x-Fast and y-Fast Tries

Outline for Today

- ***Data Structures on Integers***
  - How can we speed up operations that work on integer data?
- ***x-Fast Tries***
  - Bit manipulation meets tries and hashing.
- ***y-Fast Tries***
  - Combining RMQ, strings, balanced trees, amortization, and randomization!

Working with Integers

- Many practical problems involve working specifically with integer values.
  - ***CPU Scheduling:*** Each thread has some associated integer priority, and we need to maintain those priorities in sorted order.
  - ***Network Routing:*** Each computer has an associated IP address, and we need to figure out which connections are active.
  - ***ID Management:*** We need to store social security numbers, zip codes, phone numbers, credit card numbers, etc. and perform basic lookups and range searches on them.
- We’ve seen many general-purpose data structures for keeping things in order and looking things up.
- ***Question:*** Can we improve those data structures if we know in advance that we’re working with integer data?

Working with Integers

- Integers are interesting objects to work with:
  - Their values can directly be used as indices in lookup tables.
  - They can be treated as strings of bits, so we can use techniques from string processing.
  - They fit into machine words, so we can process the bits in parallel with individual word operations.
- The data structures we'll explore over the next few lectures will give you a sense of what sorts of techniques are possible with integer data.

An Auxiliary Motive

- Integer data structures are also a great place to see just how much you’ve learned over the quarter!
- Today’s data structures cover every single unit from the quarter (RMQ, strings, balanced trees, amortization, and randomization).
- I hope this gives you a chance to pause and reflect on just how far you’ve come!

The Setup

Our Machine Model

- We will assume we're working on a machine where memory is segmented into ***w***-bit words.
  - Although on any one fixed machine w is a constant, in general, don't assume this is the case. 32-bit was the norm until fairly recently, and before that 16-bit was standard.
- We will assume that, if we build a data structure that holds n elements, then $w = \Omega(\log n)$.
  - This is called the ***transdichotomous machine model***. Essentially, your word size has to be big enough to hold the size of your input.
- We'll assume C integer operators work in constant time, and won't assume other integer operations (say, finding most significant bits, counting 1 bits set) are available.

```
+   -   *   /   %   <<   >>   &   |   ^   =   <=
```

Besting BSTs

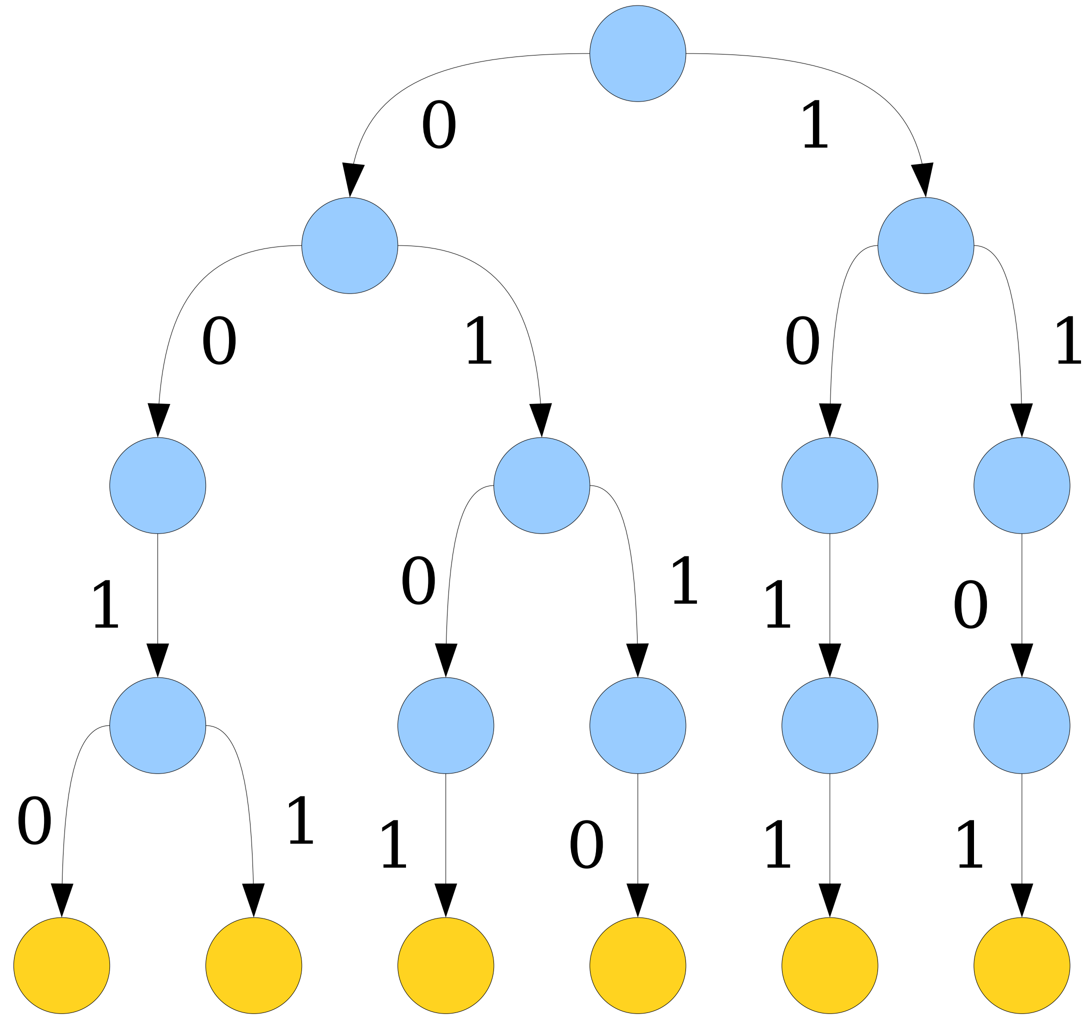
Besting BSTs

- BSTs store their elements in sorted order, which enables them to quickly answer each of the following queries:
  - ***lookup***(x), which returns whether $x \in S$;
  - ***insert***(x), which adds x to S;
  - ***delete***(x), which removes x from S;
  - ***max***() / ***min***(), which return the max/min element of S;
  - ***successor***(x), which returns the smallest element of S greater than x; and
  - ***predecessor***(x), which returns the largest element of S smaller than x.
- ***Question:*** Can we build another data structure that answers these same queries, but does so faster than a BST?

A Start: ***Bitwise Tries***

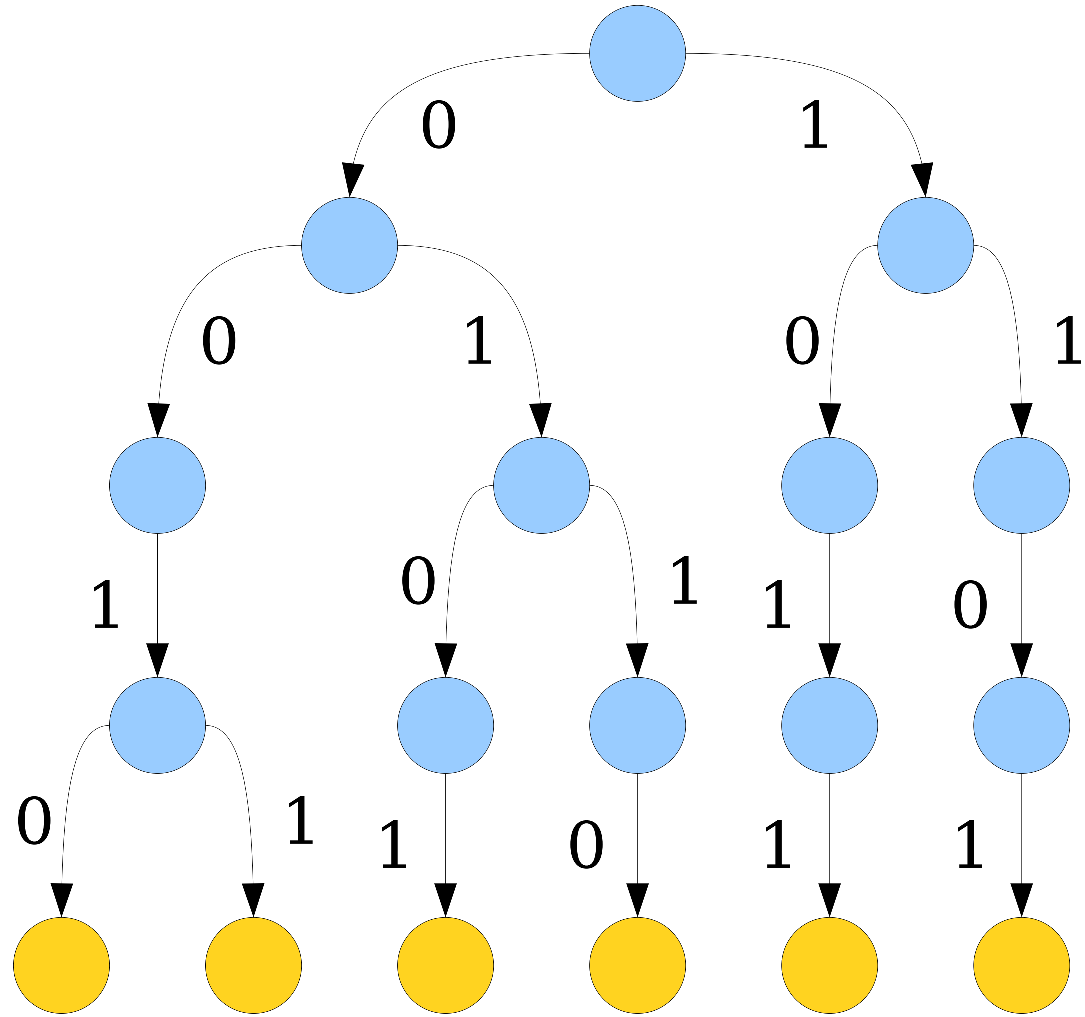
Tries Revisited

- ***Recall:*** A trie is a simple data structure for storing strings.
- Integers can be thought of as strings of bits.
- ***Idea:*** Store integers in a ***bitwise trie***.



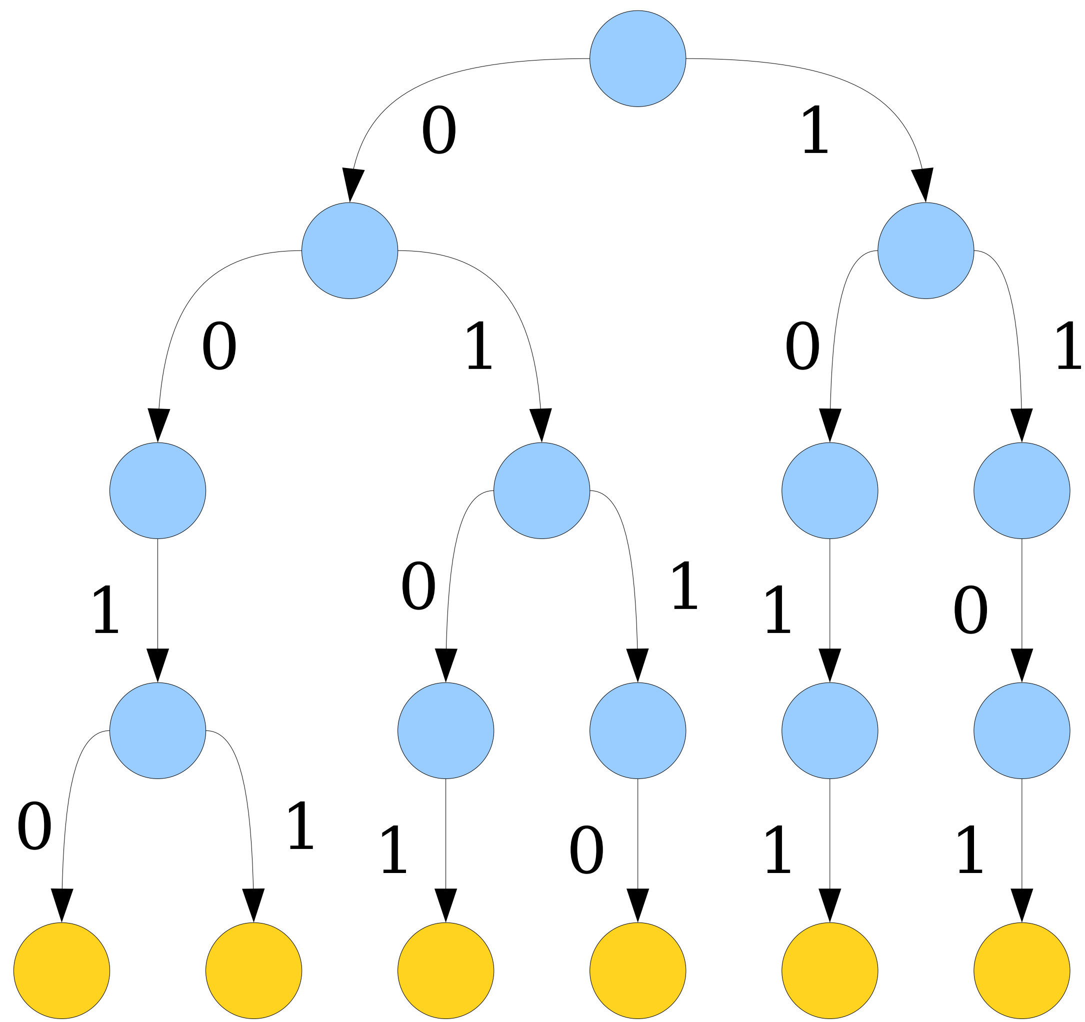
Finding Successors

- To compute ***successor***(x), do the following:
  - Search for x.
  - If x is a leaf node, its successor is the next leaf.
  - If you don't find x, back up until you find a node with a 1 child not already followed, follow the 1, then take the cheapest path down.



Bitwise Trie Efficiency

- All operations on bitwise tries take time proportional to the number of bits in each number.
- Runtime for each operation: **O(*w*)**.
  - This is worse than a BST's O(log *n*).
- ***Can we do better?***



Speeding up Successors

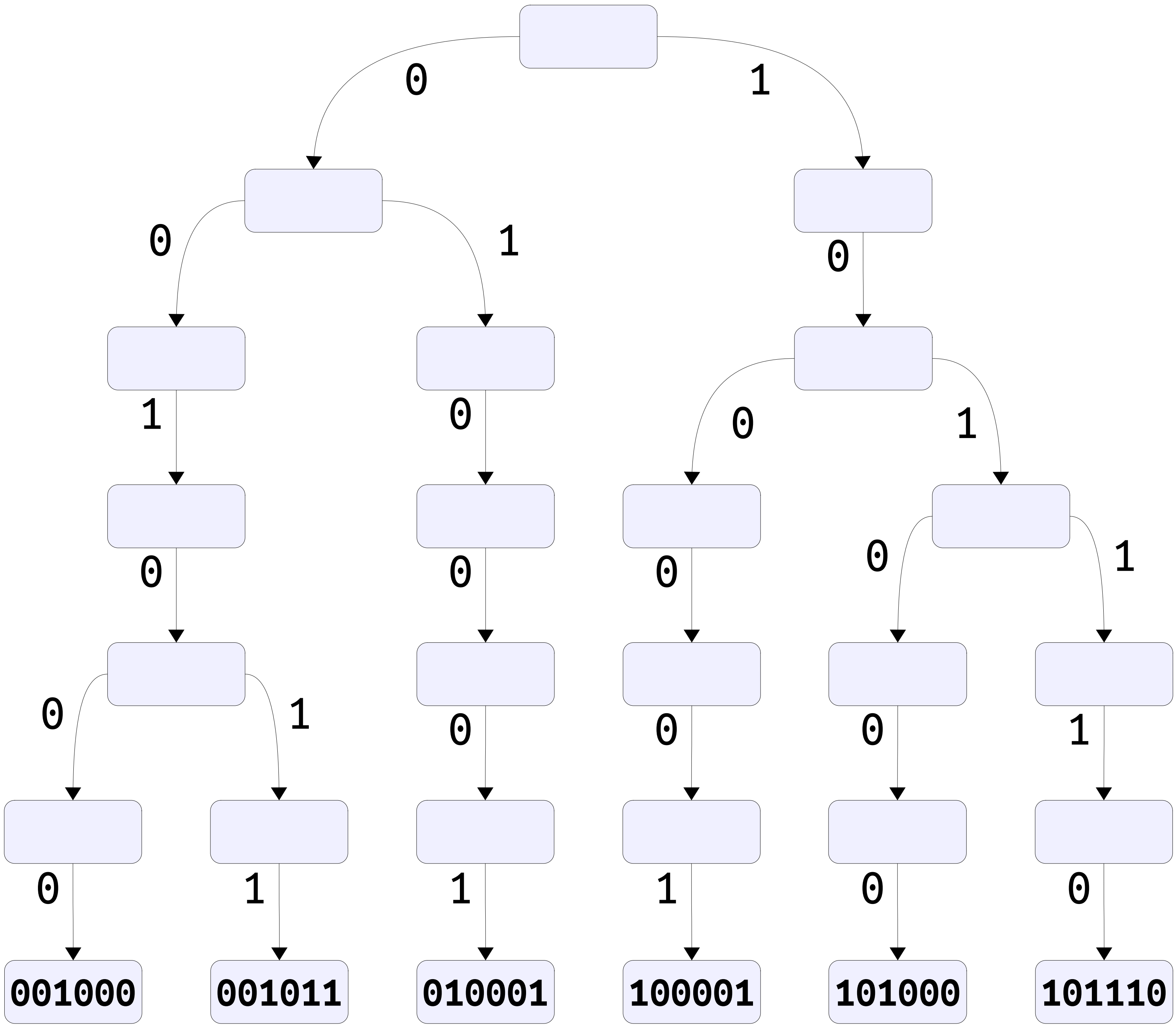
- There are two independent pieces that contribute to the O(w) runtime:
  - Need to walk down the trie following the bits of x, and there are $\Theta(w)$ of those.
  - From there, need to back up to a branching node where we can find the successor.
- Can we speed up those operations? Or at least work around them?

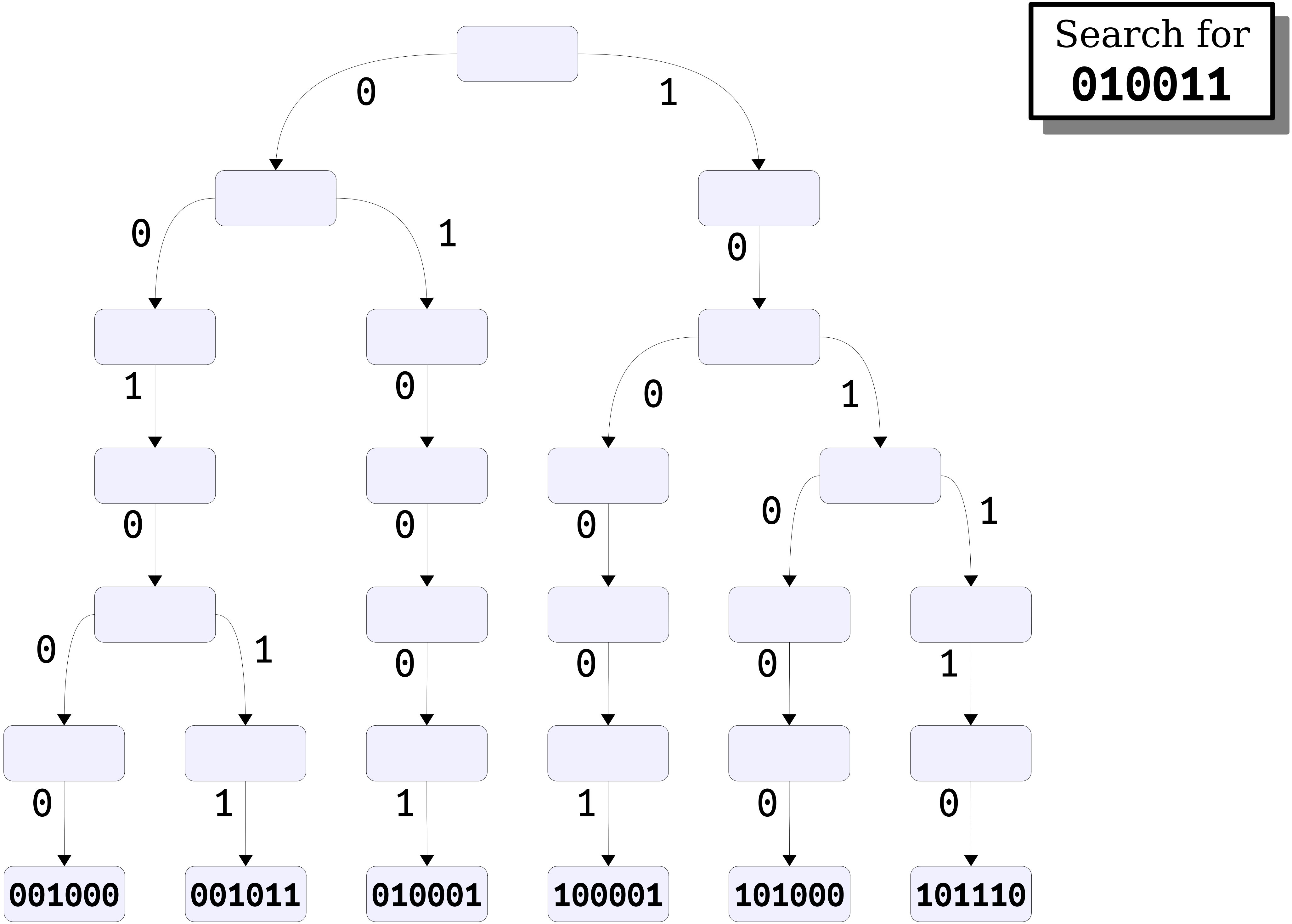
A Quick Algorithms Problem

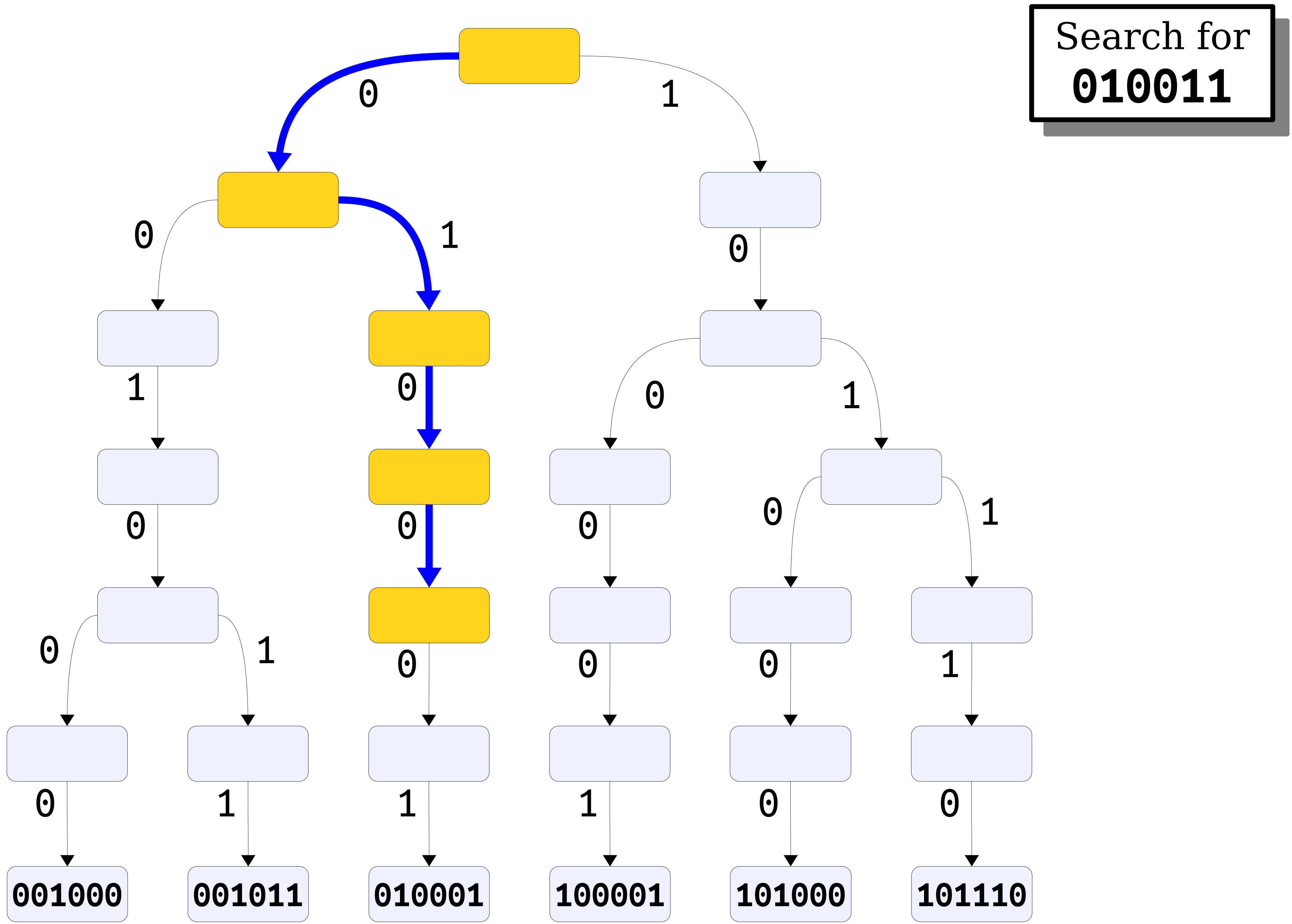
- You're given an array consisting of some number of **Y**'s followed by some number of **N**'s. There are k total letters.

