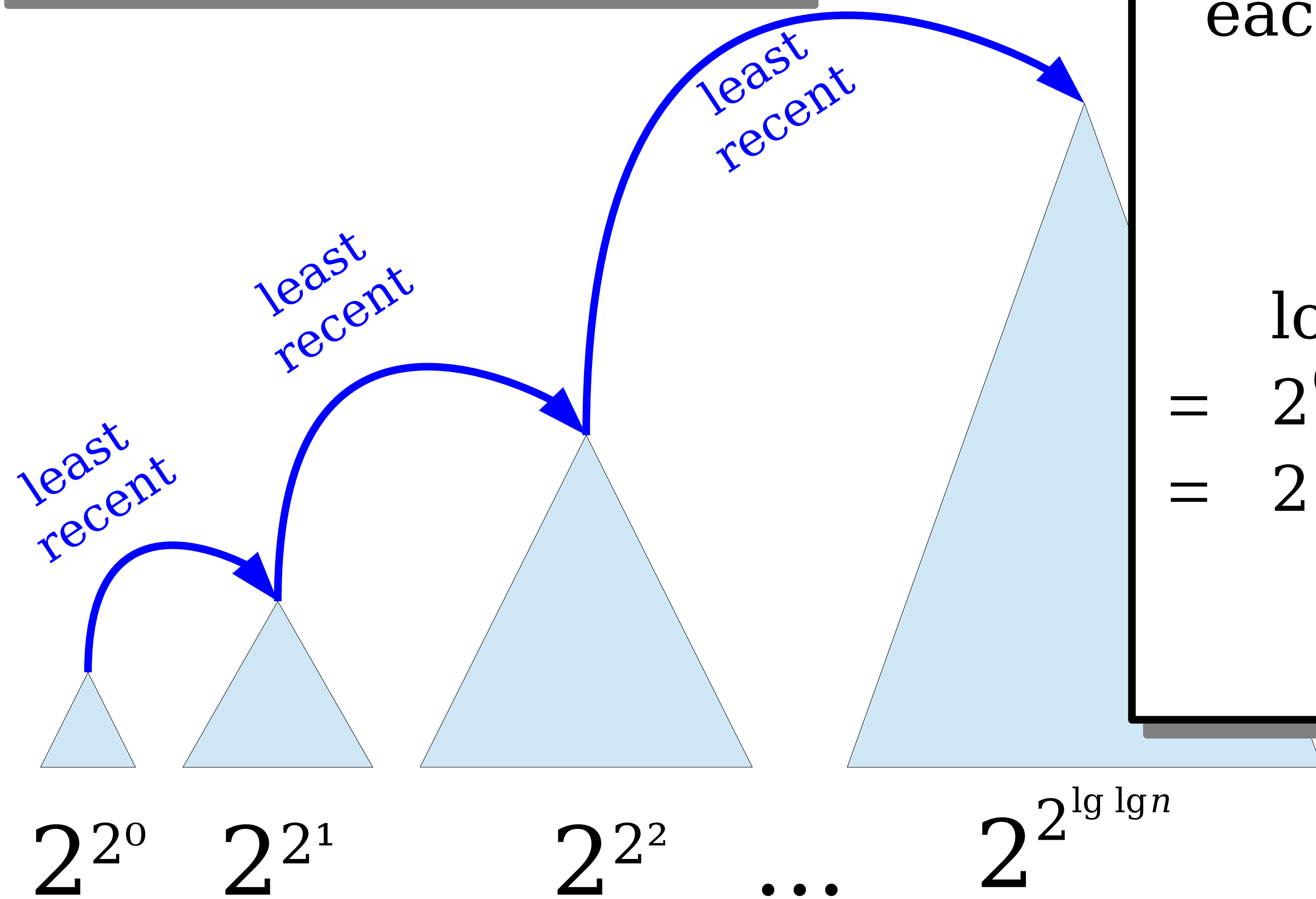
Model 4: Queries have ***temporal locality***. If a key is queried, it’s likely going to be queried again soon.

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

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:

$$\begin{aligned} & \log 2^{2^0} + \ldots + \log 2^{2^{\lg \lg n}} \\ = \; & 2^0 + 2^1 + \ldots + 2^{\lg \lg n} \\ = \; & 2^{1 + \lg \lg n} - 1 \end{aligned}$$



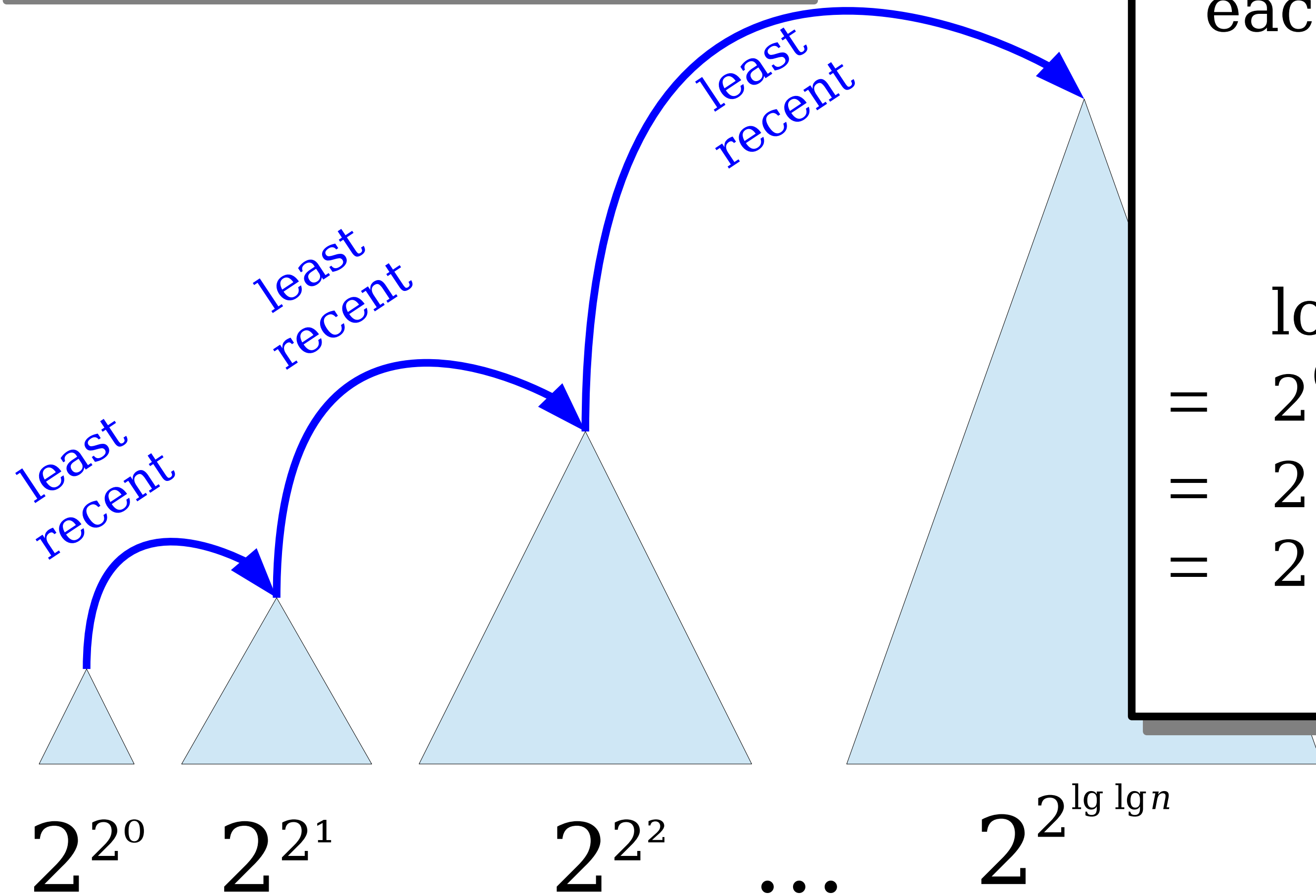
Model 4: Queries have ***temporal locality***. If a key is queried, it’s likely going to be queried again soon.