YYYYYYY...YYYNNNN...NN

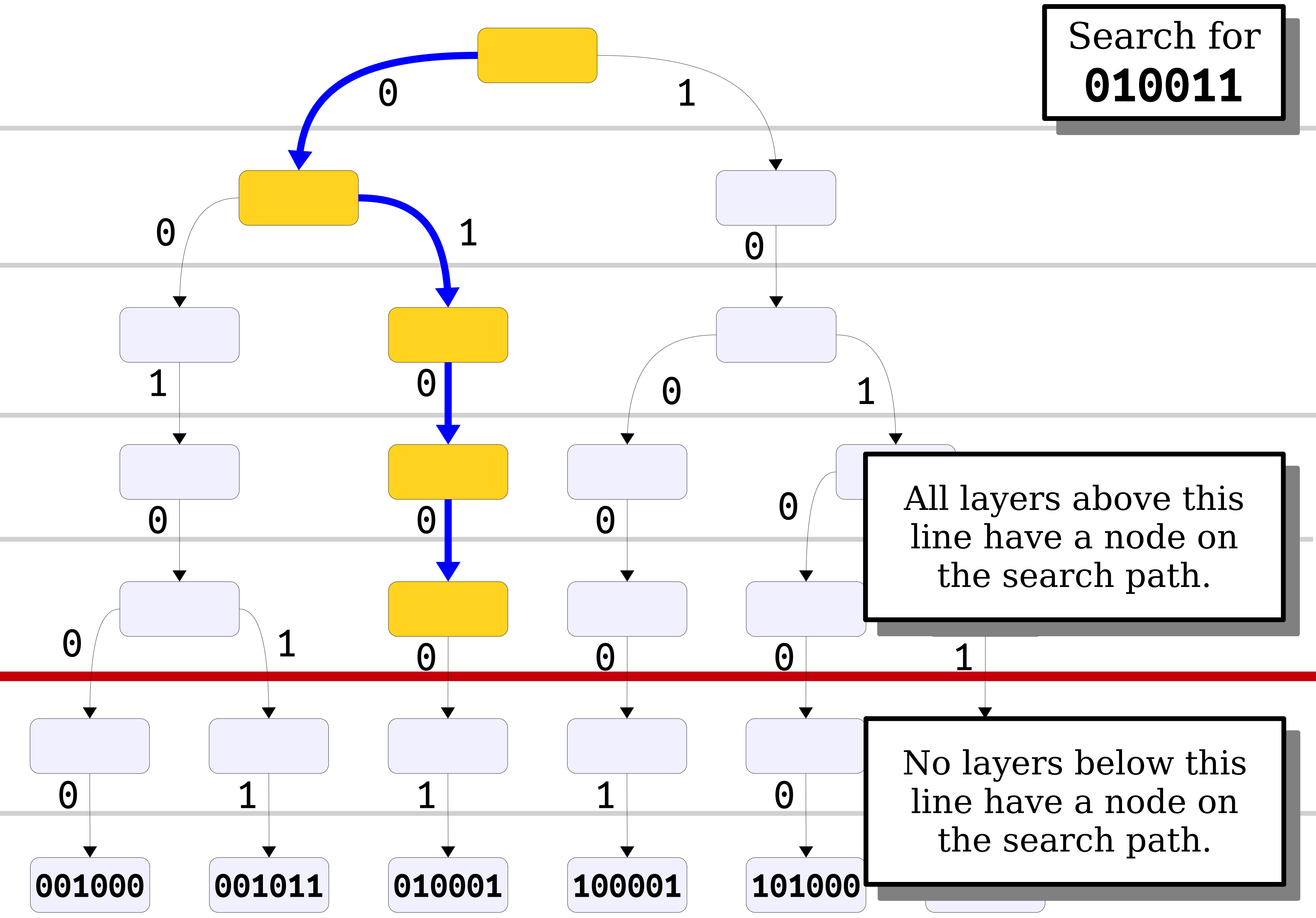
- How quickly can you find the last Y?
- ***Answer:*** Can be done in time O(log k) using a binary search.


0
1
0
1
0
1
0
1
0
0
0
0
1
0
0
1
0
0
0
1
0
1
1
0
0
001000
001011
010001
100001
101000
101110

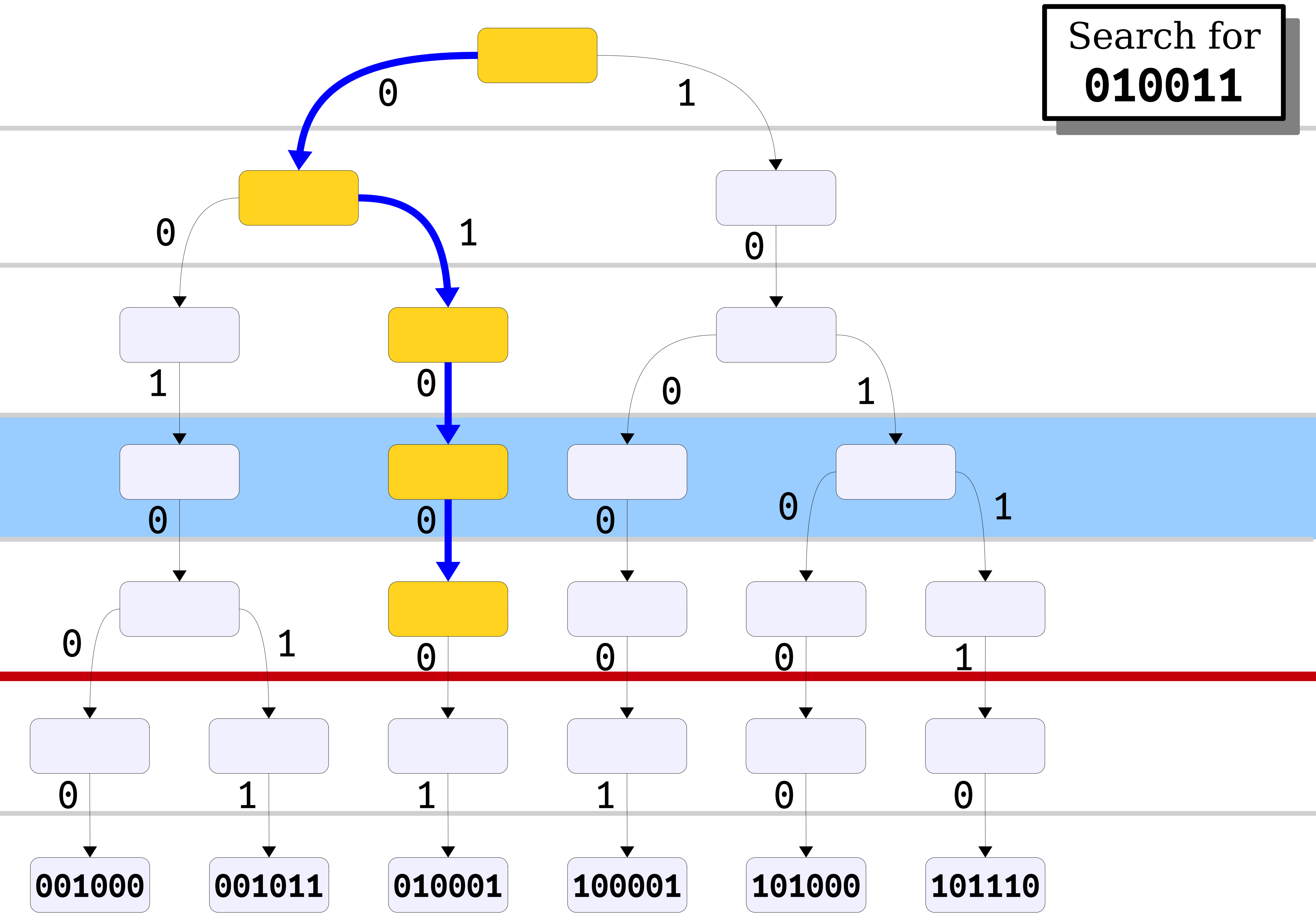

Search for
010011
0
1
0
1
0
1
0
0
1
0
0
1
0
0
0
1
0
0
0
1
1
0
1
1
1
0
0
001000
001011
010001
100001
101000
101110



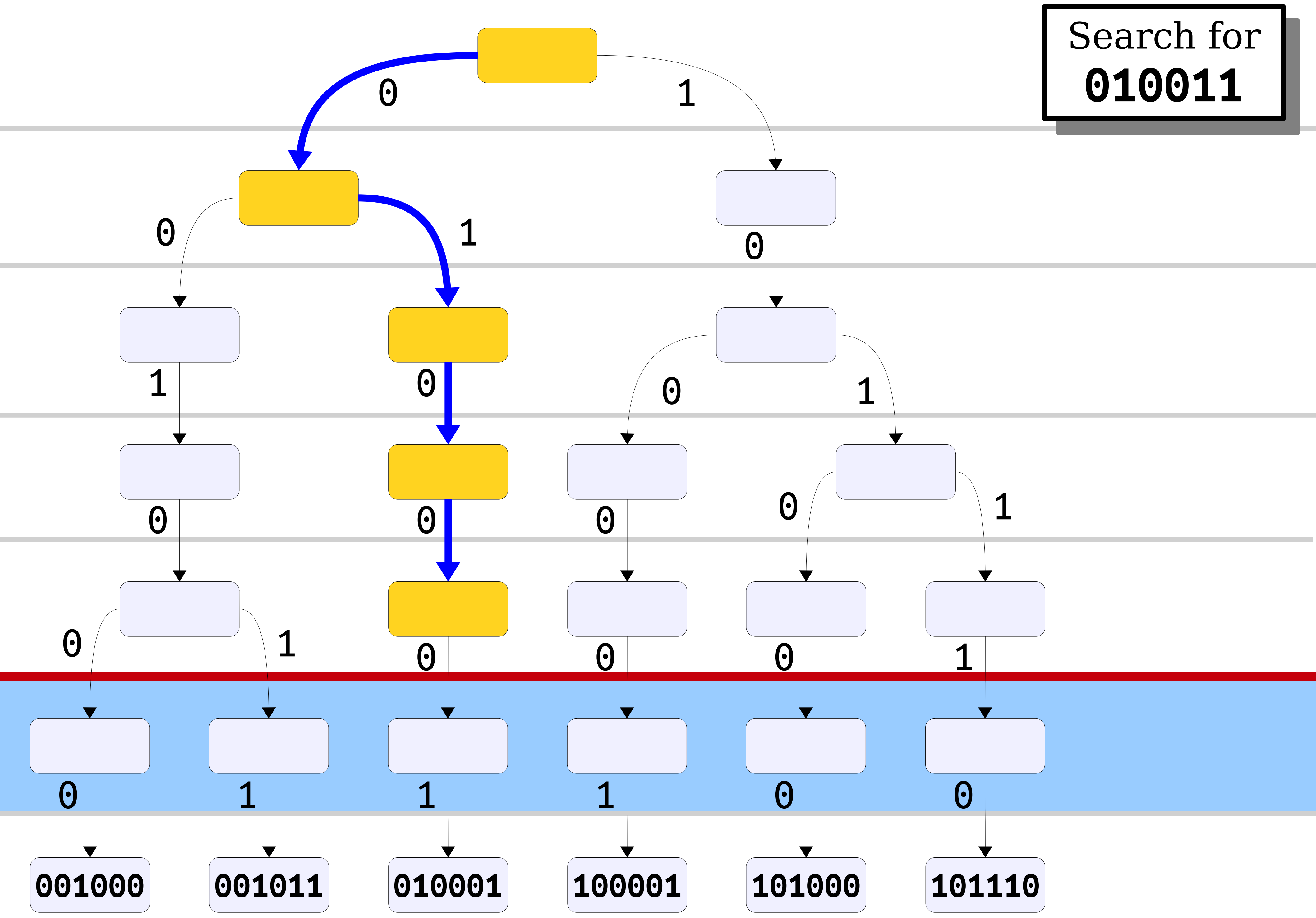
Search for
010011
0
1
0
1
0
1
0
0
1
0
0
0
1
0
1
0
1
0
0
0
1
0
1
1
1
0
0
001000
001011
010001
100001
101000
101110



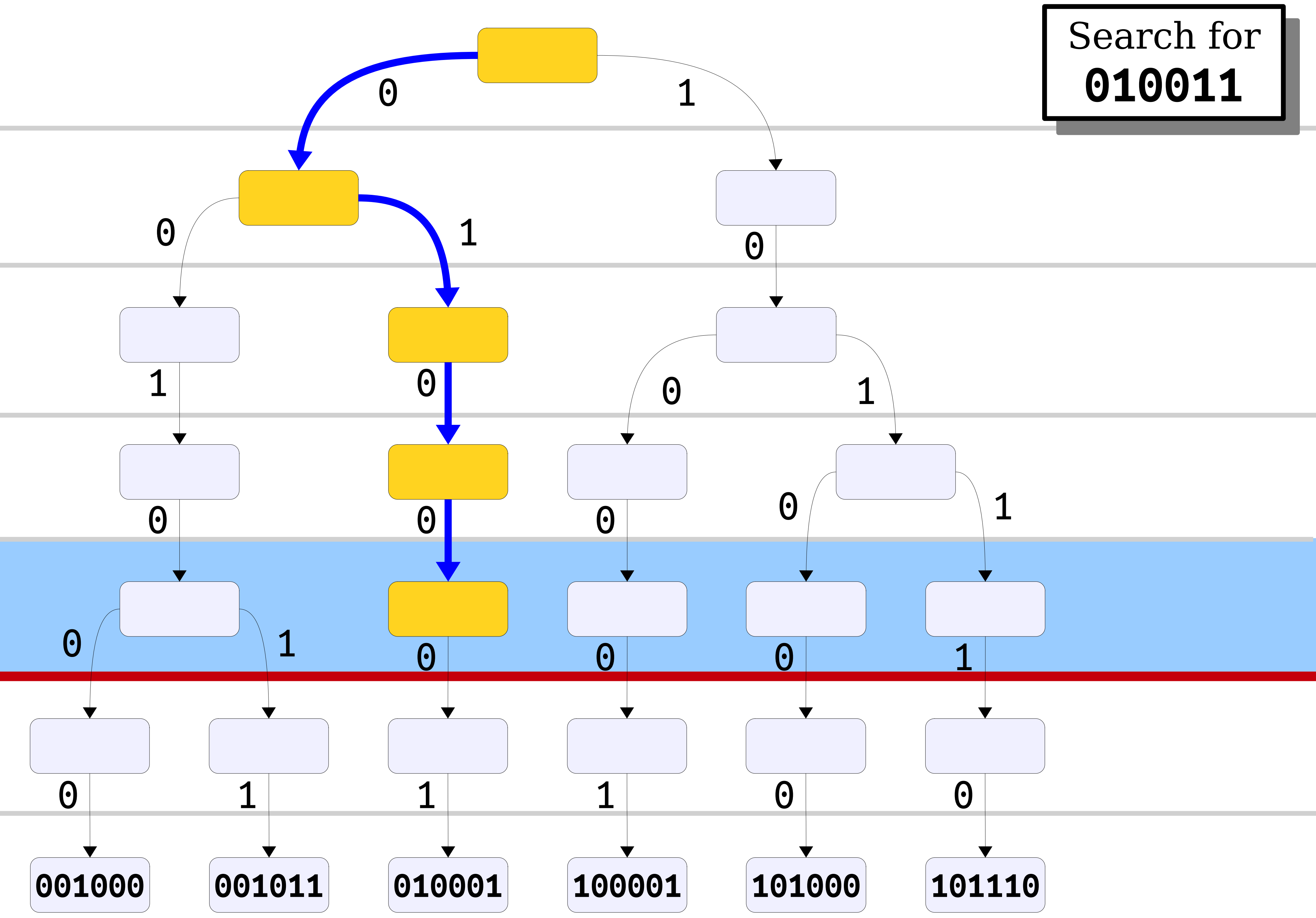
Search for
010011
0
1
0
1
0
1
0
0
1
0
0
0
0
0
0
1
0
0
0
1
0
1
1
1
0
001000
001011
010001
100001
101000
All layers above this line have a node on the search path.
No layers below this line have a node on the search path.



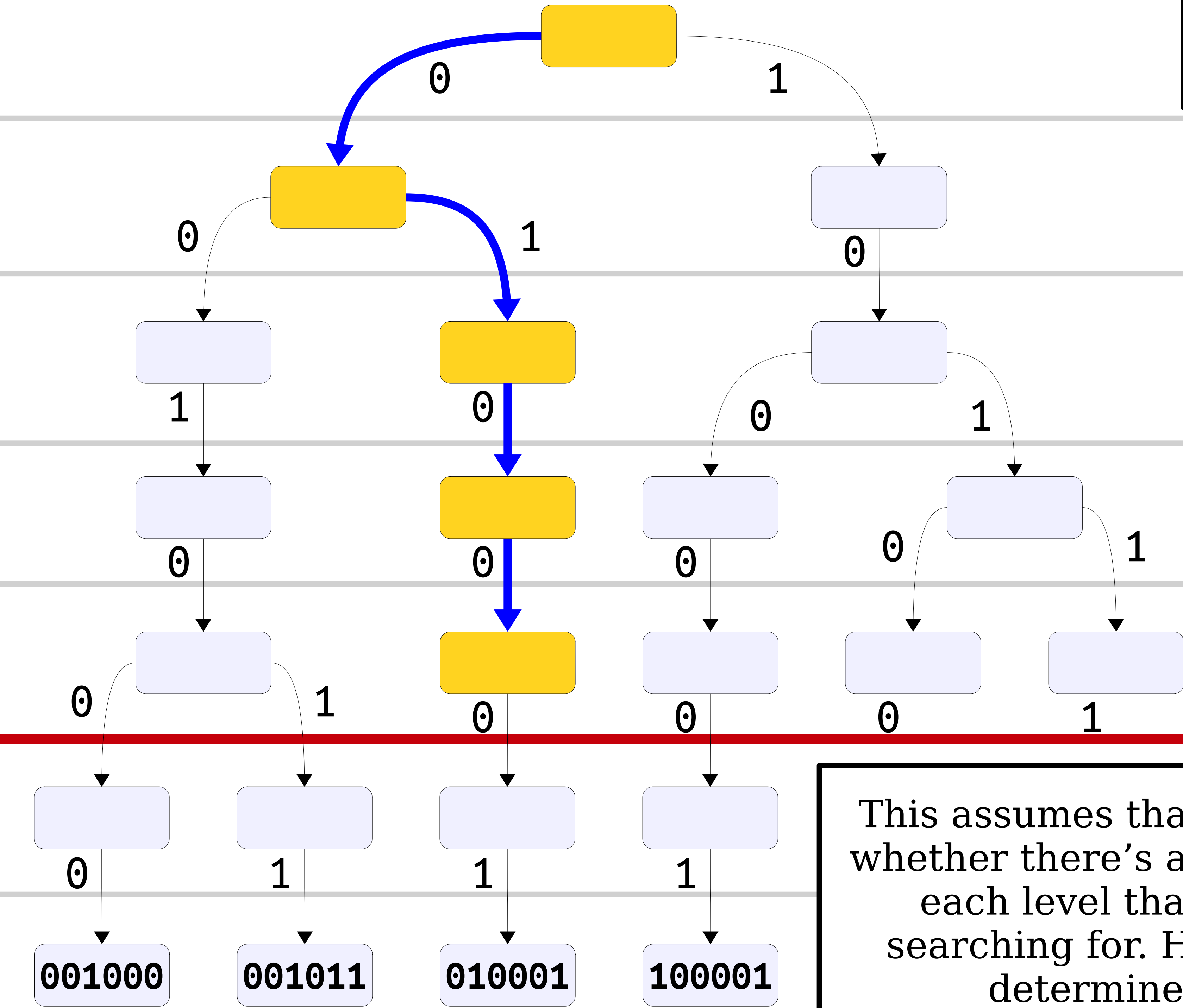
Search for
010011
0
1
0
1
0
1
0
0
1
0
0
0
0
1
0
1
0
0
0
1
0
1
1
1
0
0
001000
001011
010001
100001
101000
101110



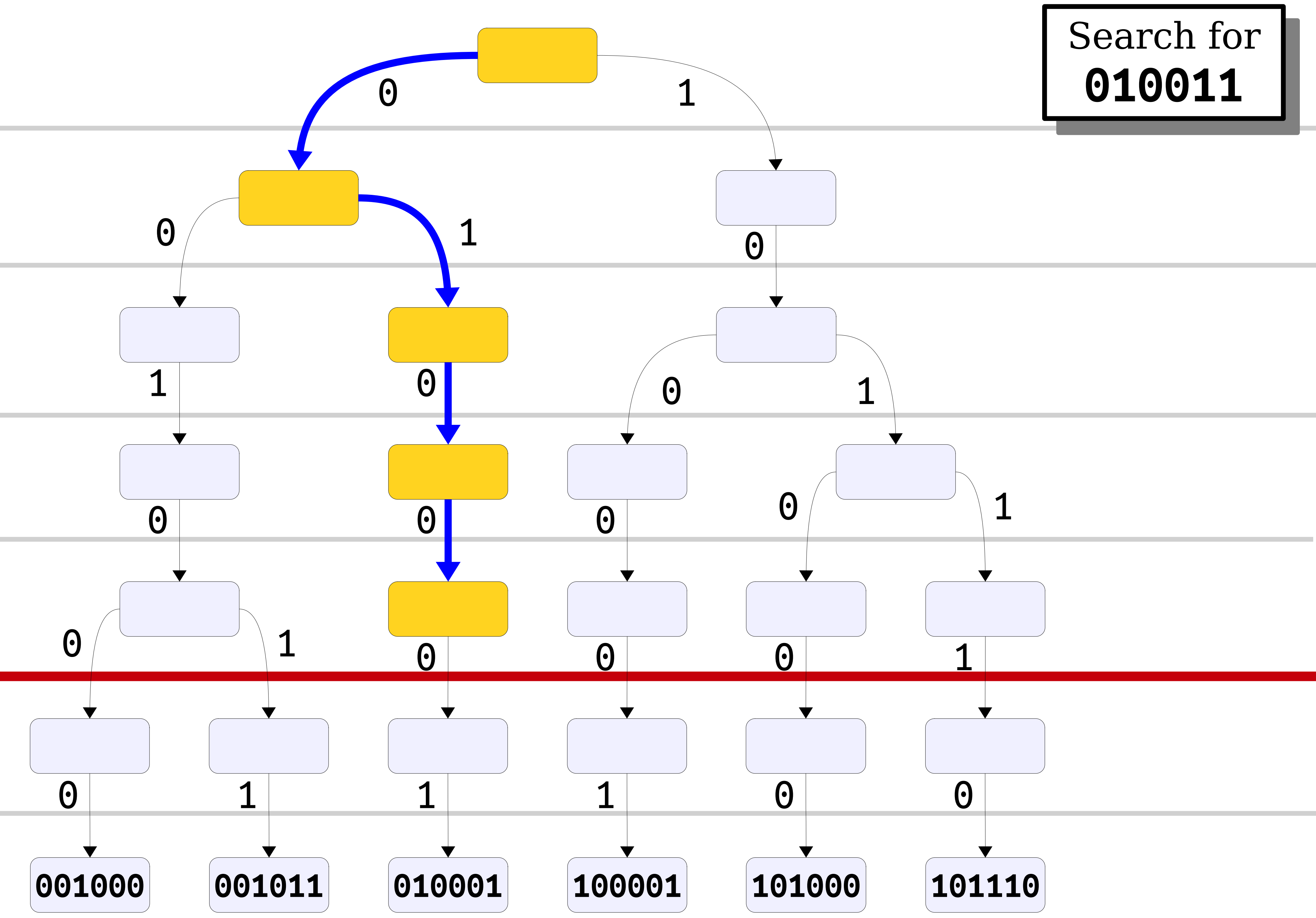
Search for
010011
0
1
0
1
0
1
0
0
1
0
0
0
0
1
0
1
0
0
0
1
0
1
1
1
0
0
001000
001011
010001
100001
101000
101110


Search for
010011
0
1
0
1
0
1
0
0
1
1
0
0
0
0
1
0
1
0
0
0
1
0
1
1
1
0
0
001000
001011
010001
100001
101000
101110

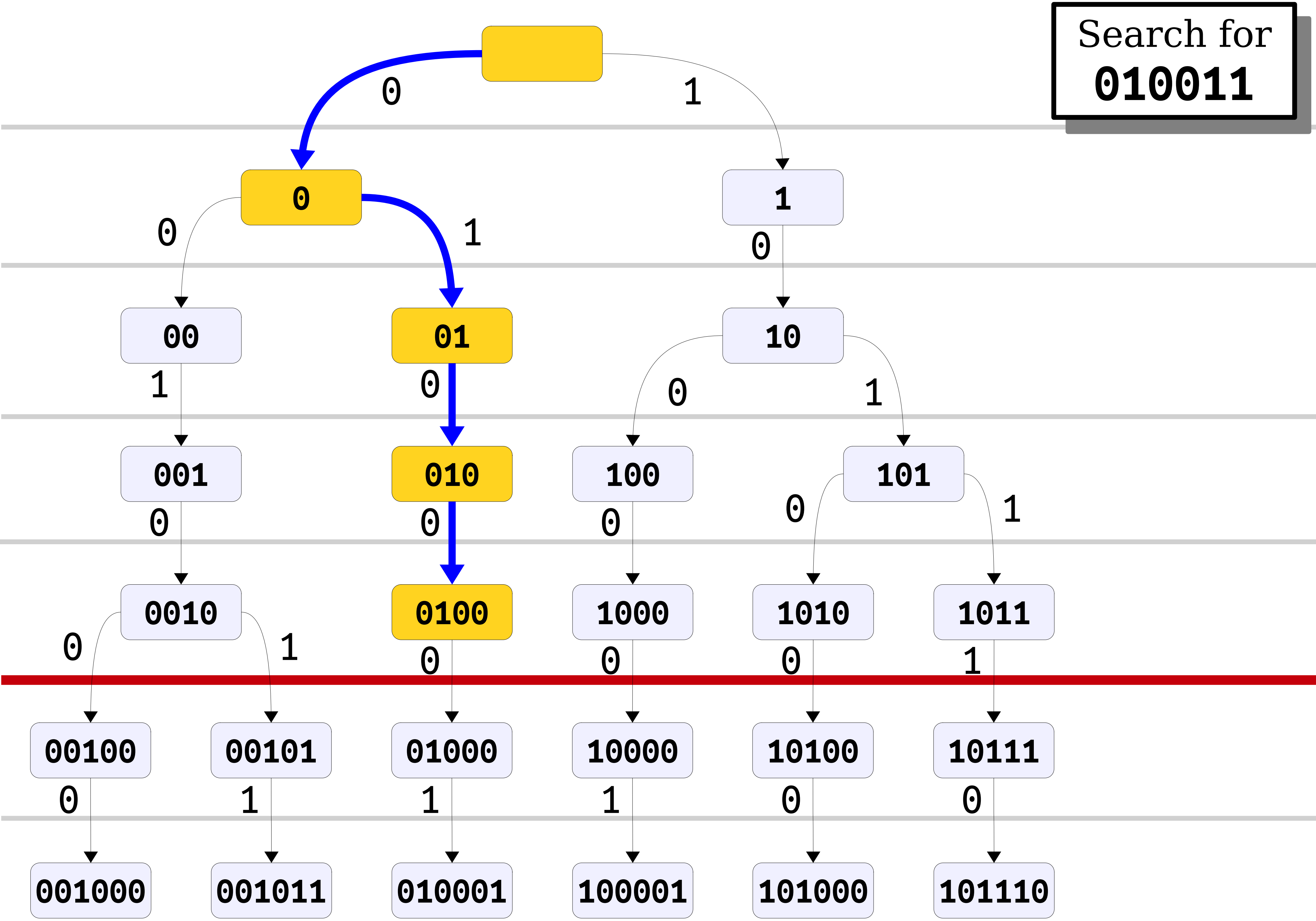
Search for
010011



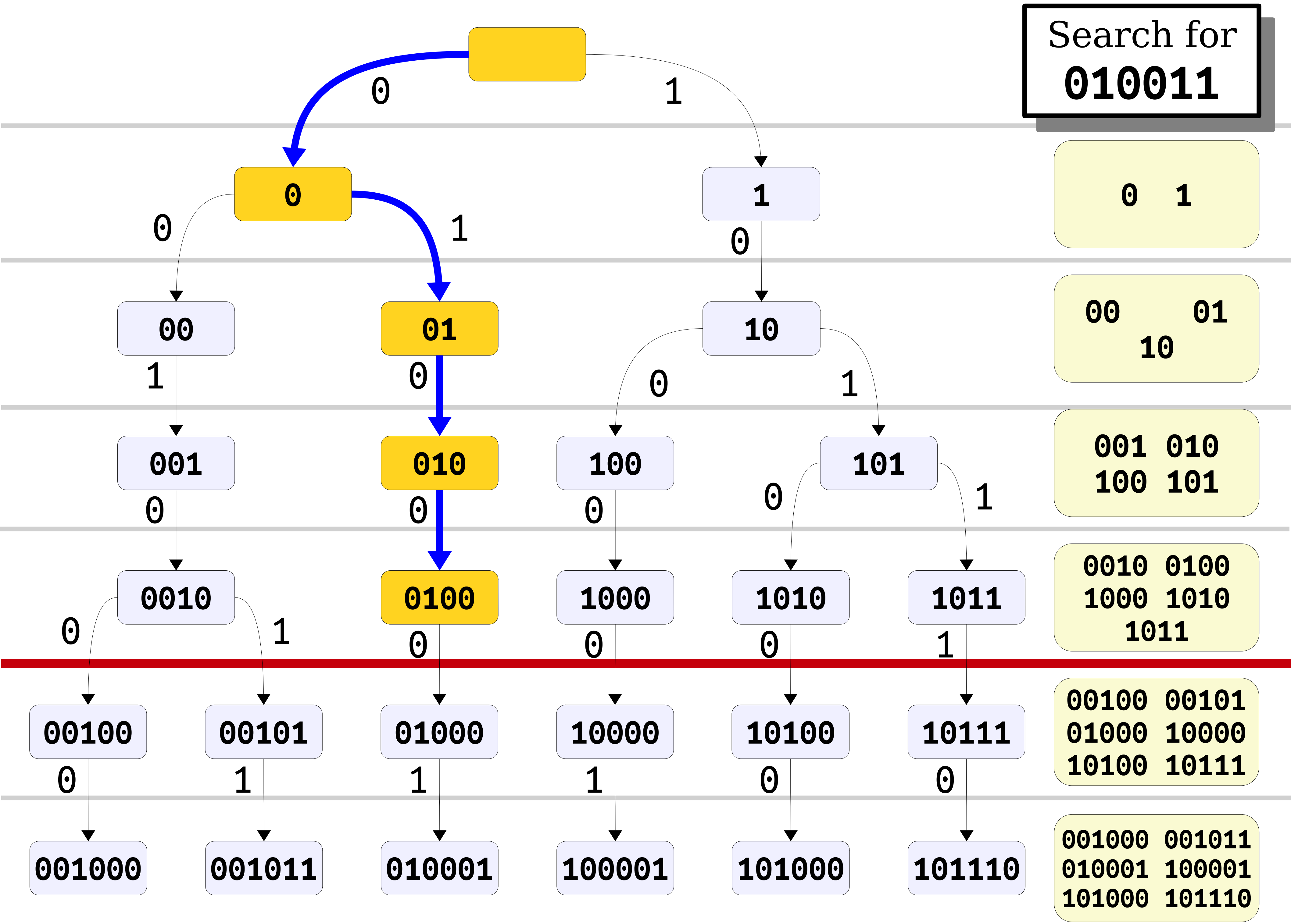
This assumes that we can tell whether there’s a trie node on each level that we’d be searching for. How can we determine that?



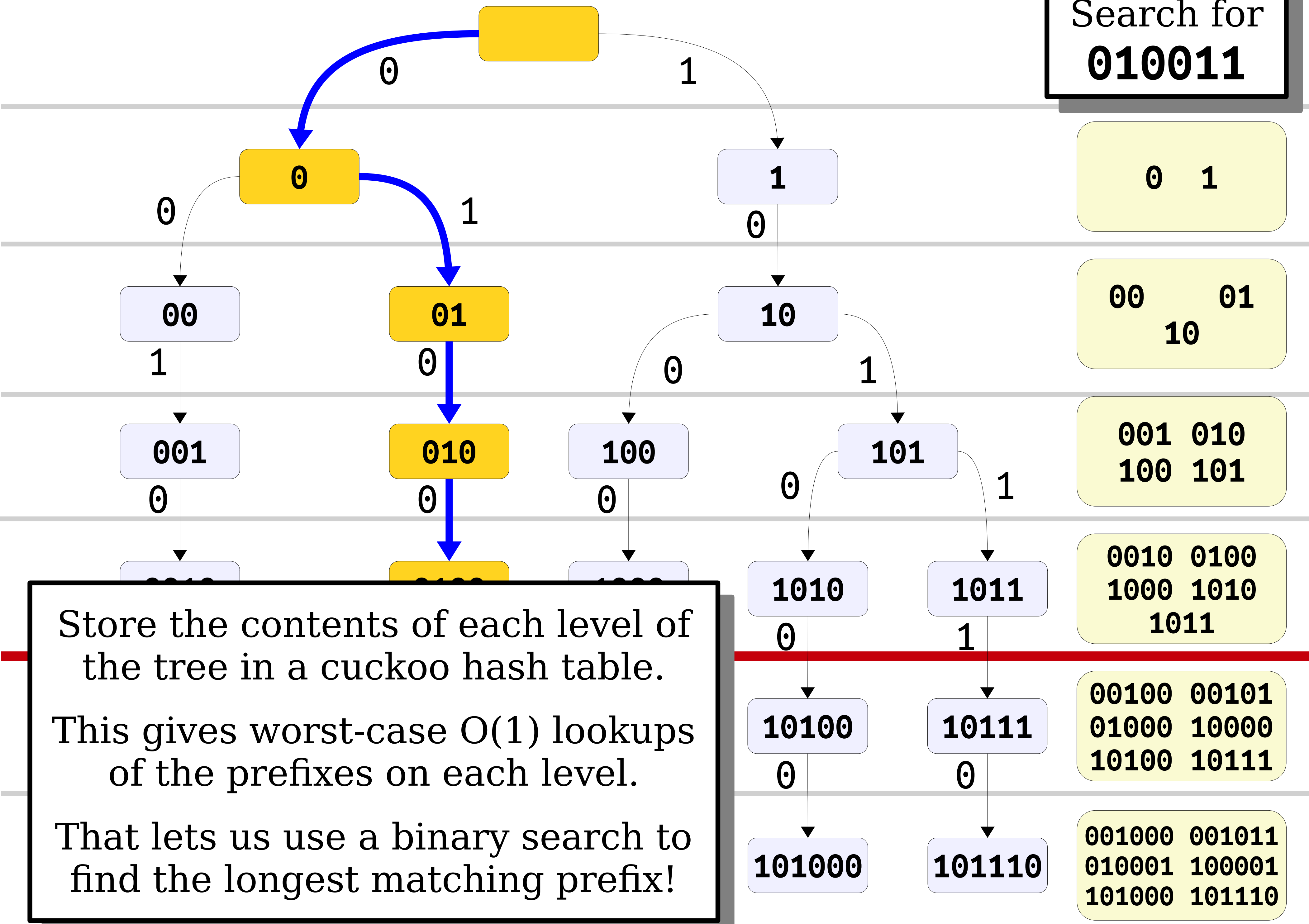
Search for
010011
0
1
0
1
0
1
0
0
1
0
0
0
1
0
1
0
0
0
1
0
1
1
0
0
001000
001011
010001
100001
101000
101110



Search for
010011
0
1
00
01
10
001
010
100
101
0010
0100
1000
1010
1011
00100
00101
01000
10000
10100
10111
001000
001011
010001
100001
101000
101110


Search for
010011
0
1
0
1
0
1
00
01
10
1
0
0
1
001
010
100
101
0
0
0
0
1
0010
0100
1000
1010
1011
0
1
0
0
0
1
00100
00101
01000
10000
10100
10111
0
1
1
1
0
0
001000
001011
010001
100001
101000
101110
0 1
00 01
10
001 010
100 101
0010 0100
1000 1010
1011
00100 00101
01000 10000
10100 10111
001000 001011
010001 100001
101000 101110

Search for **010011**



Store the contents of each level of the tree in a cuckoo hash table.

This gives worst-case O(1) lookups of the prefixes on each level.

That lets us use a binary search to find the longest matching prefix!

One Speedup

- ***Goal:*** Encode the trie so that we can do a binary search over its layers.
- ***One Solution:*** Store an array of cuckoo hash tables, one per layer of the trie, that holds all the nodes in that layer.
- Can now query, in worst-case time O(1), whether a node's prefix is present on a given layer.
- There are O(w) layers in the trie.
- Binary search will take worst-case time **O(log w)**. This is *much* better than O(log n) for any reasonable value of n.

Performing the Binary Search

- This binary search assumes that, given a number x and a length k, we can extract the first k bits of x in time O(1).
- Fortunately, we can do this!

x `11011100 10111011 11000100 11010110 11110011 01111011 11110000 10001100`

Performing the Binary Search

- This binary search assumes that, given a number x and a length k, we can extract the first k bits of x in time O(1).
- Fortunately, we can do this!

x	11011100 10111011 11000100 11010110 11110011 01111011 11110000 10001100
mask	11111111 11111111 11111111 11110000 00000000 00000000 00000000 00000000
prefix	11011100 10111011 11000100 11010000 00000000 00000000 00000000 00000000

```
uint64_t x      = /* … */;
uint64_t mask   = something magical;
uint64_t prefix = x & mask;
```

Performing the Binary Search

- This binary search assumes that, given a number x and a length k, we can extract the first k bits of x in time O(1).
- Fortunately, we can do this!

x	11011100 10111011 11000100 11010110 11110011 01111011 11110000 10001100
mask	
prefix	11011100 10111011 11000100 11010000 00000000 00000000 00000000 00000000

```
uint64_t x      = /* … */;
uint64_t mask   = something magical;
uint64_t prefix = x & mask;
```

Performing the Binary Search

- This binary search assumes that, given a number x and a length k, we can extract the first k bits of x in time O(1).
- Fortunately, we can do this!

x	**11011100 10111011 11000100 1101**0110 11110011 01111011 11110000 10001100
mask	00000000 00000000 00000000 000**1**0000 00000000 00000000 00000000 00000000
prefix	**11011100 10111011 11000100 1101**0000 00000000 00000000 00000000 00000000

```
uint64_t x      = /* … */;
uint64_t mask   =  (uint64_t(1) << (64 - k));
uint64_t prefix = x & mask;
```

Performing the Binary Search

- This binary search assumes that, given a number x and a length k, we can extract the first k bits of x in time O(1).
- Fortunately, we can do this!

x	11011100 10111011 11000100 11010110 11110011 01111011 11110000 10001100
mask	11111111 11111111 11111111 11101111 11111111 11111111 11111111 11111111
prefix	11011100 10111011 11000100 11010000 00000000 00000000 00000000 00000000

```
uint64_t x      = /* … */;
uint64_t mask   = ~(uint64_t(1) << (64 - k));
uint64_t prefix = x & mask;
```

Performing the Binary Search

- This binary search assumes that, given a number *x* and a length *k*, we can extract the first *k* bits of *x* in time O(1).
- Fortunately, we can do this!

x	11011100 10111011 11000100 11010110 11110011 01111011 11110000 10001100
mask	11111111 11111111 11111111 11110000 00000000 00000000 00000000 00000000
prefix	11011100 10111011 11000100 11010000 00000000 00000000 00000000 00000000

```
-x = ~x + 1
```

Thanks, CS107!

```
uint64_t x      = /* … */;
uint64_t mask   = ~(uint64_t(1) << (64 - k)) + 1;
uint64_t prefix = x & mask;
```

Performing the Binary Search

- This binary search assumes that, given a number *x* and a length *k*, we can extract the first *k* bits of *x* in time O(1).
- Fortunately, we can do this!

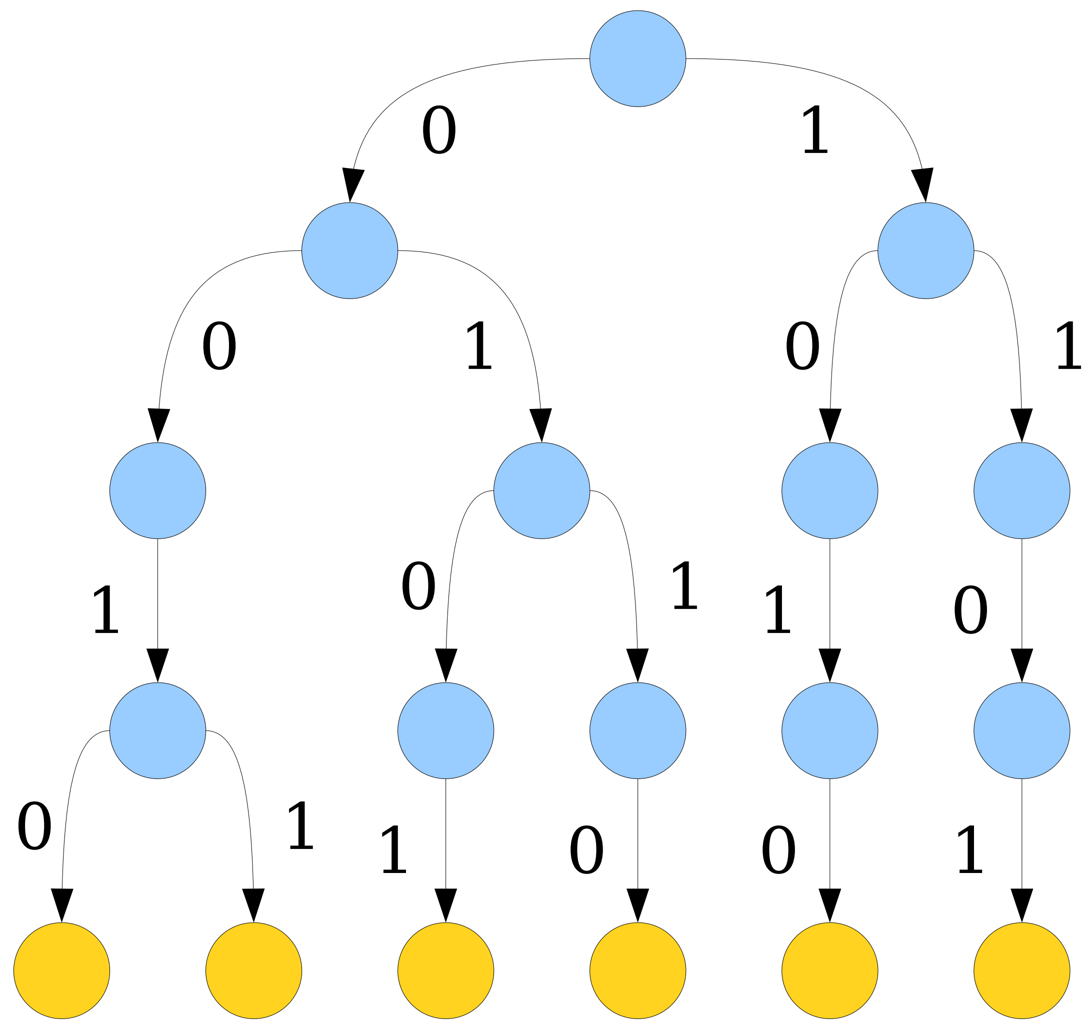
x	11011100 10111011 11000100 11010110 11110011 01111011 11110000 10001100
mask	11111111 11111111 11111111 11110000 00000000 00000000 00000000 00000000
prefix	11011100 10111011 11000100 110100

There's an edge case to handle here for $k = 0$, but that's easily special-cased. Let me know if there's a way to avoid this!

```
uint64_t x      = /* … */;
uint64_t mask   = -(uint64_t(1) << (64 - k));
uint64_t prefix = x & mask;
```

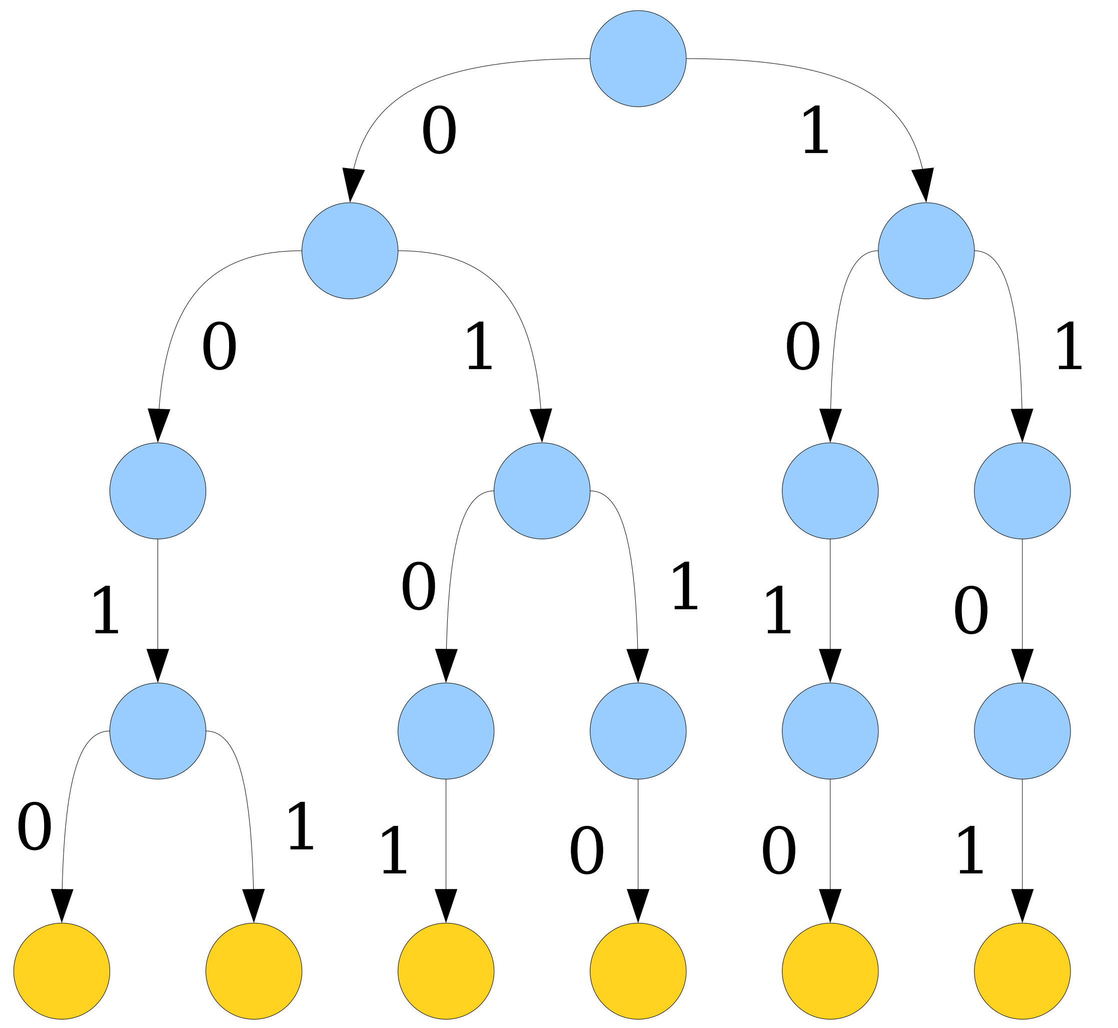
Finding Successors

- We can now find the node where the successor search would initially arrive in time O(log *w*).
- At this point, we'd normally back up until we find a branching node where we can follow a 1 child pointer, then descend from there to the leaves.
- This will take time O(*w*).
- ***Can we do better?***