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:

$$
\begin{aligned}
& \log 2^{2^0} + \ldots + \log 2^{2^{\lg \lg n}} \\
= \;& 2^0 + 2^1 + \ldots + 2^{\lg \lg n} \\
= \;& 2^{1 + \lg \lg n} - 1 \\
= \;& 2 \lg n - 1
\end{aligned}
$$



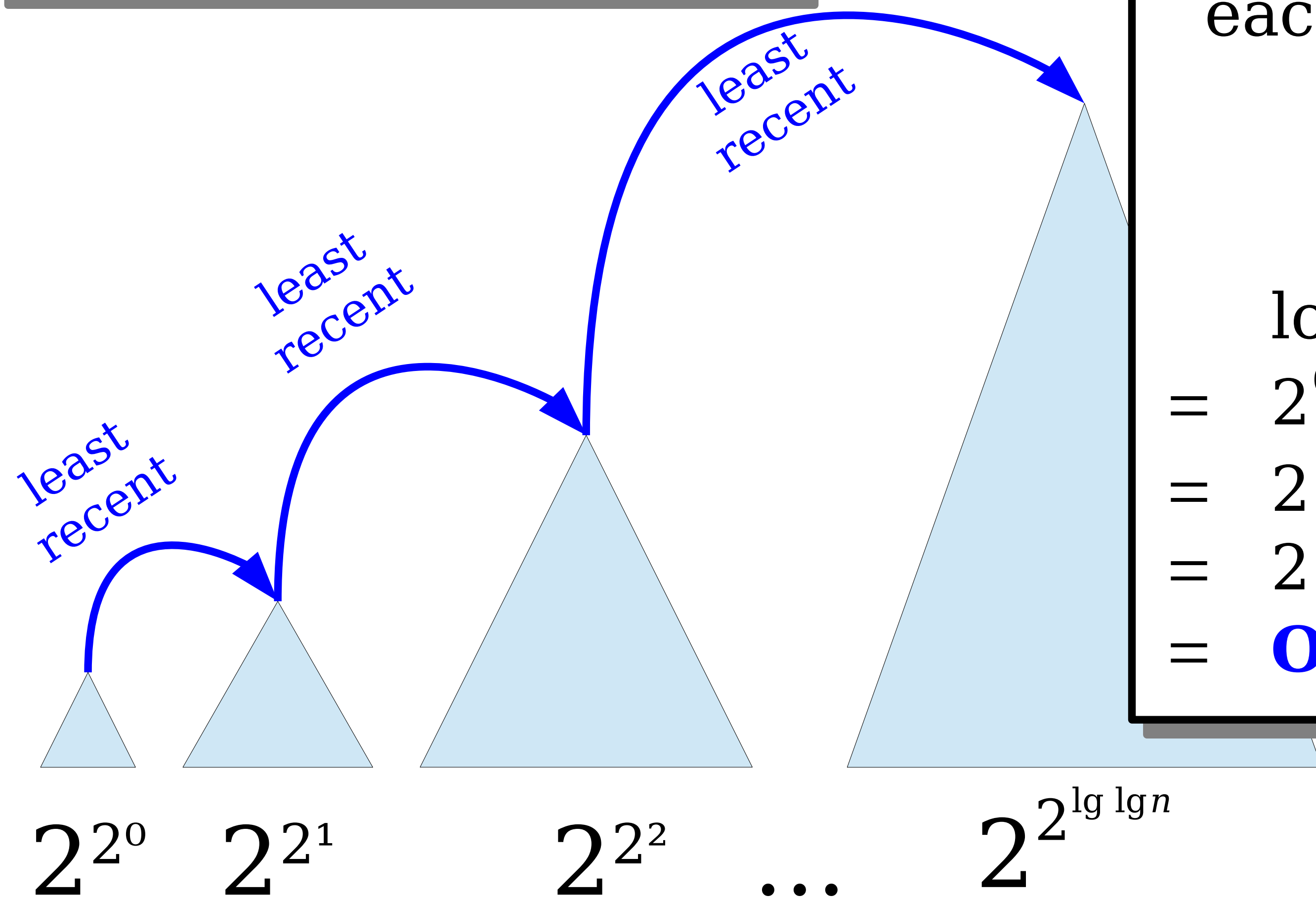
Model 4: Queries have ***temporal locality***. If a key is queried, it’s likely going to be queried again soon.

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:

$$
\begin{aligned}
& \log 2^{2^0} + \ldots + \log 2^{2^{\lg \lg n}} \\
= \ & 2^0 + 2^1 + \ldots + 2^{\lg \lg n} \\
= \ & 2^{1 + \lg \lg n} - 1 \\
= \ & 2 \lg n - 1 \\
= \ & \mathbf{O(\log n)}
\end{aligned}
$$



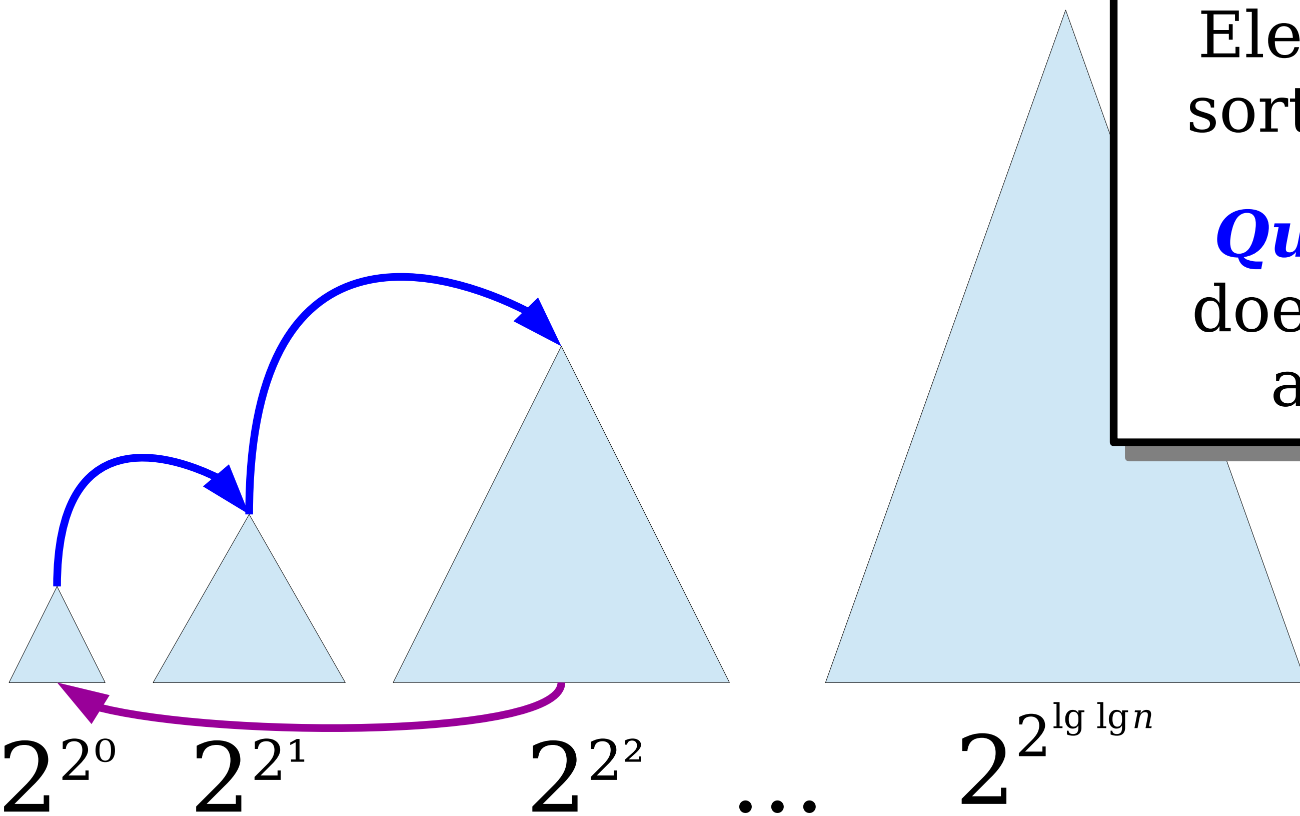
Model 4: Queries have ***temporal locality***. If a key is queried, it's likely going to be queried again soon.

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?



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.