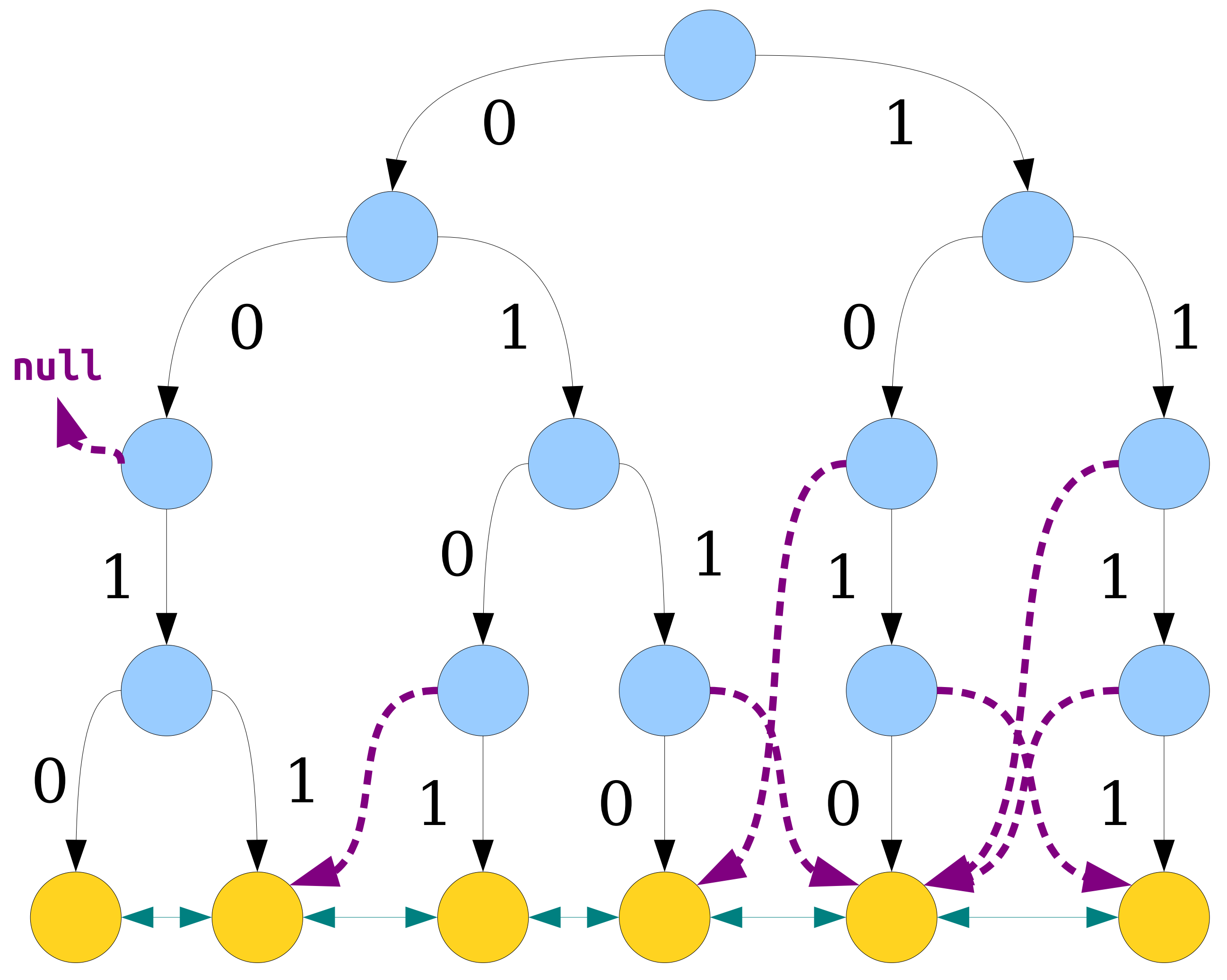
Finding Successors

- ***Claim:*** If the binary search terminates at a node *v*, that node must have at most one child.
- If it doesn't, it has both a 0 child and a 1 child, so there's a longer prefix that can be matched.
- ***Idea:*** Steal the missing pointers and use them to speed up successor and predecessor searches.



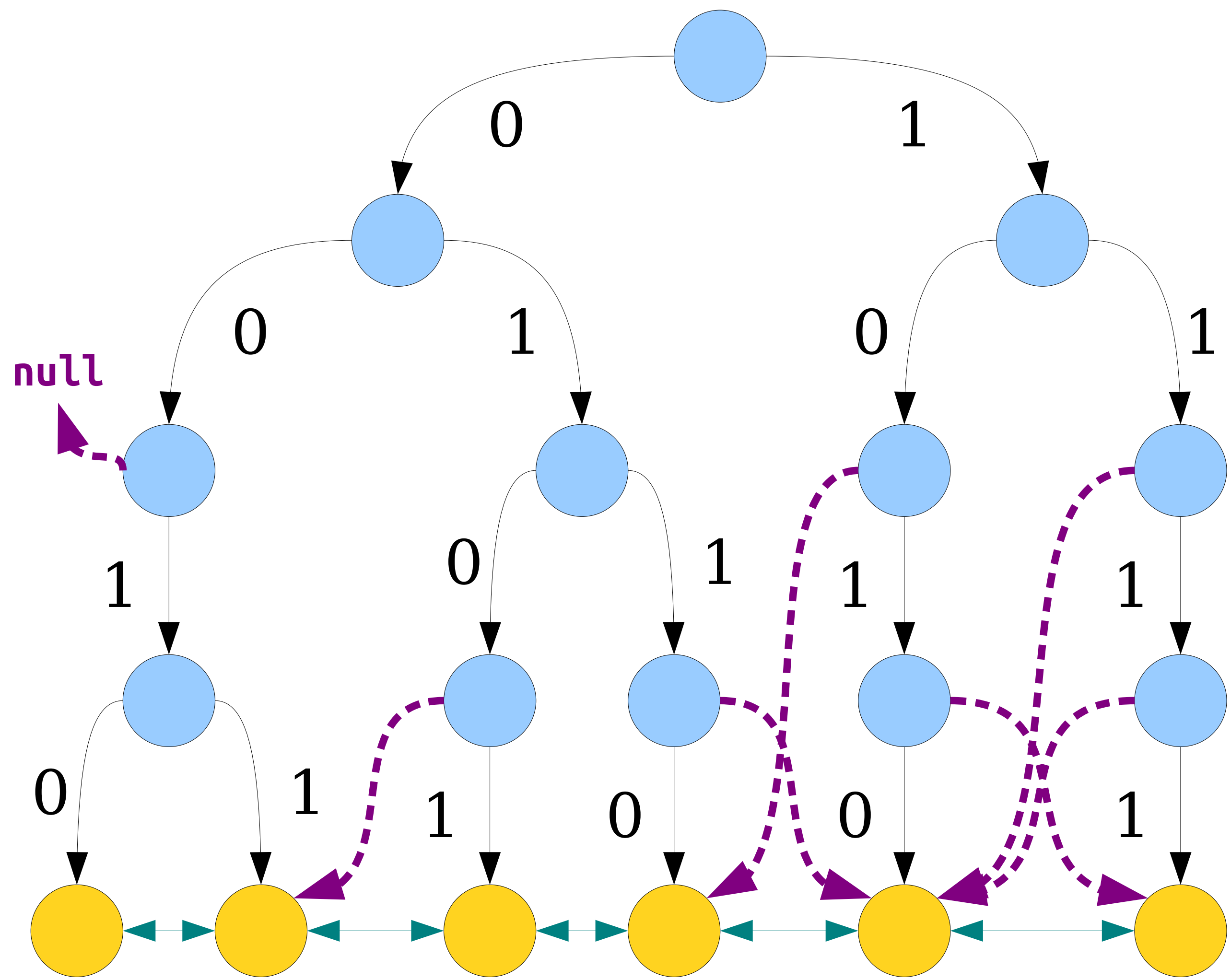
x-Fast Tries

- An ***x-fast trie*** is a modified binary trie.
- Each missing 1 pointer points to a node's successor.
- Each missing 0 pointer points to a node's predecessor.
- Each layer of the tree is stored in a cuckoo hash table for fast binary search.



x-Fast Tries

- ***Claim:*** Can determine ***successor***(*x*) in time O(log *w*).
- Binary search for the longest prefix of *x*.
- If that node has a missing 1 pointer, it points to the successor.
- Otherwise, it has a missing 0 pointer. Follow it to a leaf, then follow the leaf's 1 pointer.

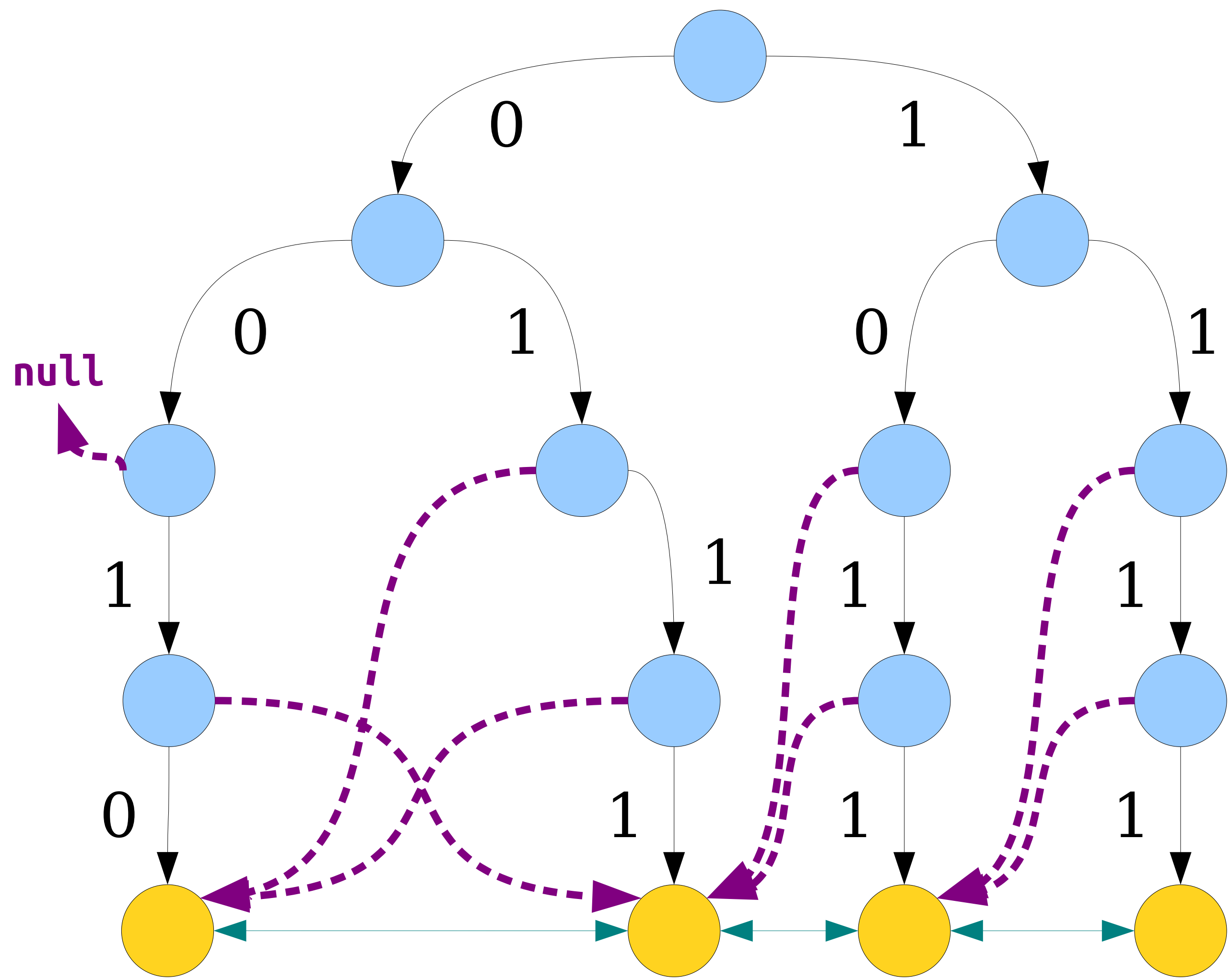


x-Fast Trie Maintenance

- Based on what we've seen:
  - ***lookup*** takes worst-case time O(1).
  - ***successor*** and ***predecessor*** queries take worst-case time O(log w).
  - ***min*** and ***max*** can be done in time O(1), assuming we cache those values.
- How efficiently can we support ***insert*** and ***delete***?

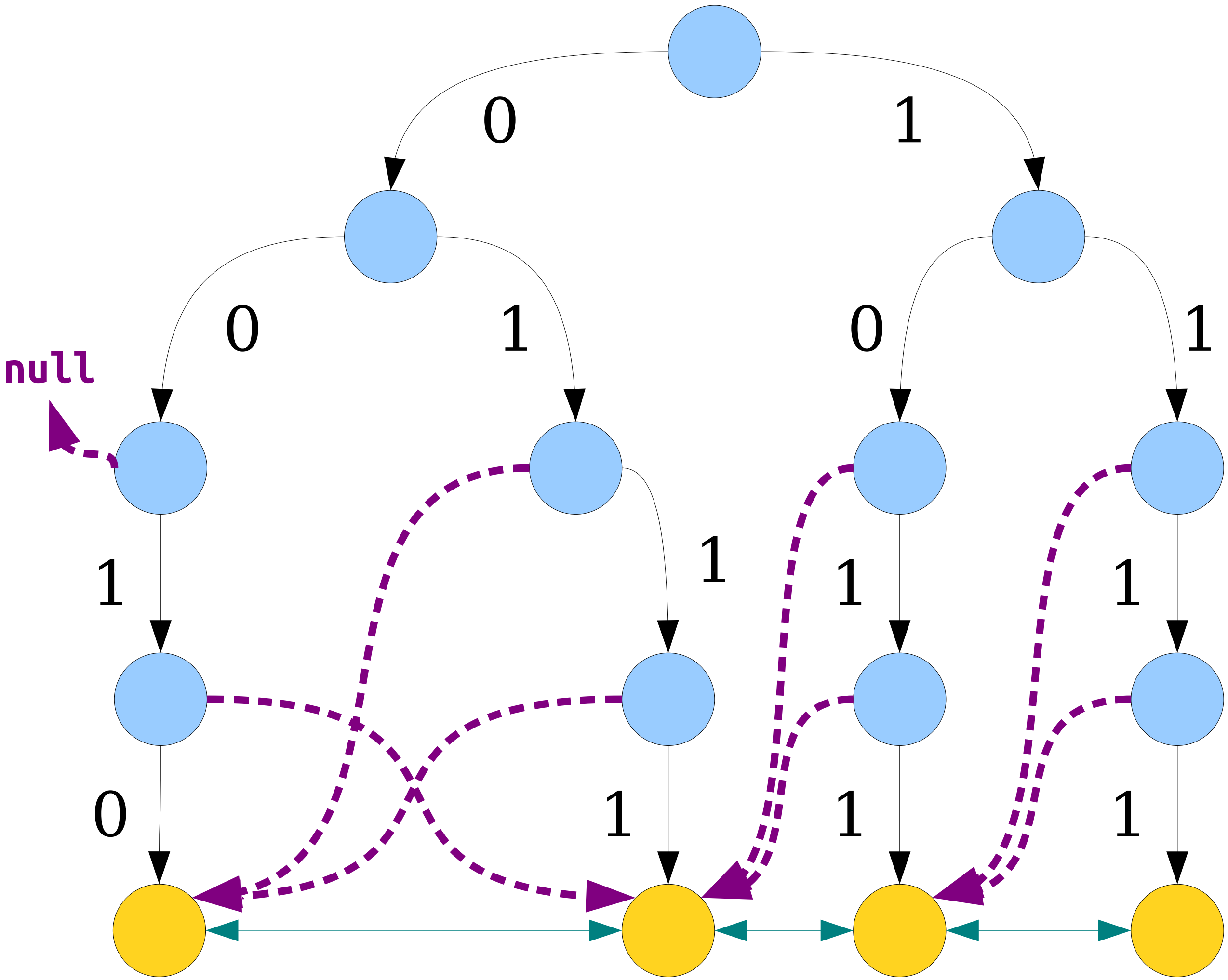
x-Fast Tries

- If we ***insert***(x), we need to
  - add some new nodes to the trie;
  - wire x into the doubly-linked list of leaves; and
  - update the thread pointers to include x.
- Worst-case will be **$\Omega(w)$** due to the first and third steps.



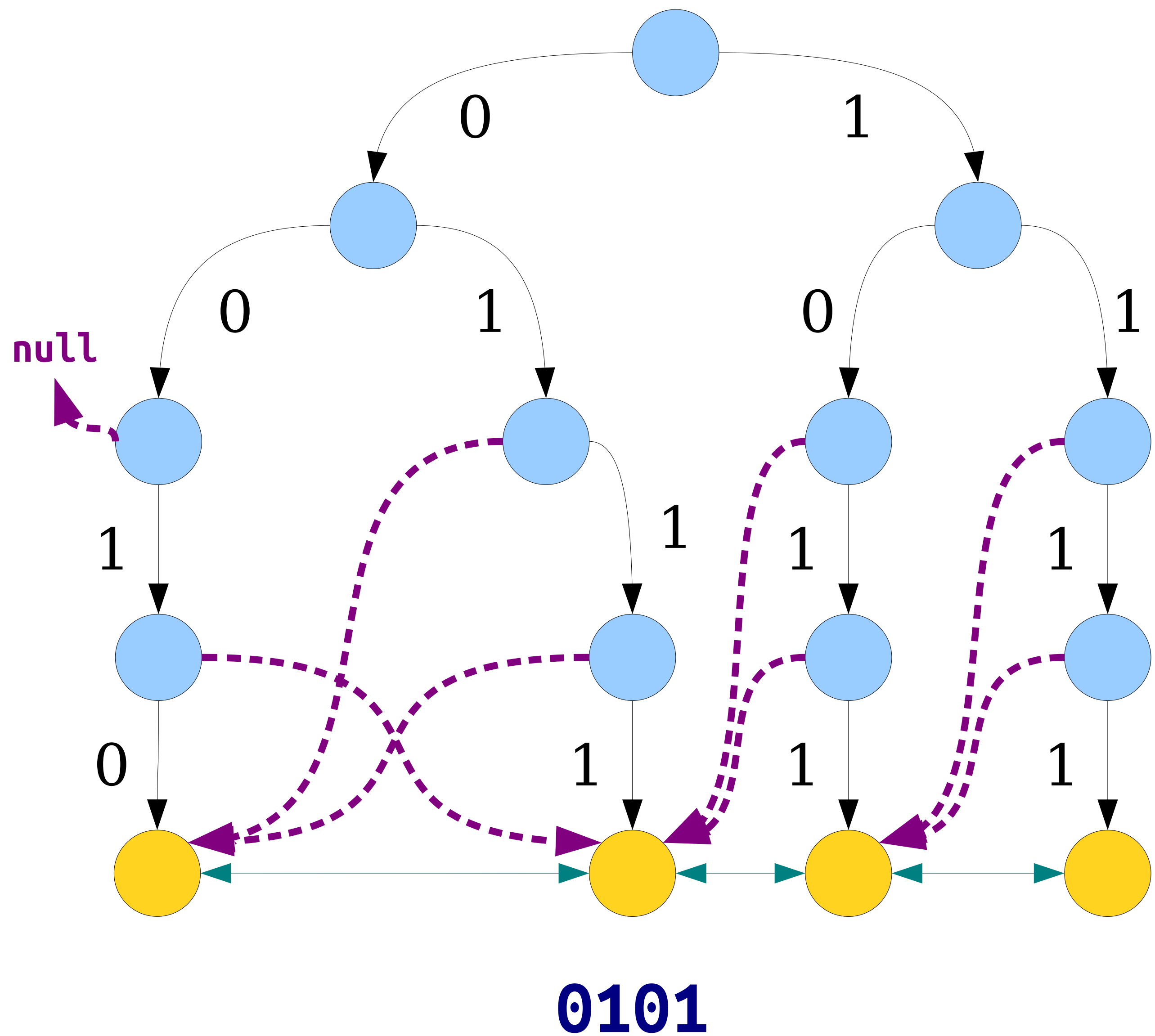
x-Fast Tries

- Here is an (amortized, expected) O(*w*)-time algorithm for ***insert***(*x*):
  - Find ***successor***(*x*).
  - Add *x* to the trie.
  - Using the successor from before, wire *x* into the linked list.
  - Walk up from *x*, its successor, and its predecessor and update threads.



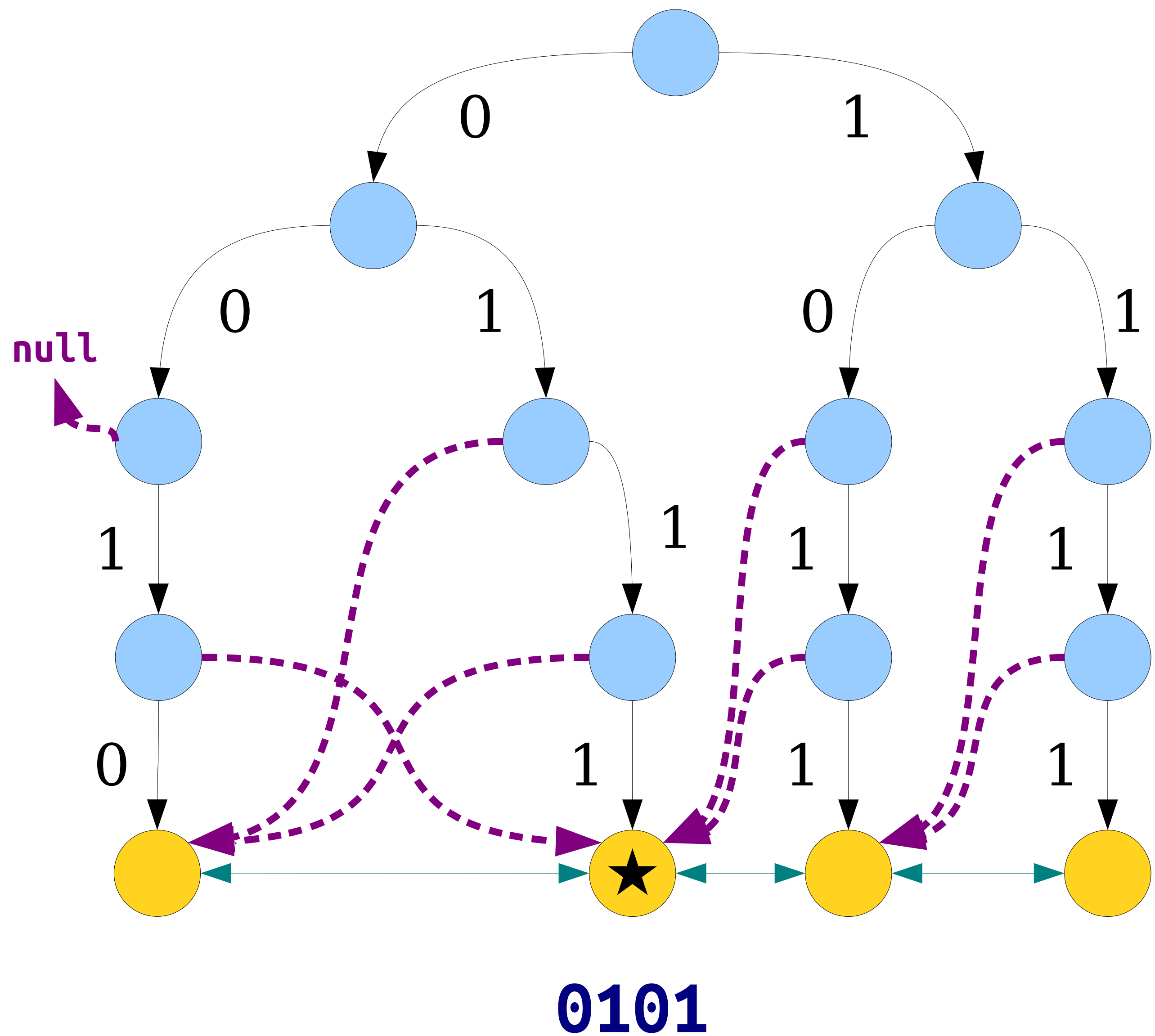
x-Fast Tries

- Here is an (amortized, expected) O(w)-time algorithm for ***insert***(x):
  - Find ***successor***(x).
  - Add x to the trie.
  - Using the successor from before, wire x into the linked list.
  - Walk up from x, its successor, and its predecessor and update threads.