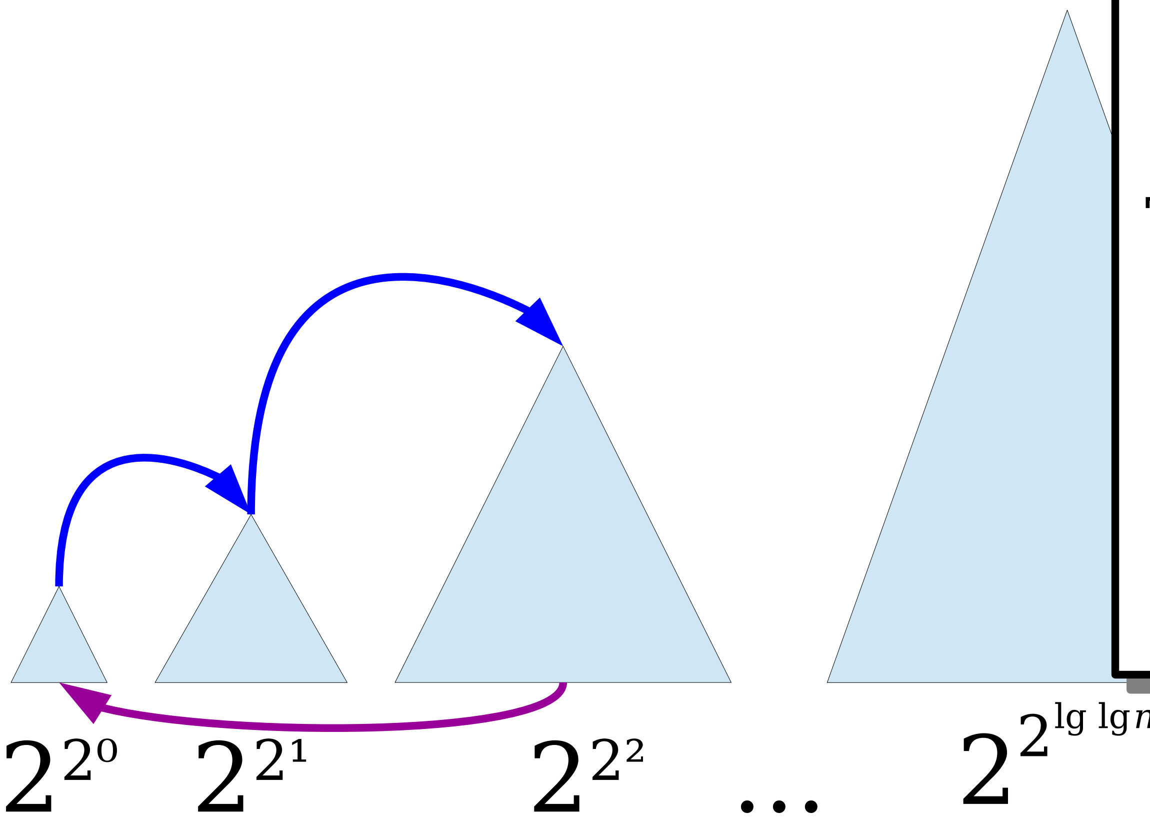
The cost of looking up an item x depends on how long it's been since we last queried it.

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

Then x is in, at most, the $(1 + \lg \lg t)$th tree.

Cost of querying x:

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

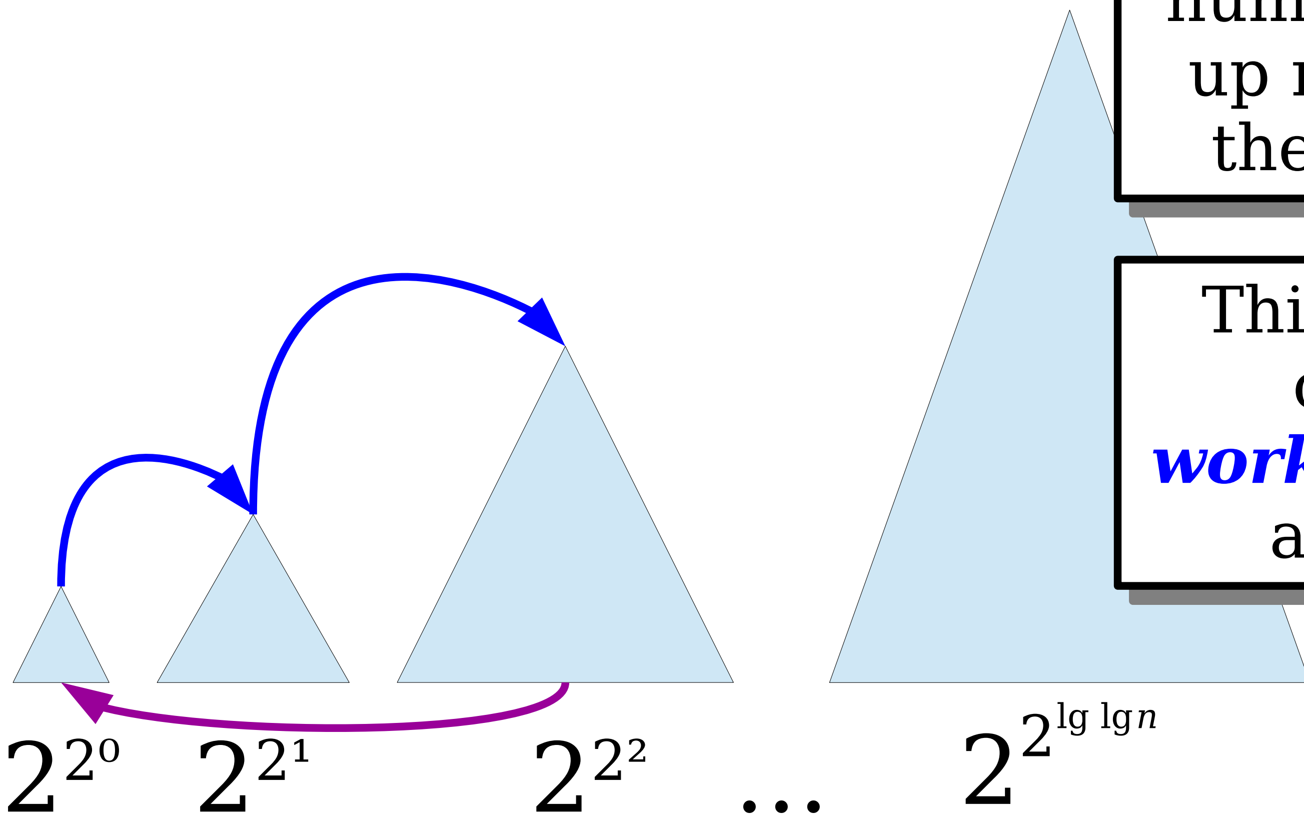


Model 4: Queries have ***temporal locality***. If a key is queried, it's likely going to be queried again soon.

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.

Property	***Description***	***Met by***
Balance	Lookups take time O(log n).	Traditional balanced BST
Entropy	Lookups take expected time O(1 + H).	Weight-equalized trees
Dynamic Finger	Lookups take O(log Δ); Δ is rank distance.	Level-linked BST with finger
Working Set	Lookups take O(log t); t is recency.	Iacono's structure

These models are in tension with one another.

All elements are equally important!

Property		***Met by***
Balance	Lookups take time O(log n).	Traditional balanced BST
Entropy	Lookups take expected time O(1 + H).	Weight-equalized trees
Dynamic Finger	Lookups take O(log Δ); Δ is rank distance.	Level-linked BST with finger
Working Set	Lookups take O(log t); t is recency.	Iacono’s structure

These models are in tension with one another.

All elements are equally important!

No they aren't! Some get queried more!

Property		***Met by***
Balance	Lookups take time	Traditional balanced BST
Entropy	Lookups take expected time O(1 + H).	Weight-equalized trees
Dynamic Finger	Lookups take O(log Δ); Δ is rank distance.	Level-linked BST with finger
Working Set	Lookups take O(log t); t is recency.	Iacono's structure

These models are in tension with one another.

All elements are equally important!

No they aren't! Some get queried more!

And some are similar to the last query!