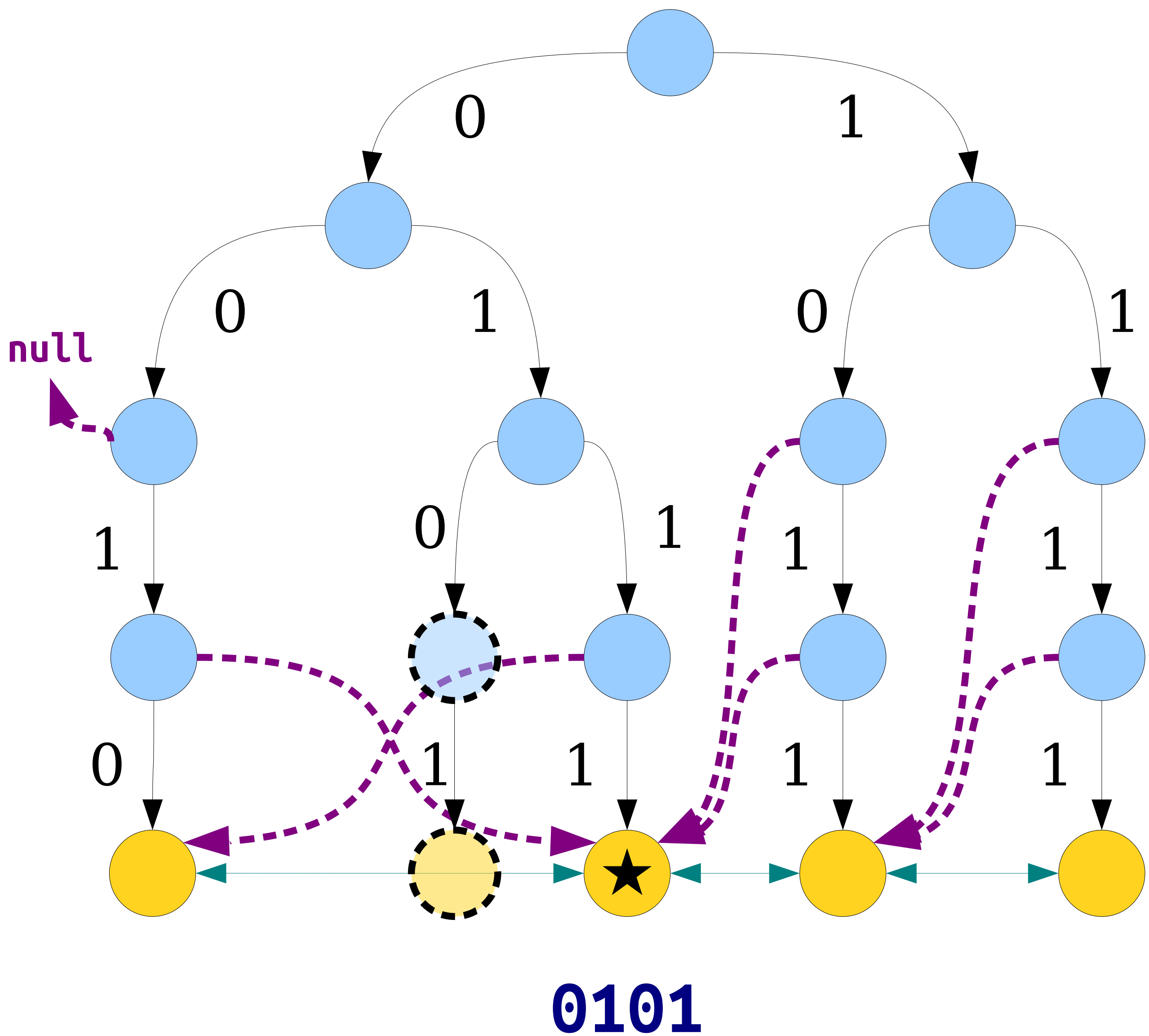
x-Fast Tries

- Here is an (amortized, expected) O(*w*)-time algorithm for ***insert***(*x*):
  - Find ***successor***(*x*).
  - Add *x* to the trie.
  - Using the successor from before, wire *x* into the linked list.
  - Walk up from *x*, its successor, and its predecessor and update threads.



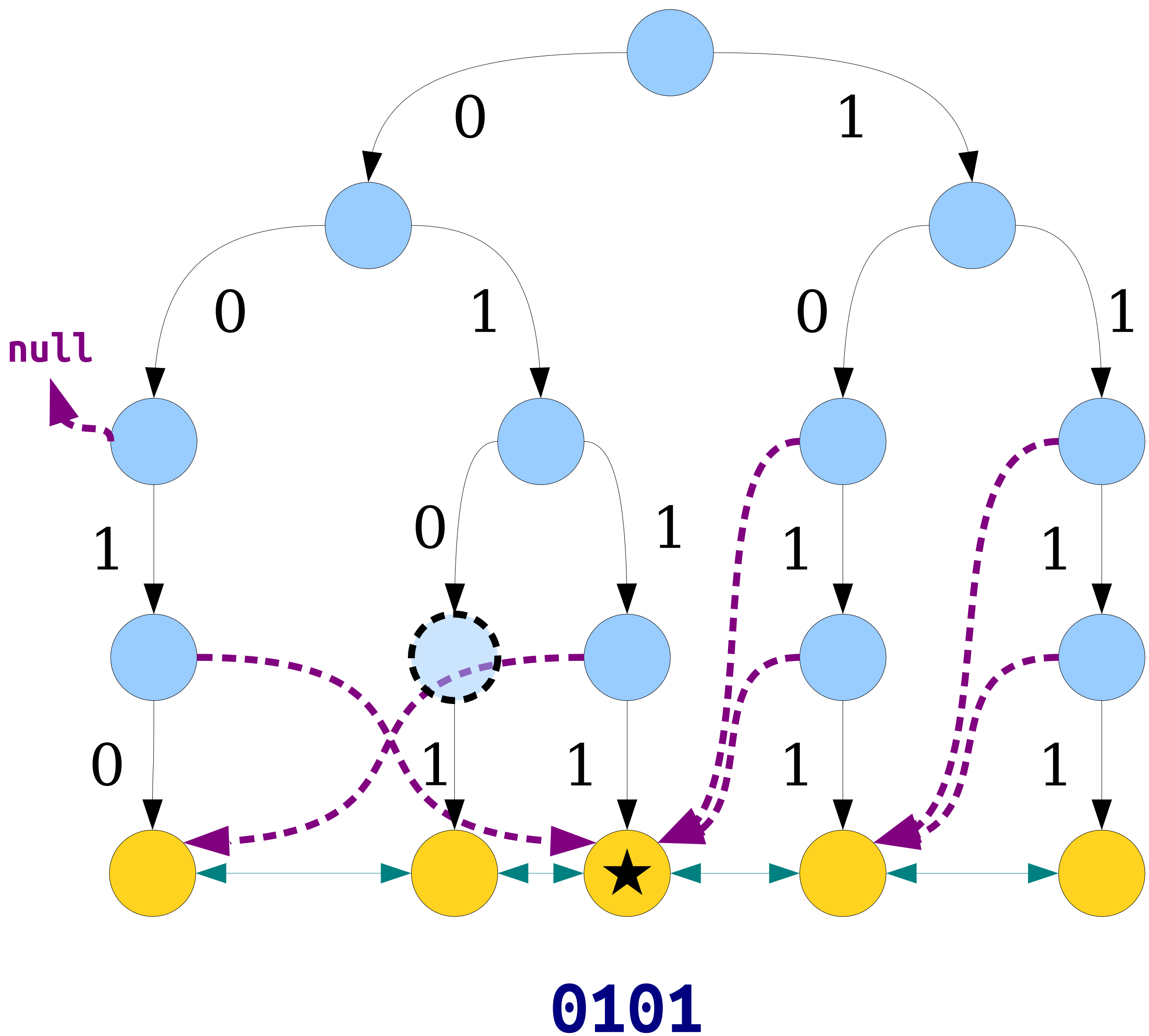
x-Fast Tries

- Here is an (amortized, expected) O(*w*)-time algorithm for ***insert***(*x*):
    - Find ***successor***(*x*).
    - Add *x* to the trie.
    - Using the successor from before, wire *x* into the linked list.
    - Walk up from *x*, its successor, and its predecessor and update threads.



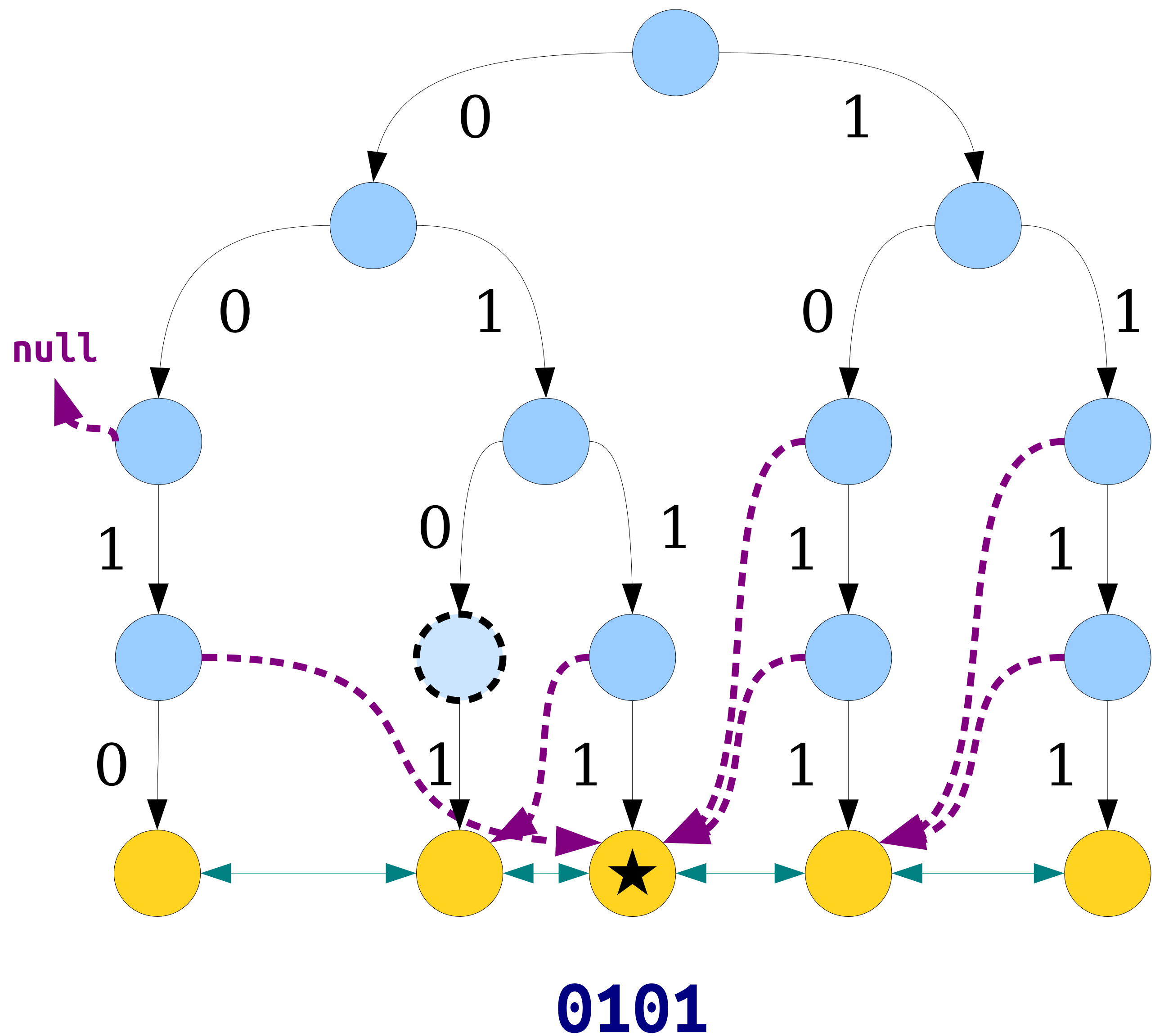
x-Fast Tries

- Here is an (amortized, expected) O(*w*)-time algorithm for ***insert***(*x*):
  - Find ***successor***(*x*).
  - Add *x* to the trie.
  - Using the successor from before, wire *x* into the linked list.
  - Walk up from *x*, its successor, and its predecessor and update threads.



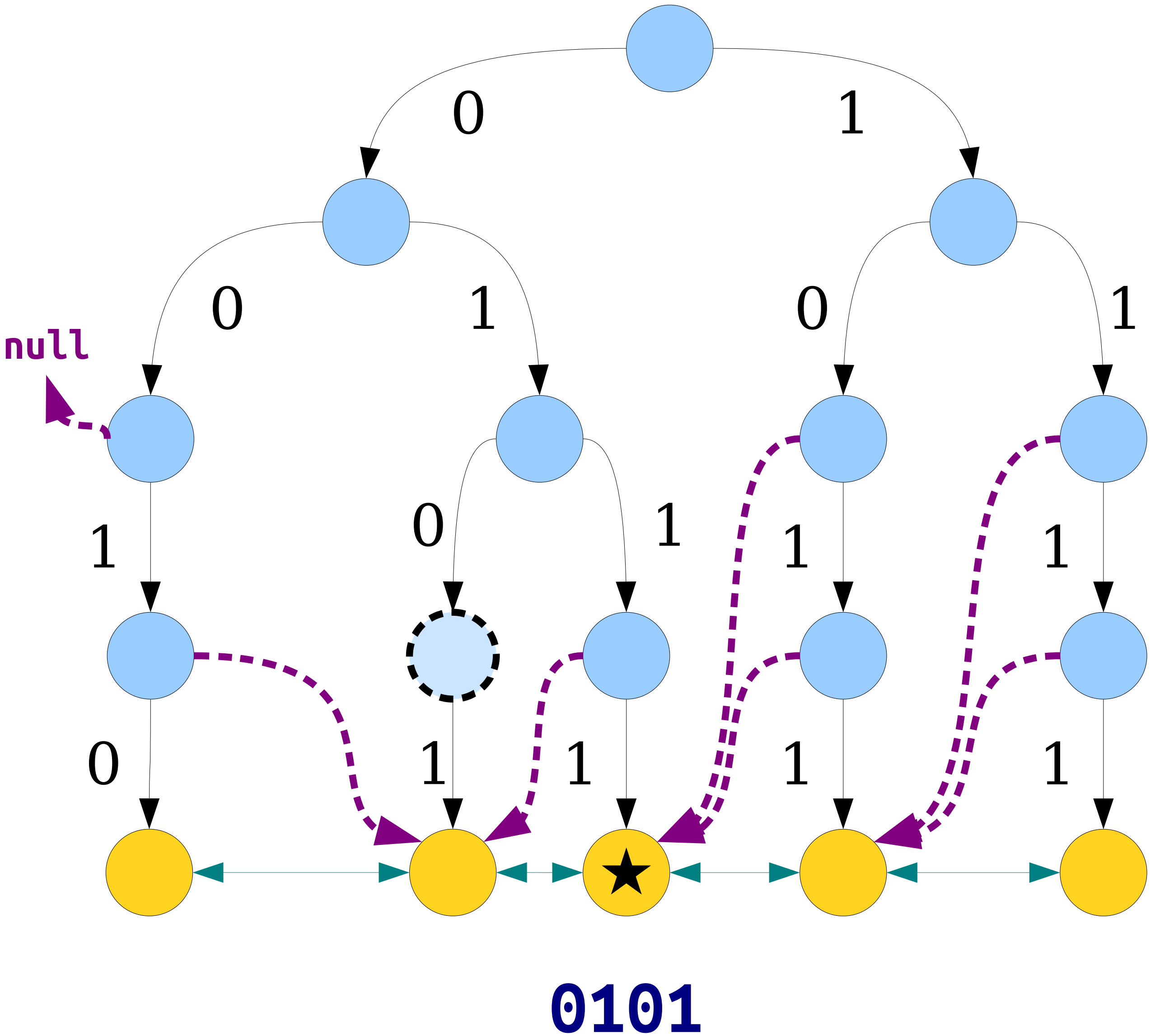
x-Fast Tries

- Here is an (amortized, expected) O(*w*)-time algorithm for ***insert***(*x*):
  - Find ***successor***(*x*).
  - Add *x* to the trie.
  - Using the successor from before, wire *x* into the linked list.
  - Walk up from *x*, its successor, and its predecessor and update threads.



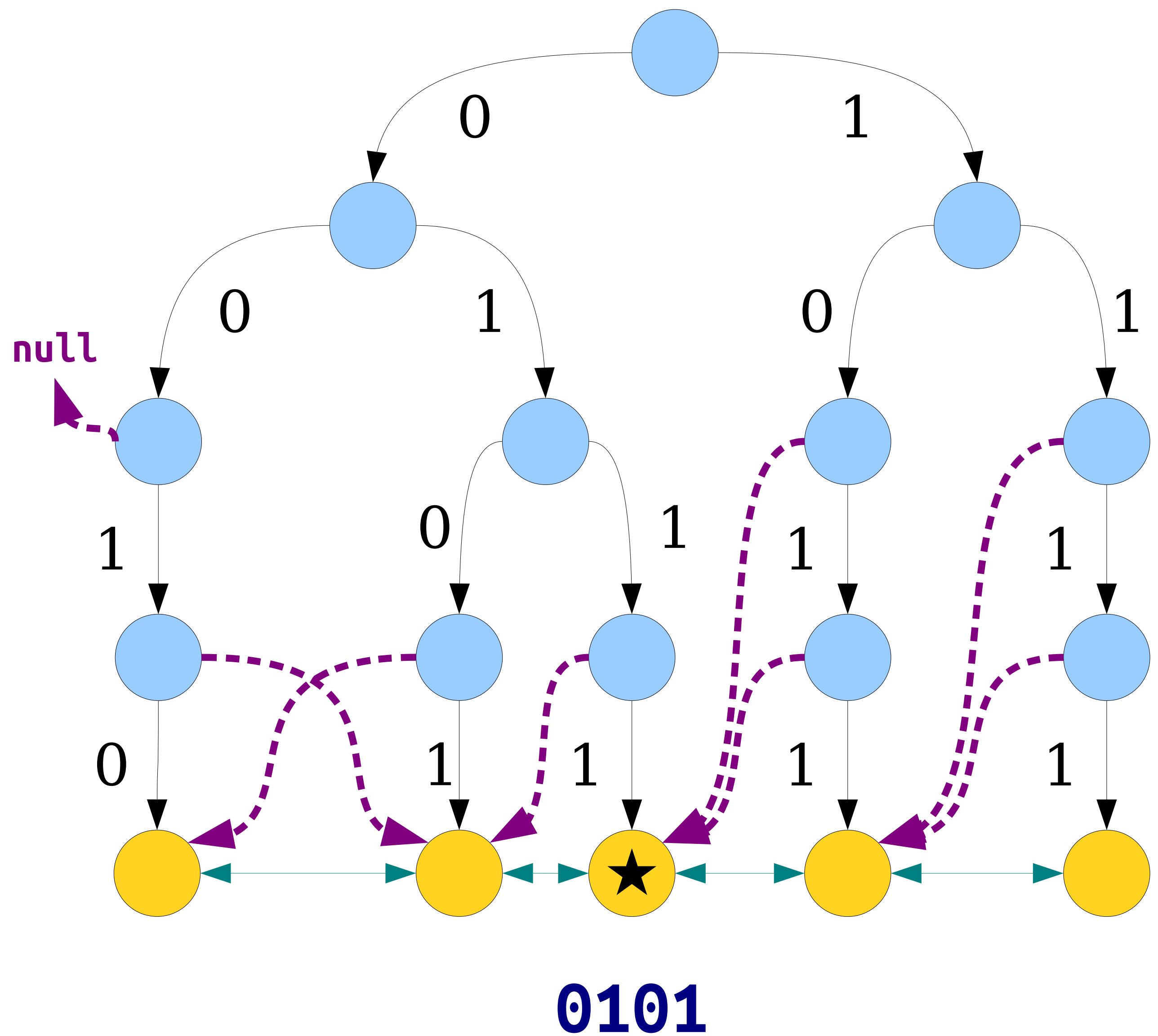
x-Fast Tries

- Here is an (amortized, expected) O(*w*)-time algorithm for ***insert***(*x*):
  - Find ***successor***(*x*).
  - Add *x* to the trie.
  - Using the successor from before, wire *x* into the linked list.
  - Walk up from *x*, its successor, and its predecessor and update threads.



x-Fast Tries

- Here is an (amortized, expected) O(*w*)-time algorithm for ***insert***(*x*):
  - Find ***successor***(*x*).
  - Add *x* to the trie.
  - Using the successor from before, wire *x* into the linked list.
  - Walk up from *x*, its successor, and its predecessor and update threads.

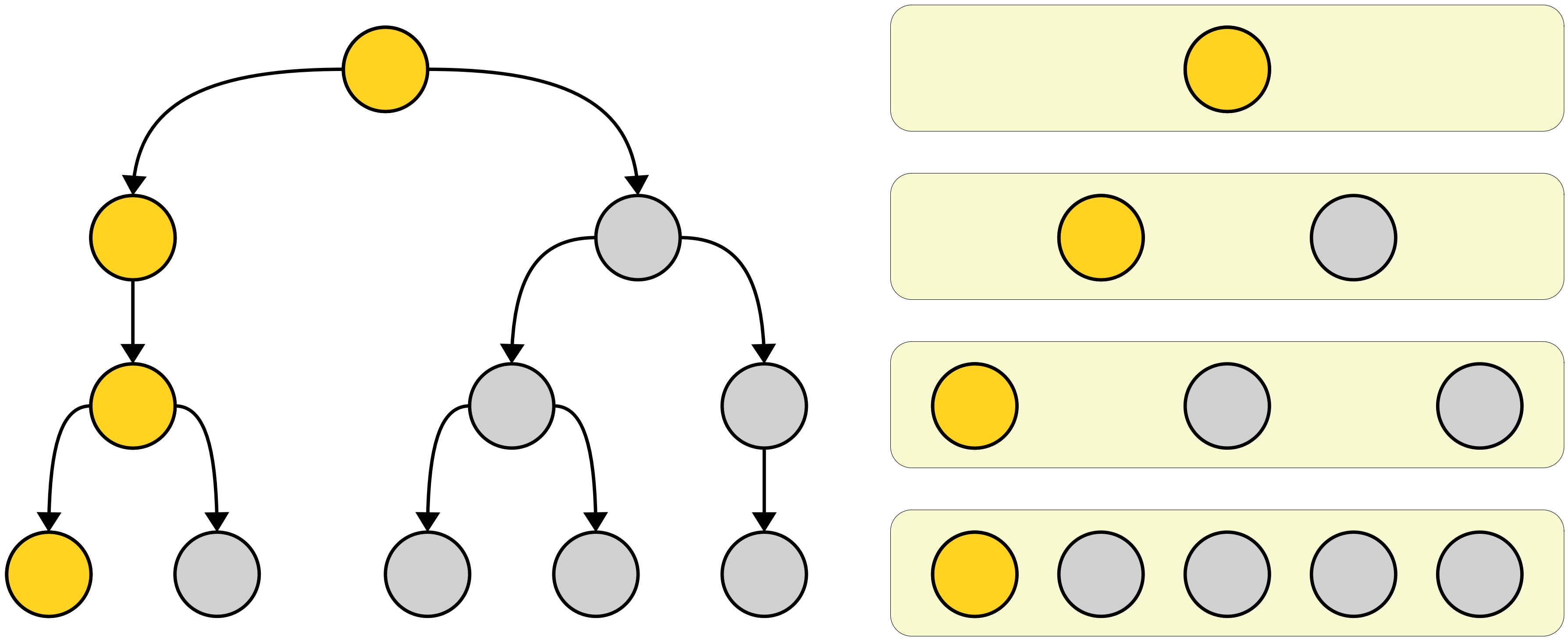


Deletion

- To ***delete***(x), we need to
  - Remove x from the trie.
  - Splice x out of its linked list.
  - Update thread pointers from x's former predecessor and successor.
- Runs in expected, amortized time **O(*w*)**.
- Full details are left as a proverbial Exercise to the Reader. ☺

Space Usage

- Each leaf node (item stored in the x-fast trie) contributes at most $O(w)$ nodes in the trie and at most $O(w)$ entries into the hash tables.
- Total space: **$O(nw)$**.