Property		***Met by***
Balance	Lookups take time	Traditional balanced BST
Entropy	Lookups take expected time O(1 + H)	Weight-equalized trees
Dynamic Finger	Δ is rank distance.	Level-linked BST with finger
Working Set	Lookups take O(log t); t is recency.	Iacono's structure

These models are in tension with one another.

Property		*Met by*
Balance	Lookups take time	Traditional balanced BST
Entropy		Weight-equalized trees
Dynamic Finger	Δ is rank distance.	Level-linked BST with finger
Working Set	Look	Iacono's structure

All elements are equally important!

No they aren't! Some get queried more!

And some are similar to the last query!

And some were queried more recently!

These models are in tension with one another.

Property	***Description***	***Met by***
Balance	Lookups take time O(log n).	Traditional balanced BST
Entropy	Lookups take expected time O(1 + H).	Weight-equalized trees
Dynamic Finger	Lookups take O(log Δ); Δ is rank distance.	Level-linked BST with finger
Working Set	Lookups take O(log t); t is recency.	Iacono's structure

These models are in tension with one another.

Lookups are sampled from a fixed distribution.

Property		***Met by***
Balance	L ps take time O(log n).	Traditional balanced BST
Entropy	Lookups take expected time O(1 + H).	Weight-equalized trees
Dynamic Finger	Lookups take O(log Δ); Δ is rank distance.	Level-linked BST with finger
Working Set	Lookups take O(log t); t is recency.	Iacono's structure

These models are in tension with one another.

Lookups are sampled from a fixed distribution.

What if I do a linear scan?

Property		***Met by***
Balance	L ps take time O(	Traditional lanced BST
Entropy	Lookups time O(*H*).	-equalized trees
Dynamic Finger	Lookups take O(log Δ); Δ is rank distance.	Level-linked BST with finger
Working Set	Lookups take O(log *t*); *t* is recency.	Iacono's structure

These models are in tension with one another.

Property		*Met by*
Balance	L … ps take time O(l …)	Traditional b … nced BST
Entropy	Lookups … time O(… H).	… -equalized trees
Dynamic Finger	Lookups t … Δ is ran …	… linked … h finger
Working Set	Lookups take O(log t); t is recency.	Iacono's structure

Lookups are sampled from a fixed distribution.

What if I do a linear scan?

What if there are correlations?

These models are in tension with one another.

Property	***Description***	***Met by***
Balance	Lookups take time O(log n).	Traditional balanced BST
Entropy	Lookups take expected time O(1 + H).	Weight-equalized trees
Dynamic Finger	Lookups take O(log Δ); Δ is rank distance.	Level-linked BST with finger
Working Set	Lookups take O(log t); t is recency.	Iacono's structure

These models are in tension with one another.

Property	*Description*	*Met by*
Balance		Traditional balanced BST
Entropy	Lookups take expected time $O(1 + H)$.	Weight-equalized trees
Dynamic Finger	Lookups take $O(\log \Delta)$; Δ is rank distance.	Level-linked BST with finger
Working Set	Lookups take $O(\log t)$; t is recency.	Iacono's structure

*Distance in **key space** is what's important!*

These models are in tension with one another.

Property	***Description***	***Met by***
Balance		Traditional balanced BST
Entropy	Loo… s take expected time O(…	Weight-equalized … trees
Dynamic Finger	Lookups … Δ is rank … ance.	… linked BST with finger
Working Set	Lookups take O(log t); t is recency.	Iacono's structure

Distance in ***key space*** *is what's important!*

Distance in ***time*** *is what's important!*

These models are in tension with one another.

Property	*Description*	*Met by*
Balance	Lookups take time O(log n).	Traditional balanced BST
Entropy	Lookups take expected time O(1 + H).	Weight-equalized trees
Dynamic Finger	Lookups take O(log Δ); Δ is rank distance.	Level-linked BST with finger
Working Set	Lookups take O(log t); t is recency.	Iacono's structure

Is there a single BST that guarantees all of these properties?

Property	***Description***	***Met by***
Balance	Lookups take time O(log n).	***Splay tree***
Entropy	Lookups take expected time O(1 + H).	***Splay tree***
Dynamic Finger	Lookups take O(log Δ); Δ is rank distance.	***Splay tree***
Working Set	Lookups take O(log t); t is recency.	***Splay tree***

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?