Where We Stand

- Right now, we have a reasonably fast data structure for storing a sorted set of integers.
- If we have a *static* set of integers that we want to make lots of queries on, this is pretty good as-is!
- As you'll see, though, we can make this even better with some kitchen sink techniques.

x-Fast Trie:

- ***lookup***: O(1)
- ***insert***: $O(w)^*$
- ***delete***: $O(w)^*$
- ***max***: O(1)
- ***succ***: $O(\log w)$
- ***is-empty***: O(1)
- Space: $O(nw)$

* Expected, amortized

Where We Stand

- Where is there room for improvement in this data structure?
- Ideally, we'd like to improve these highlighted costs, which are places where this structure currently is beaten by a standard BST.

x-Fast Trie:

- ***lookup***: O(1)
- ***insert***: **O(*w*)***
- ***delete***: **O(*w*)***
- ***max***: O(1)
- ***succ***: O(log *w*)
- ***is-empty***: O(1)
- Space: **O(*nw*)**

* Expected, amortized

Shaving Off Logs

- We're essentially at a spot where we need to shave off a factor of w from a couple of operations.
- Figure that w is kinda sorta ish like log n, so this is like shaving off a log factor.
- ***Question:*** What techniques have we developed so far to do this?

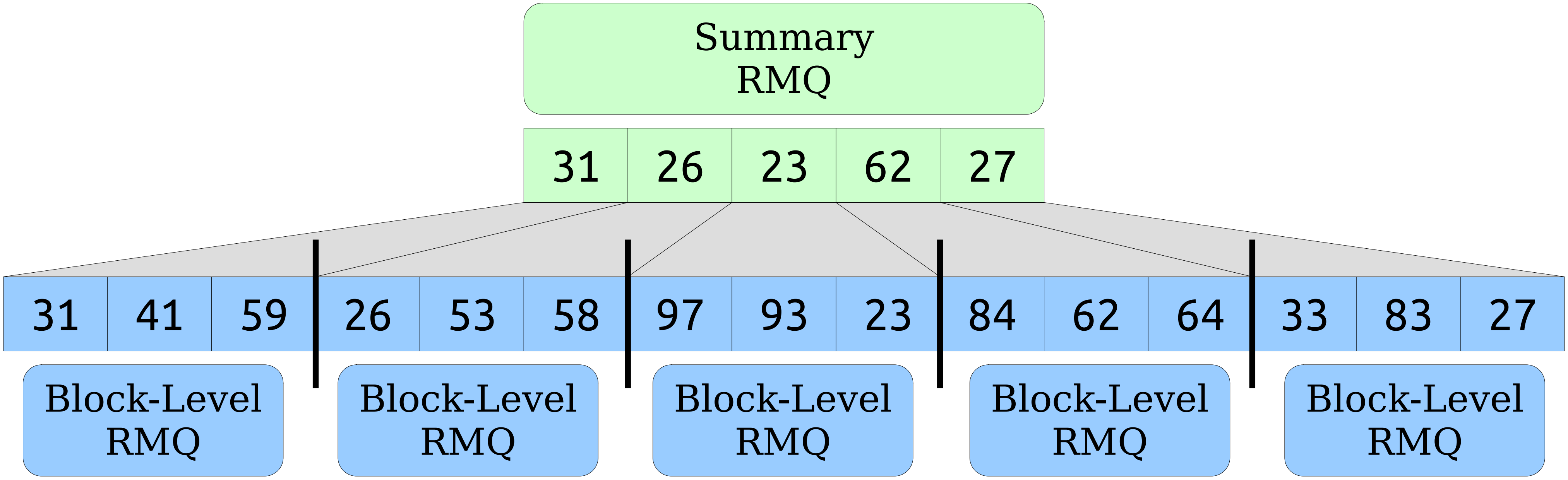
x-Fast Trie:

- ***lookup***: O(1)
- ***insert***: **O(w)** *
- ***delete***: **O(w)** *
- ***max***: O(1)
- ***succ***: O(log w)
- ***is-empty***: O(1)
- Space: **O(nw)**

* Expected, amortized

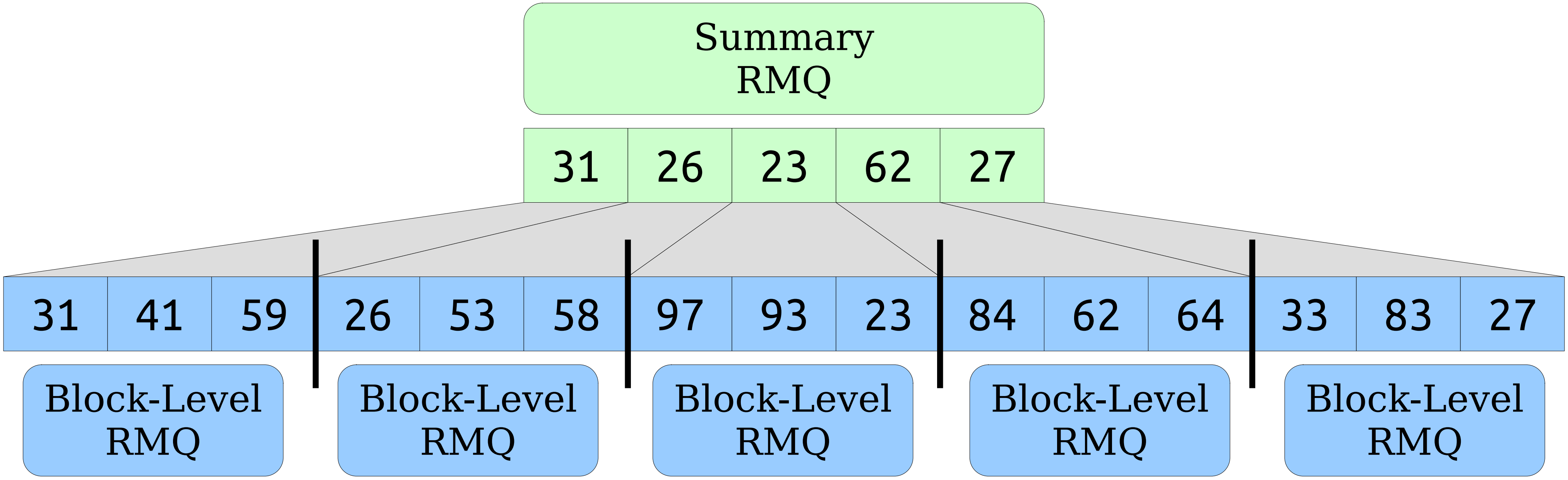
Two-Level Structures

- Think back to the hybrid approach we used for solving RMQ.
- It consisted of a two-tiered structure:
  - A bunch of small, lower-level structures that each solve the problem in small cases.
  - A single, larger, top-level structure that helps aggregate those solutions together.



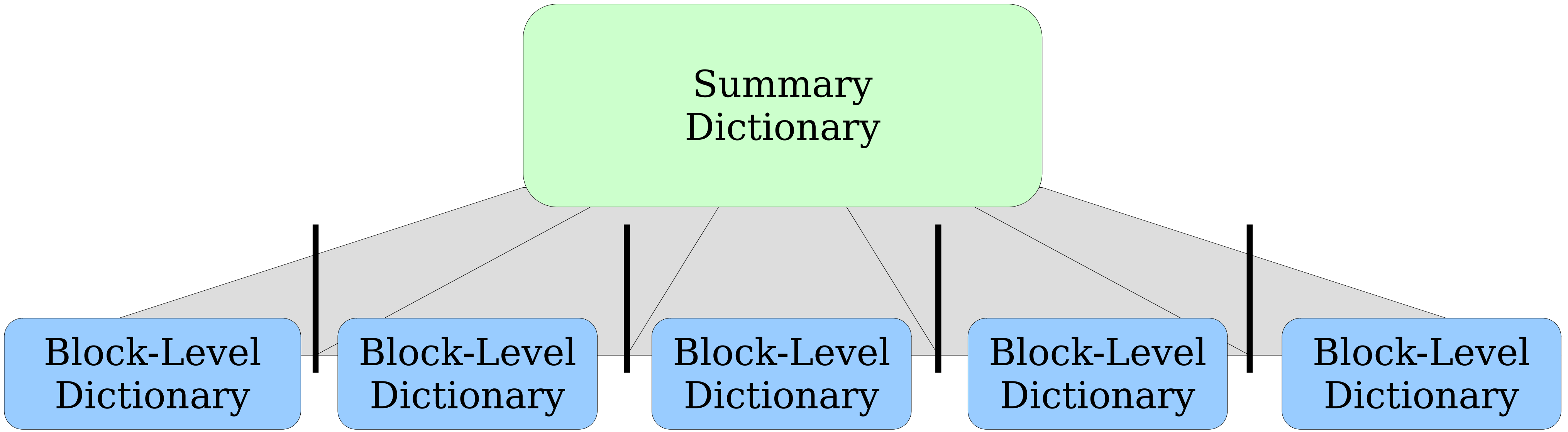
Two-Level Structures

- ***Main Idea:*** Partition the input into blocks that are really, really small.
- Small blocks make the block-level structures run quickly.
- Assuming they're not "too small," small blocks reduce the size of the inputs to the summary as well.



The Idea

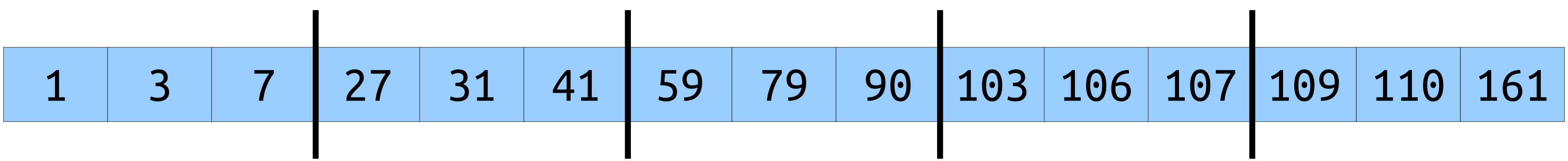
- Build a two-level ordered dictionary out of existing ordered dictionaries.
- Split the keys apart into logarithmic-sized blocks.
- Build ordered dictionaries for each of the block-level dictionaries.
- Build a summary dictionary to aggregate the blocks together.



The *y*-Fast Trie

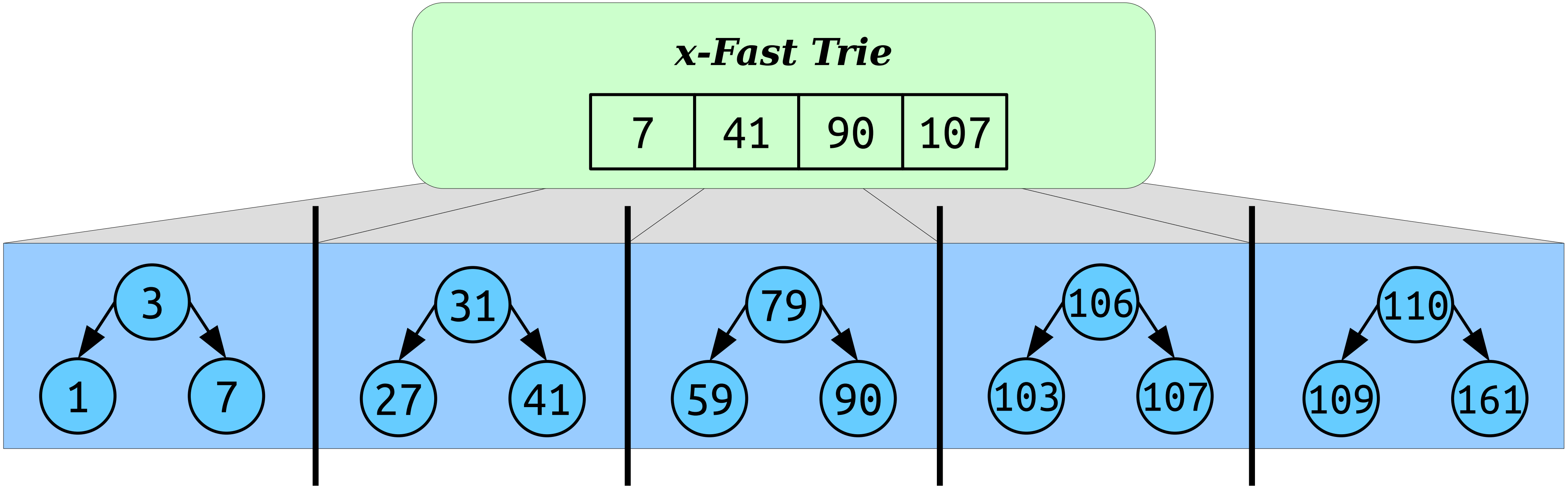
The Setup

- For now, assume all keys are given to us in advance, in sorted order.
- Split the keys apart into blocks of size $\Theta(w)$ and store them in balanced BSTs.



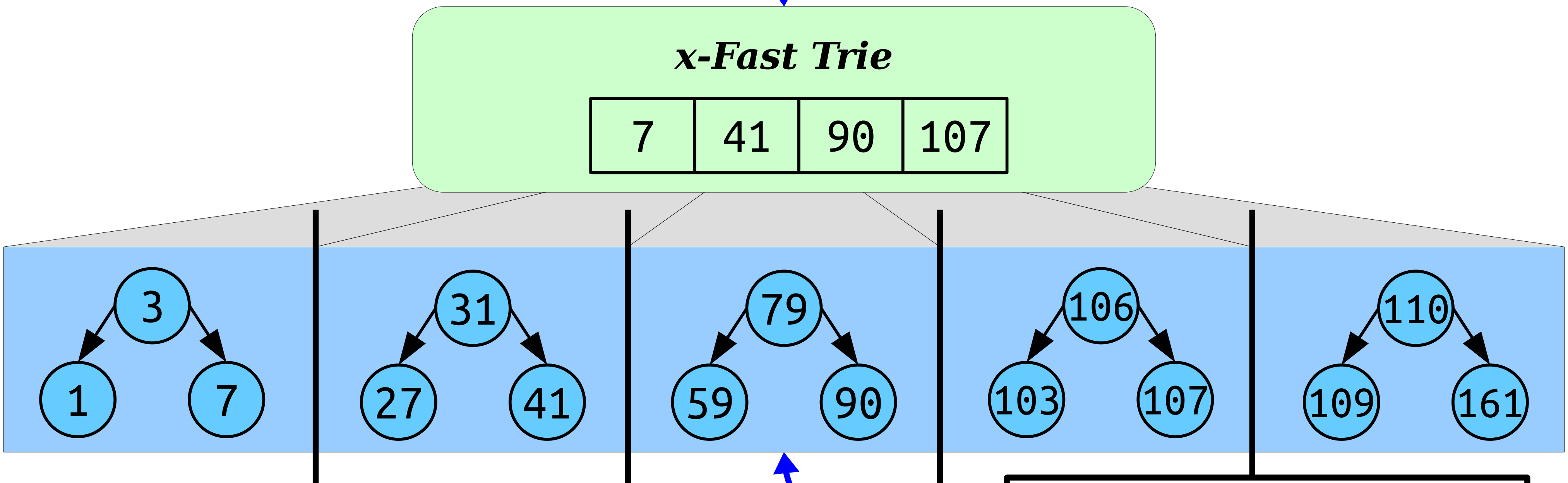
The Setup

- For now, assume all keys are given to us in advance, in sorted order.
- Split the keys apart into blocks of size $\Theta(w)$ and store them in balanced BSTs.
- Create a summary x-fast trie that stores the maximum key from each block but the last.



The Setup

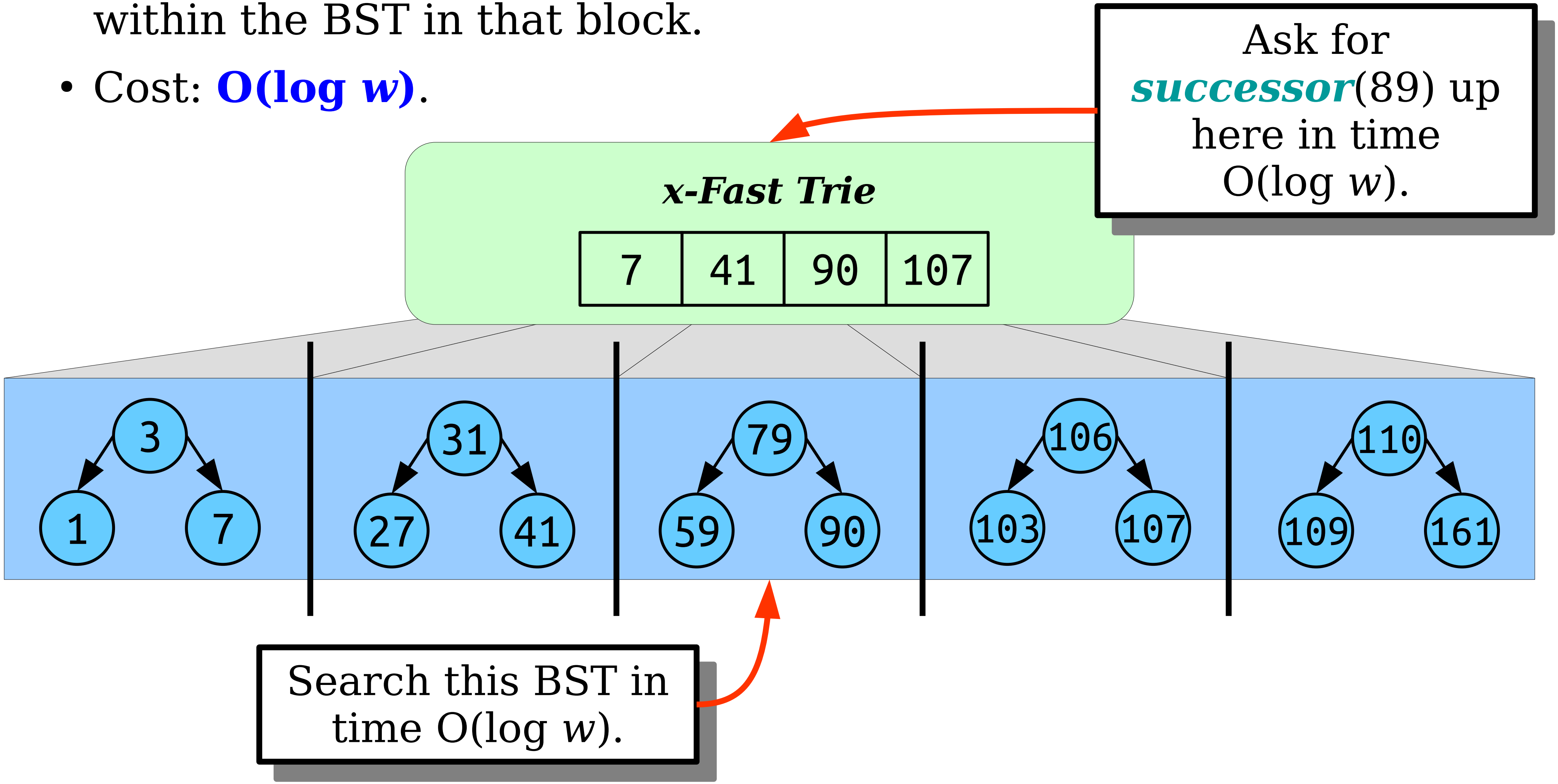
This summary structure is used for ***routing information*** so we know which BSTs to look at.



These trees are used for ***storage*** of the actual items.

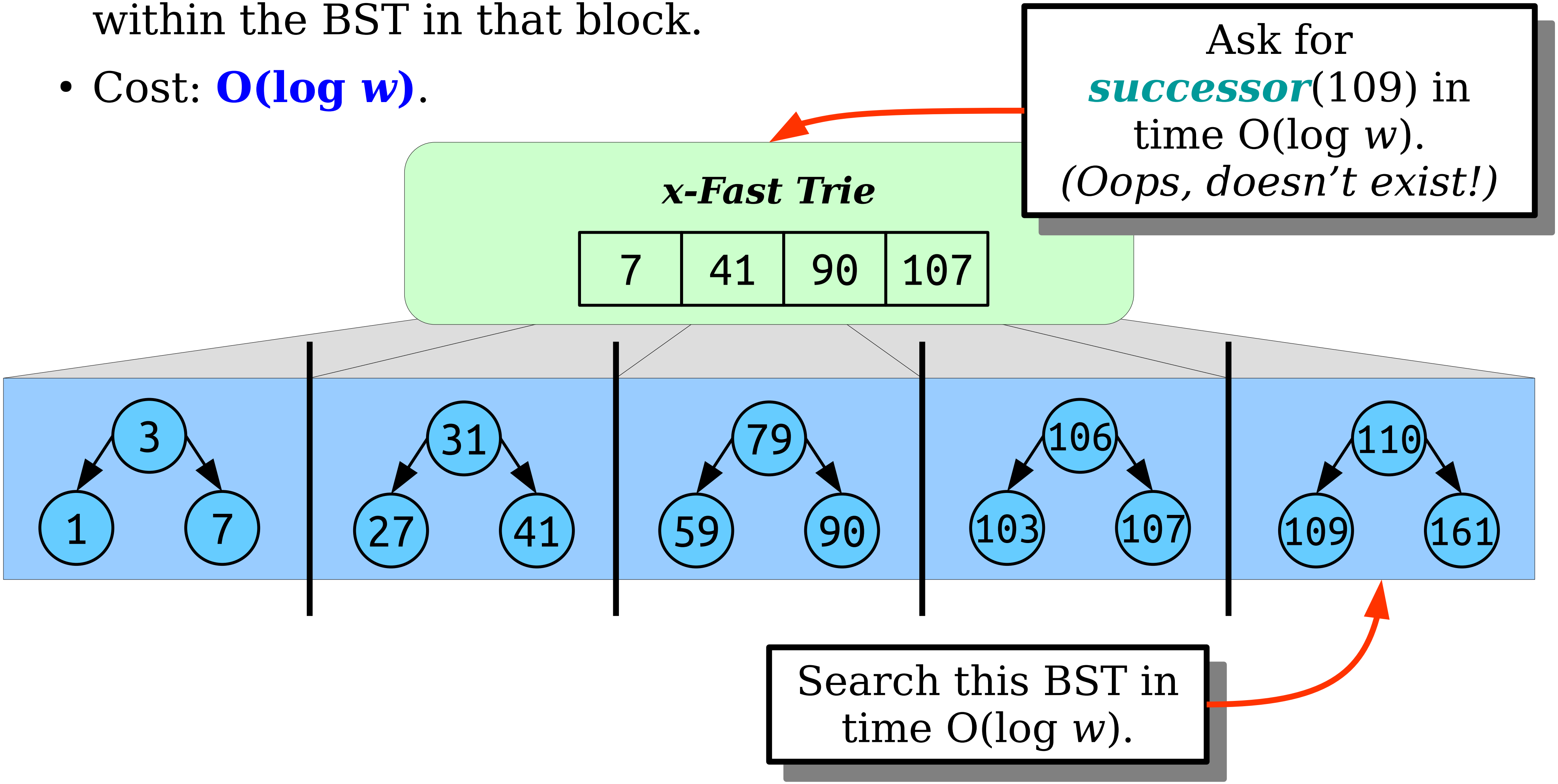
Performing a Lookup

- Suppose we want to perform ***lookup***(90).
- ***Idea:*** figure out which block 90 would belong to, then search within the BST in that block.
- Cost: **O(log *w*)**.



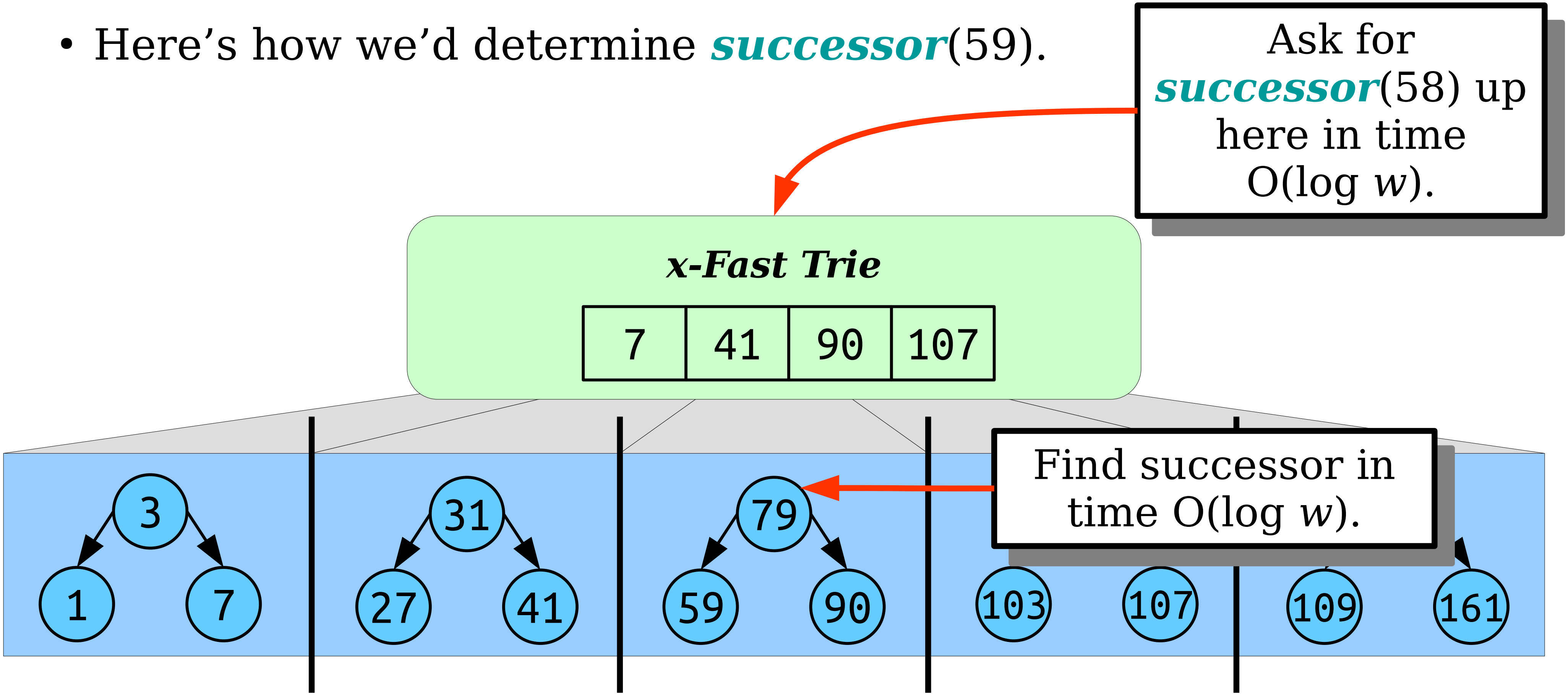
Performing a Lookup

- Suppose we want to perform ***lookup***(110).
- ***Idea:*** figure out which block 109 would belong to, then search within the BST in that block.
- Cost: **O(log *w*)**.



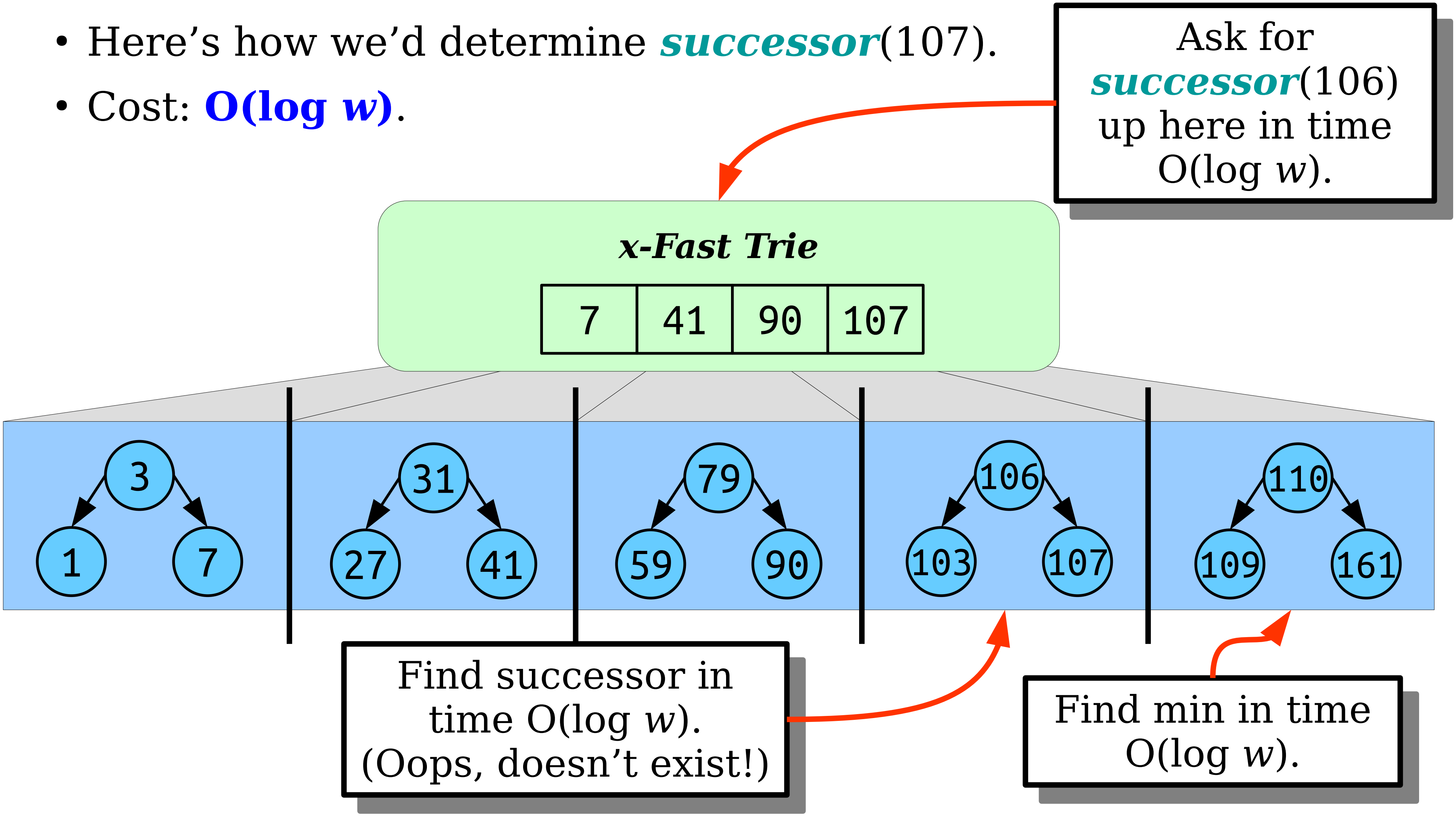
Successor Queries

- How might we perform ***successor*** queries?
- Here's how we'd determine ***successor***(59).



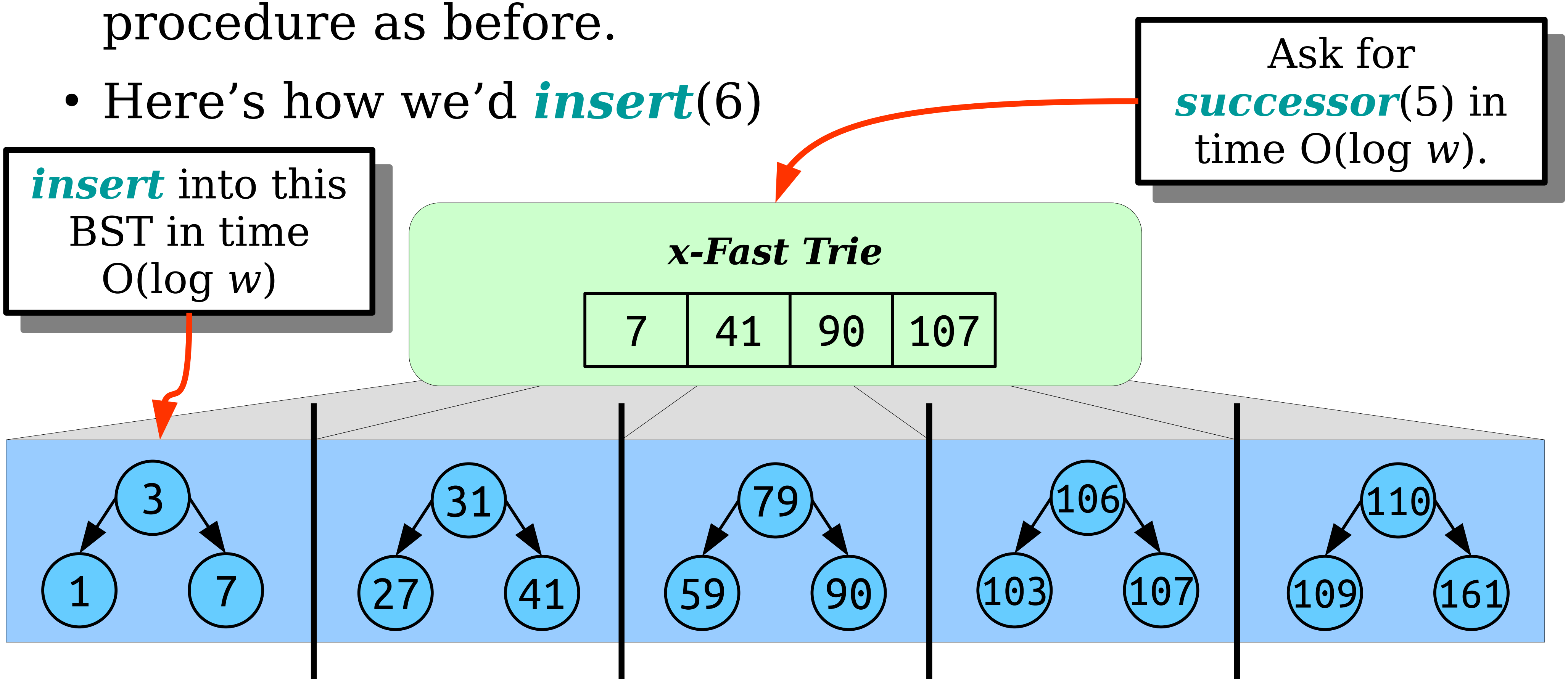
Successor Queries

- How might we perform ***successor*** queries?
- Here's how we'd determine ***successor***(107).
- Cost: **O(log *w*)**.



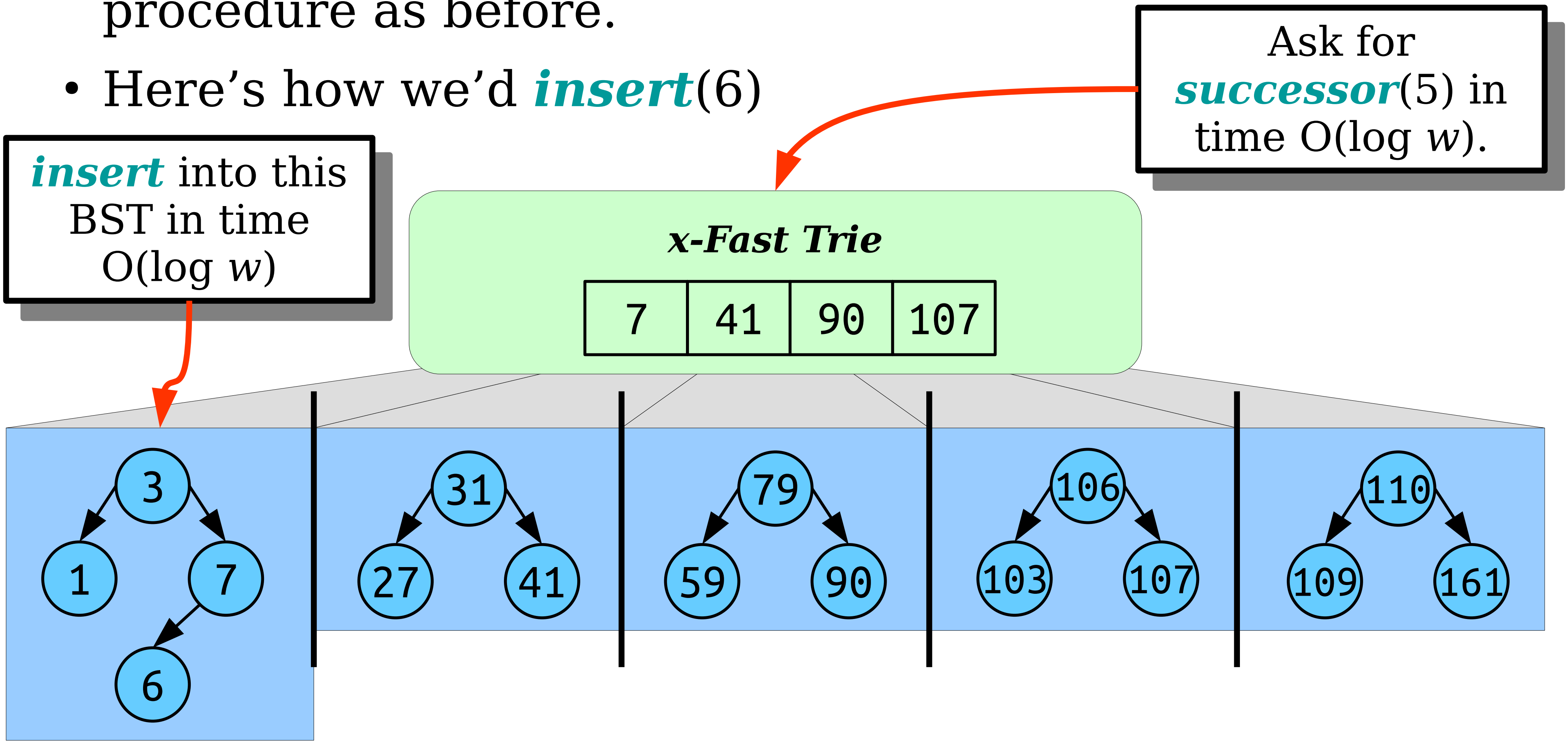
Making Edits

- With a major caveat, insertions follow the same procedure as before.
- Here's how we'd ***insert***(6)



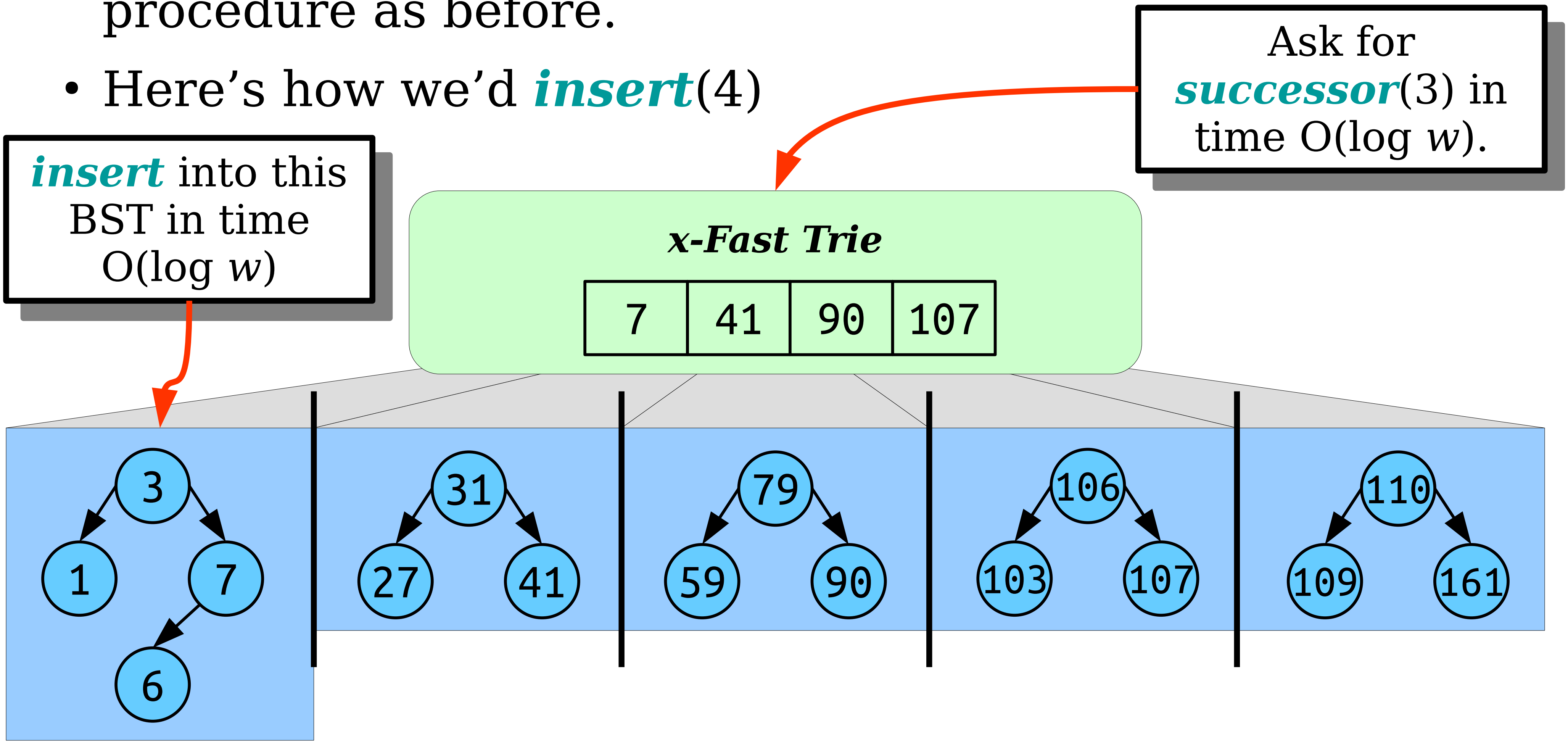
Making Edits

- With a major caveat, insertions follow the same procedure as before.
- Here's how we'd ***insert***(6)



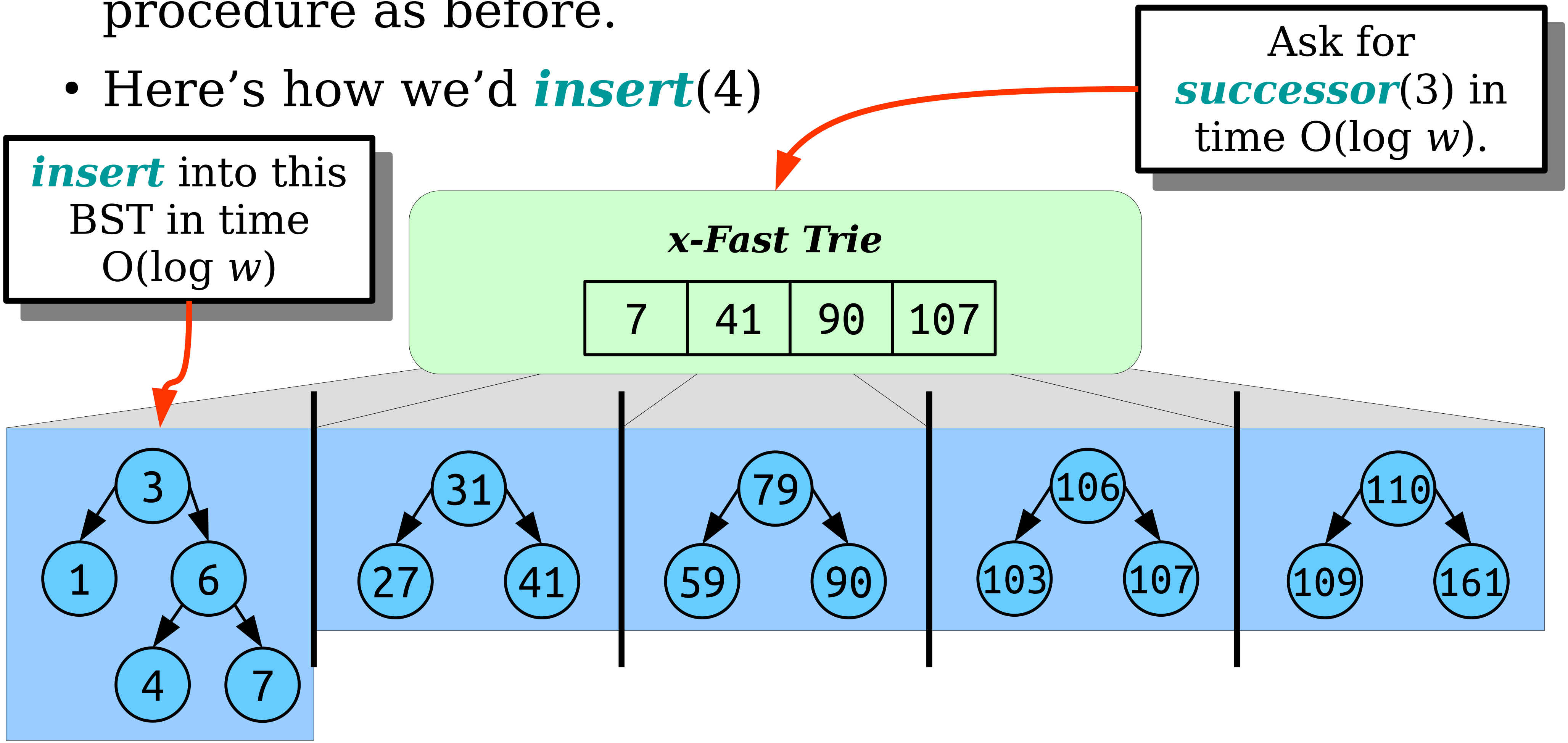
Making Edits

- With a major caveat, insertions follow the same procedure as before.
- Here's how we'd ***insert***(4)



Making Edits

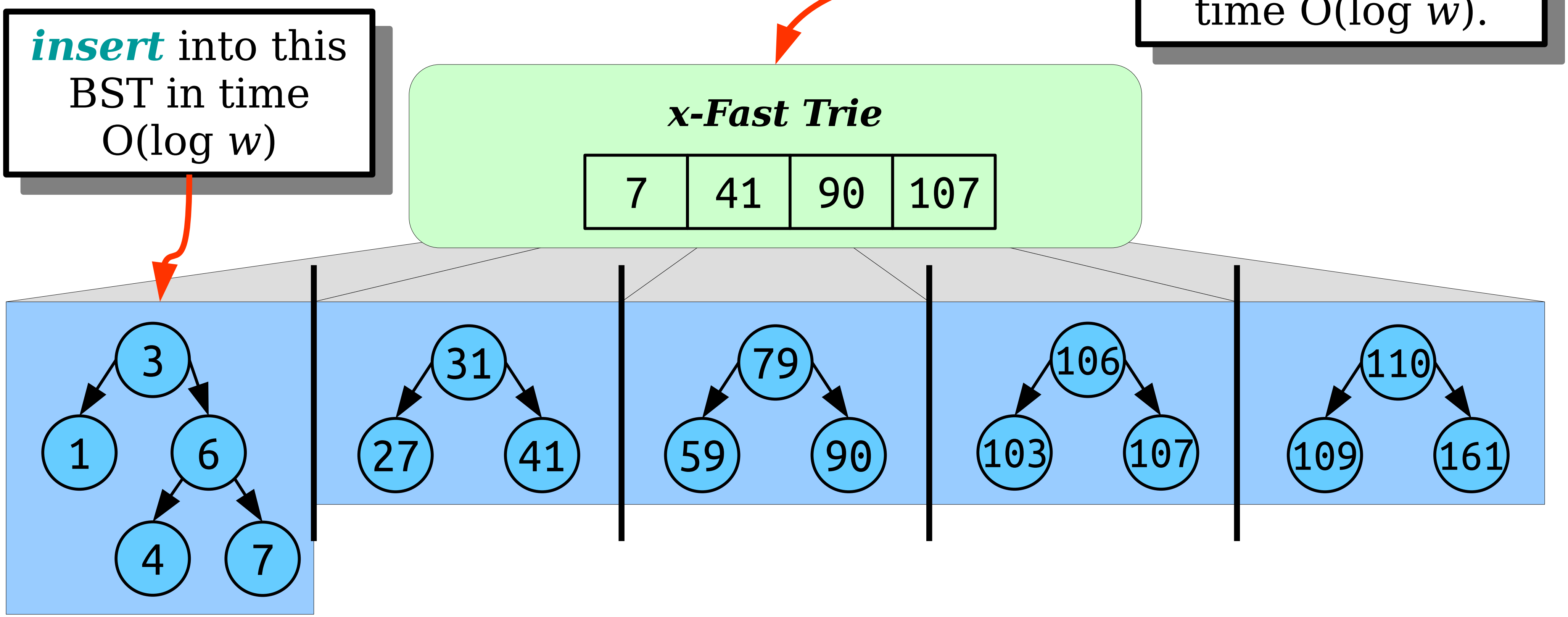
- With a major caveat, insertions follow the same procedure as before.
- Here's how we'd ***insert***(4)



Making Edits

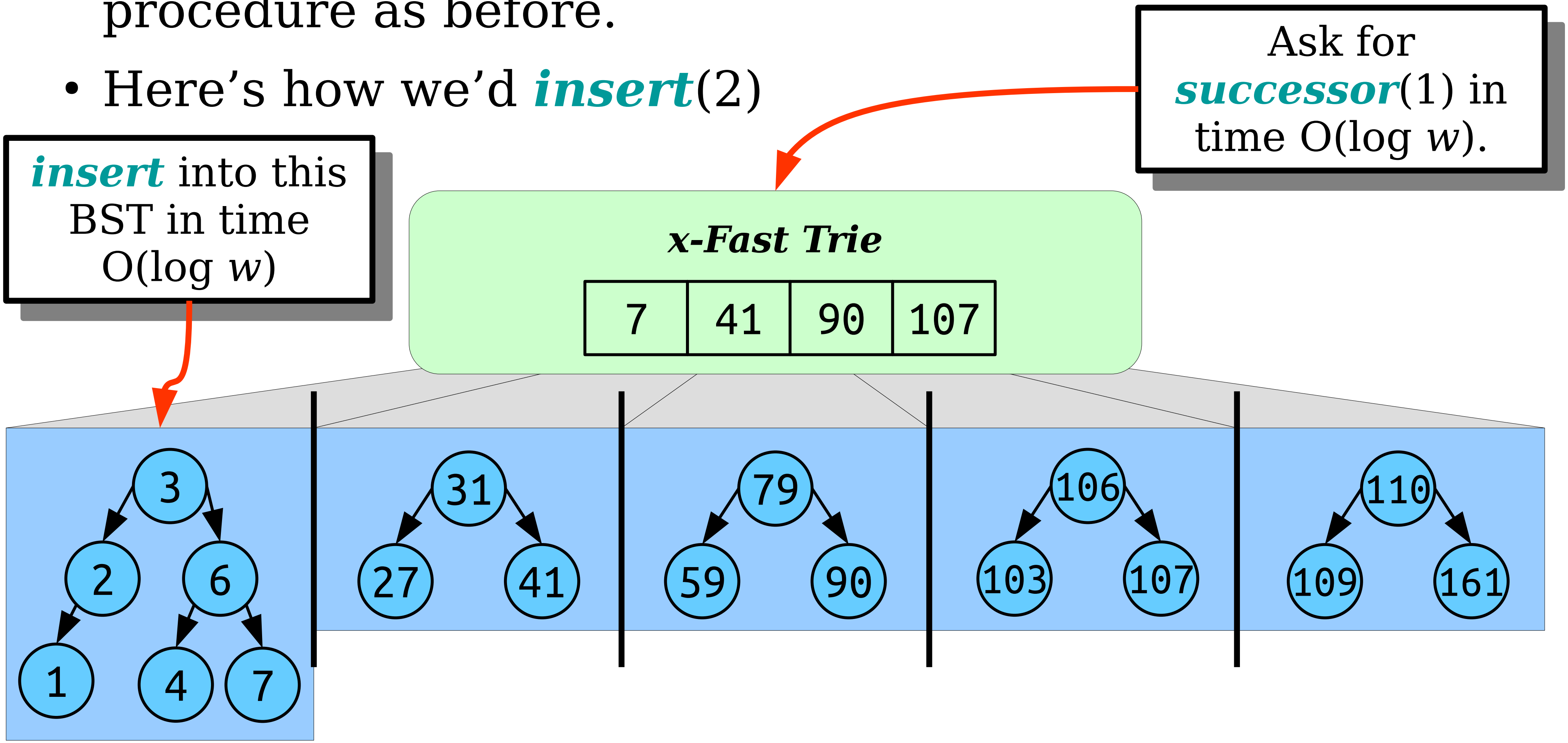
- With a major caveat, insertions follow the same procedure as before.
- Here's how we'd ***insert***(2)

Ask for ***successor***(1) in time O(log *w*).



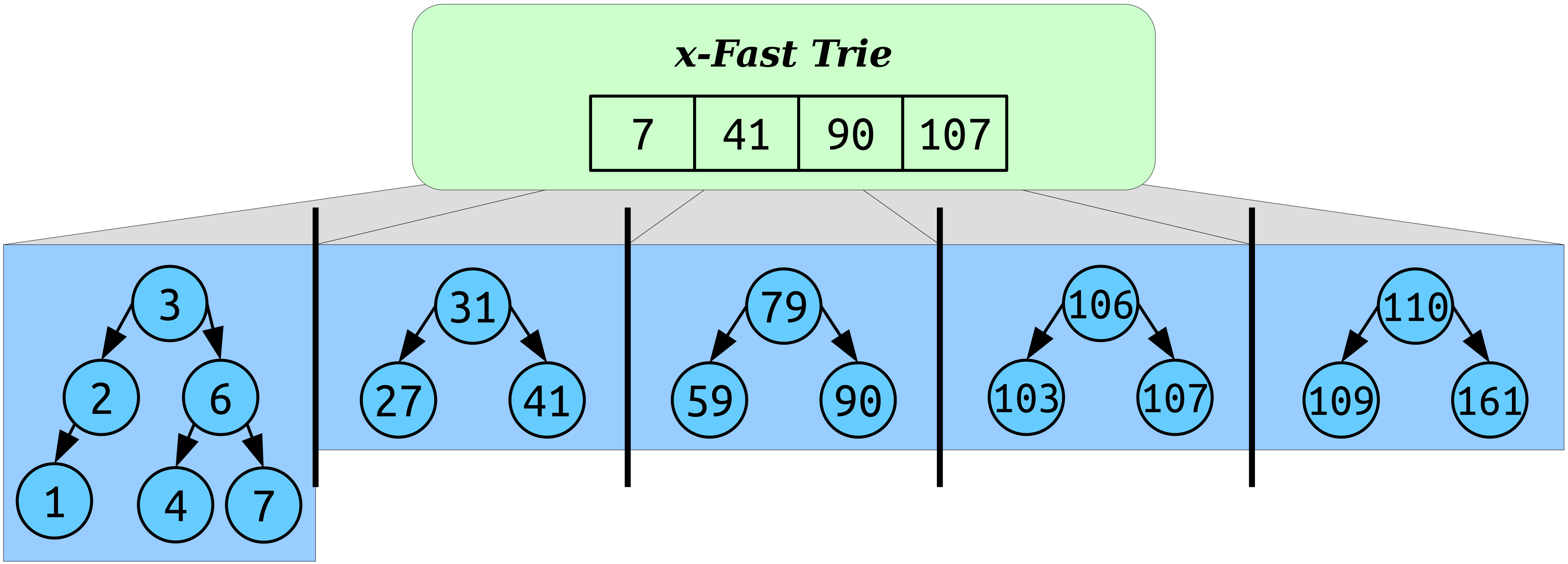
Making Edits

- With a major caveat, insertions follow the same procedure as before.
- Here's how we'd ***insert***(2)



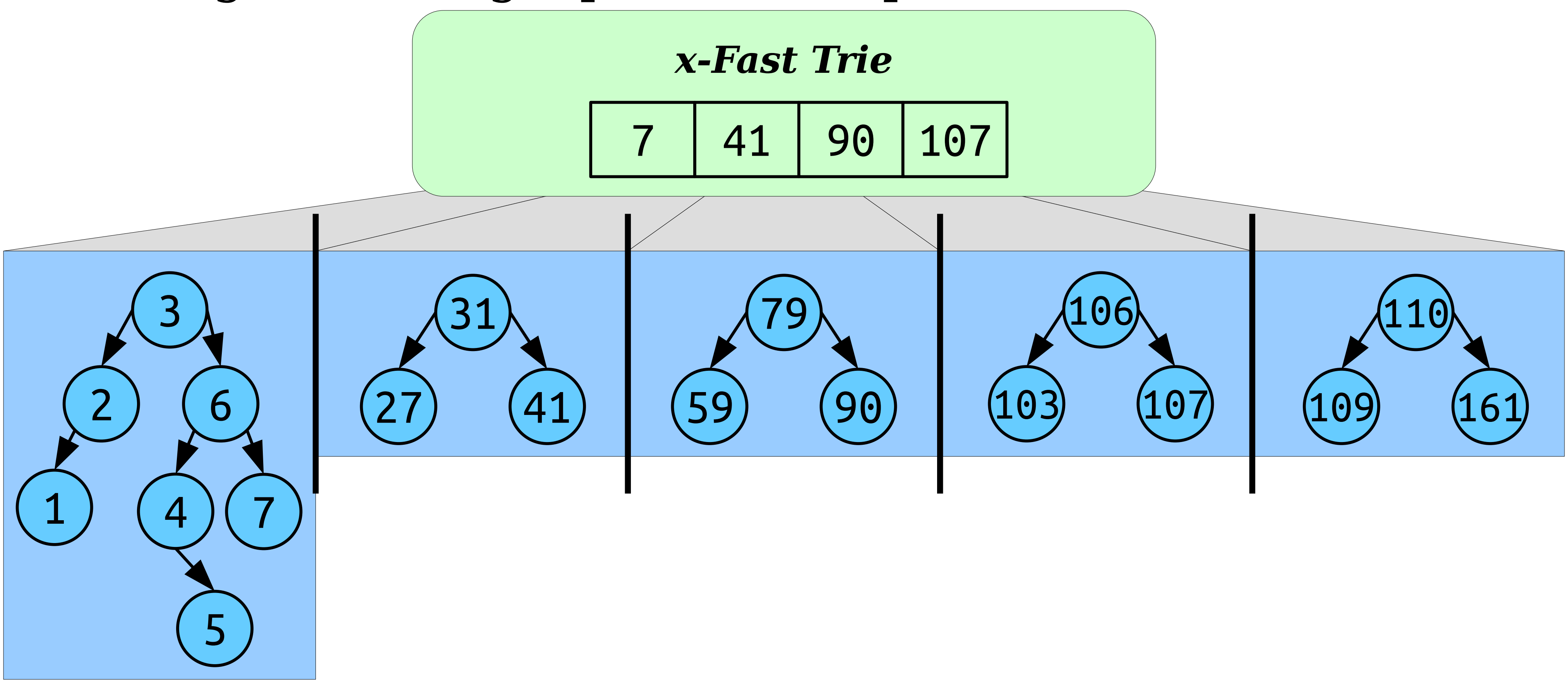
The Problem

- If our trees get too big, we may lose our O(log w) time bound. *(Why?)*
- ***Idea:*** Require each tree to have at most $2w$ elements. If it gets too big, split it and update the x-fast trie.



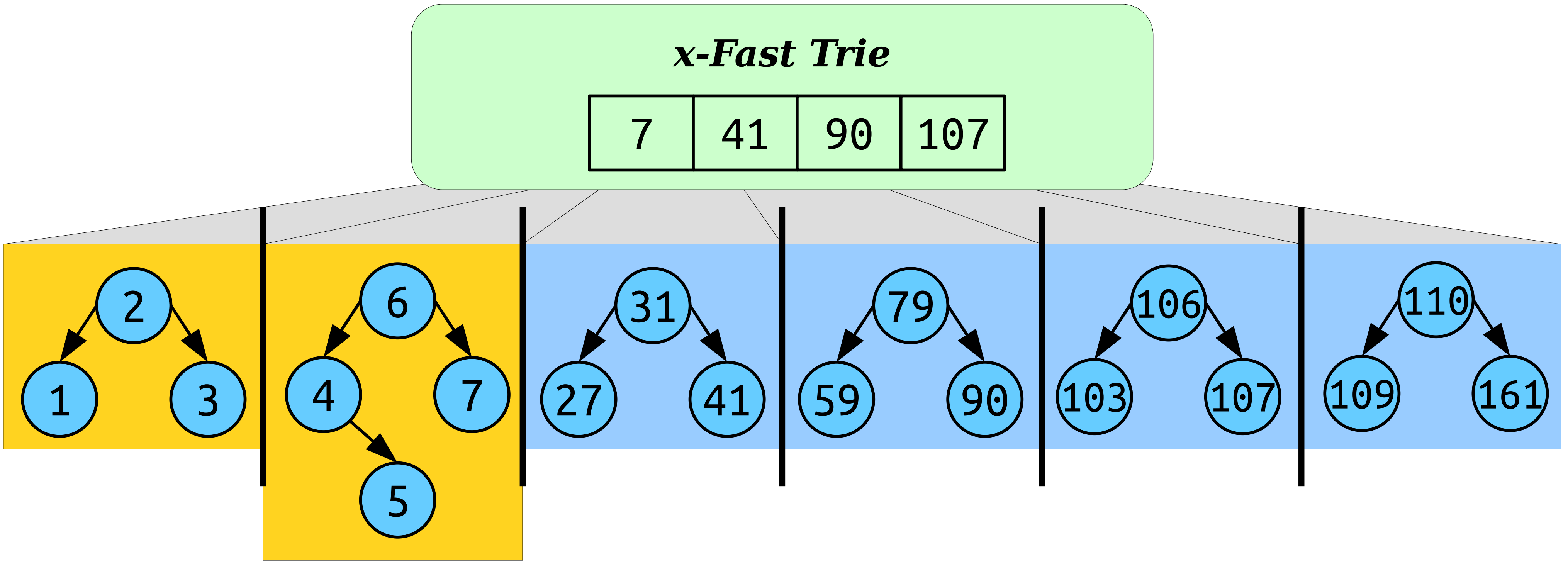
The Problem

- If our trees get too big, we may lose our O(log w) time bound. *(Why?)*
- ***Idea:*** Require each tree to have at most $2w$ elements. If it gets too big, split it and update the x-fast trie.



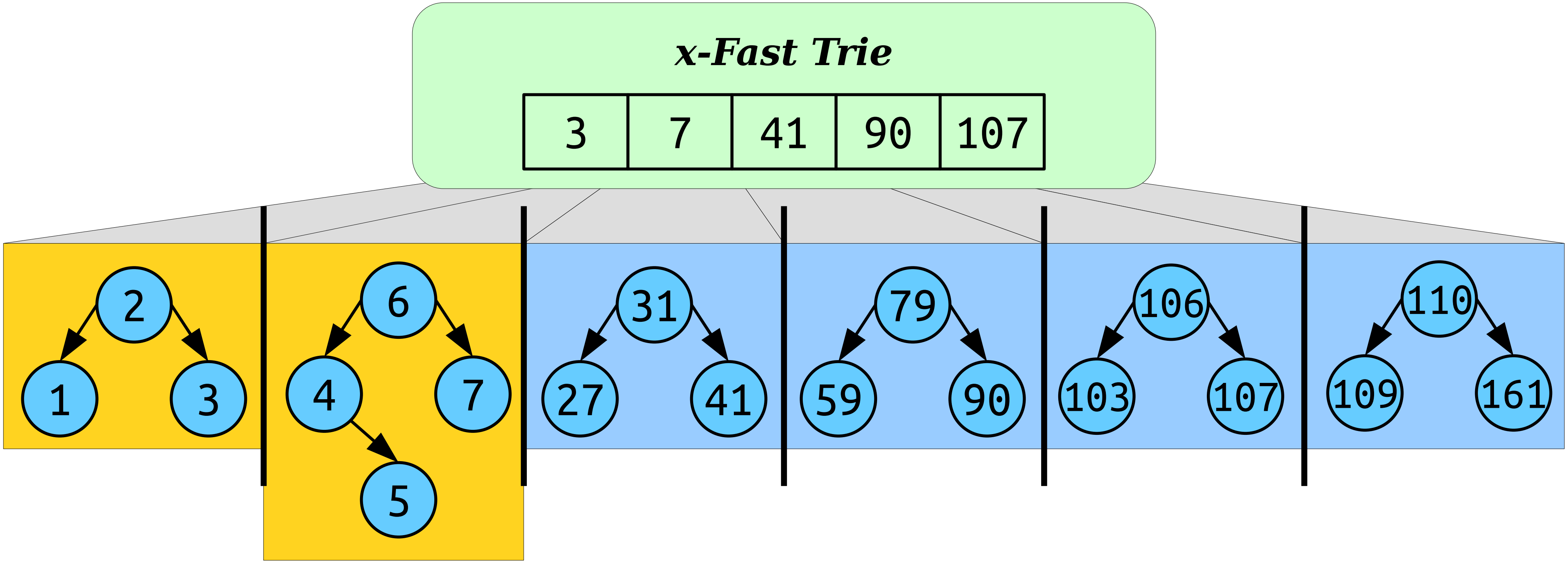
The Problem

- If our trees get too big, we may lose our O(log w) time bound. *(Why?)*
- ***Idea:*** Require each tree to have at most 2w elements. If it gets too big, split it and update the x-fast trie.



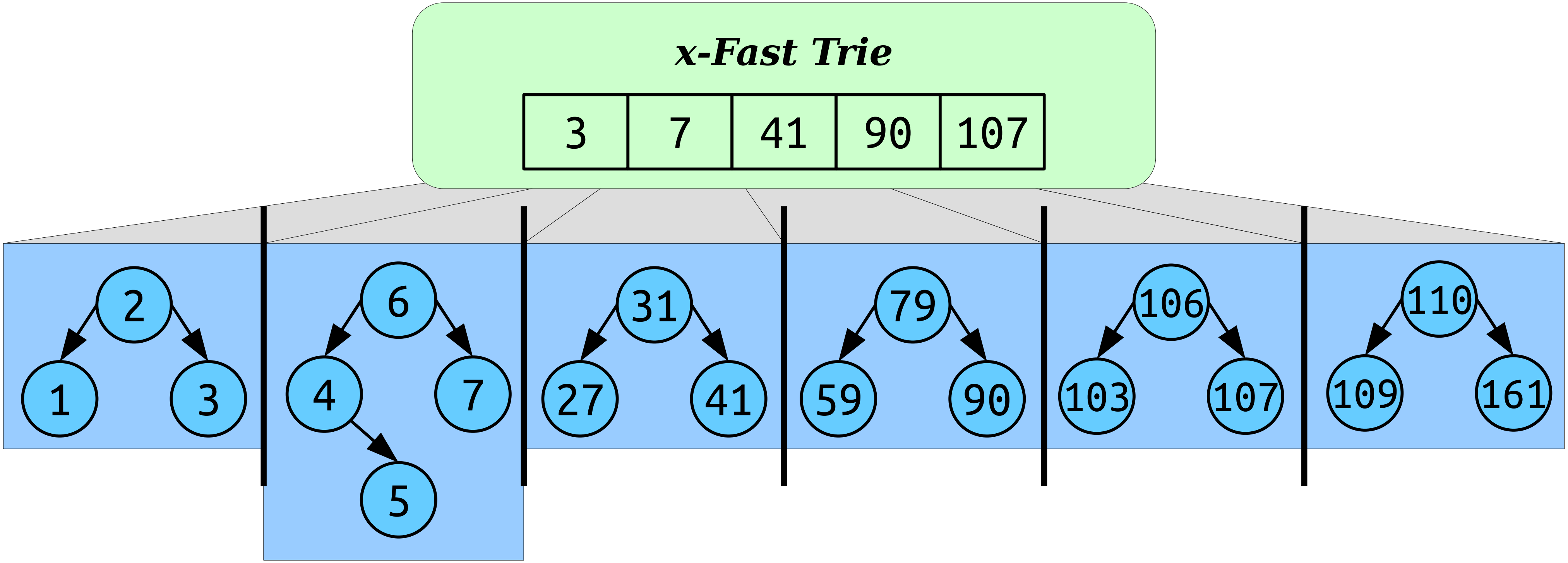
The Problem

- If our trees get too big, we may lose our O(log w) time bound. *(Why?)*
- ***Idea:*** Require each tree to have at most $2w$ elements. If it gets too big, split it and update the x-fast trie.



The Problem

- If our trees get too big, we may lose our O(log w) time bound. *(Why?)*
- ***Idea:*** Require each tree to have at most $2w$ elements. If it gets too big, split it and update the x-fast trie.



Analyzing an Insertion

- If we perform an ***insert*** and don't end up doing a resize, the cost is O(log w).
- If we perform an ***insert*** and *do* have to do a resize, the work done is
  - O(log w) to ***split*** the binary search tree (say, using a splay tree), and
  - O(w) to insert into the x-fast trie.
- Total work: **O(w)**.

Analyzing an Insertion

- If we perform an ***insert*** and don't end up doing a resize, the cost is O(log *w*).
- If we perform an ***insert*** and *do* have to do a resize, the work done is
  - O(log *w*) to ***split*** the binary search tree (say, using a splay tree), and
  - O(*w*) to insert into the *x*-fast trie.
- Total work: **O(*w*)**.

But this is uncommon!
We only do this if a tree
got way too big.

An Amortized Analysis

- Whenever we do an insertion, place a credit on the newly-inserted element.
  - Cost of a "light" ***insert*** still O(log w).
- If we have to split a tree, the tree size was above $2w$, so there must be w credits on it (one for each element above w).
- The ***amortized*** cost of a "heavy" insert is then

$$O(\log w) + O(w) - \Theta(w) = \mathbf{O(\log w)}.$$

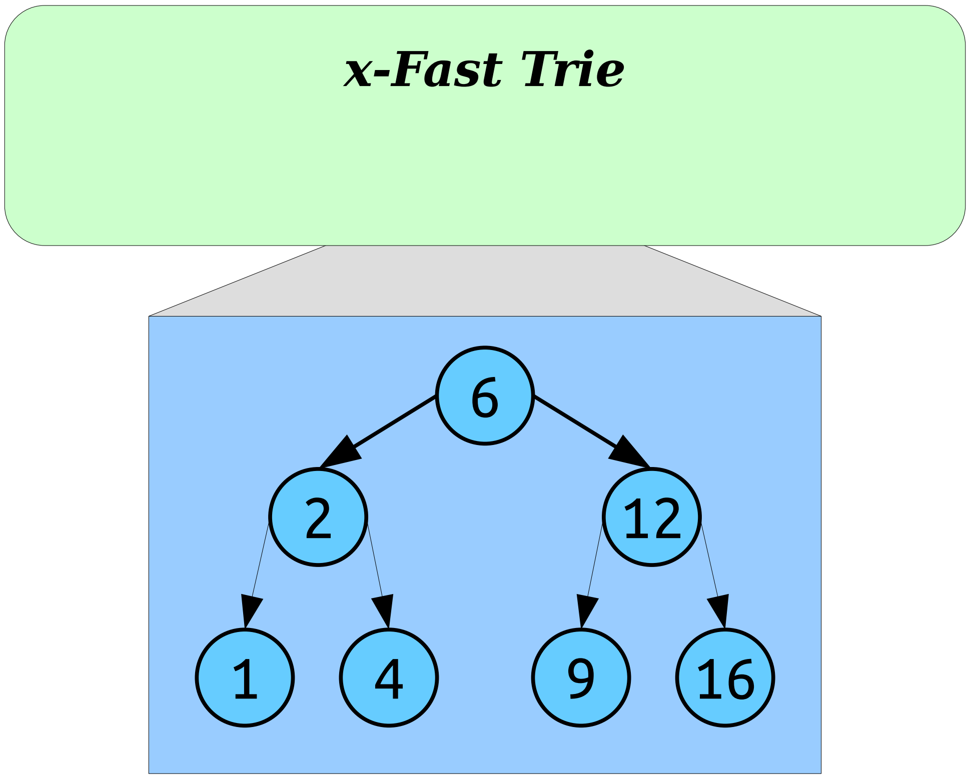
Cost of a regular insert, plus the tree split.

Cost of adding to the x-fast trie.

Credits spent.

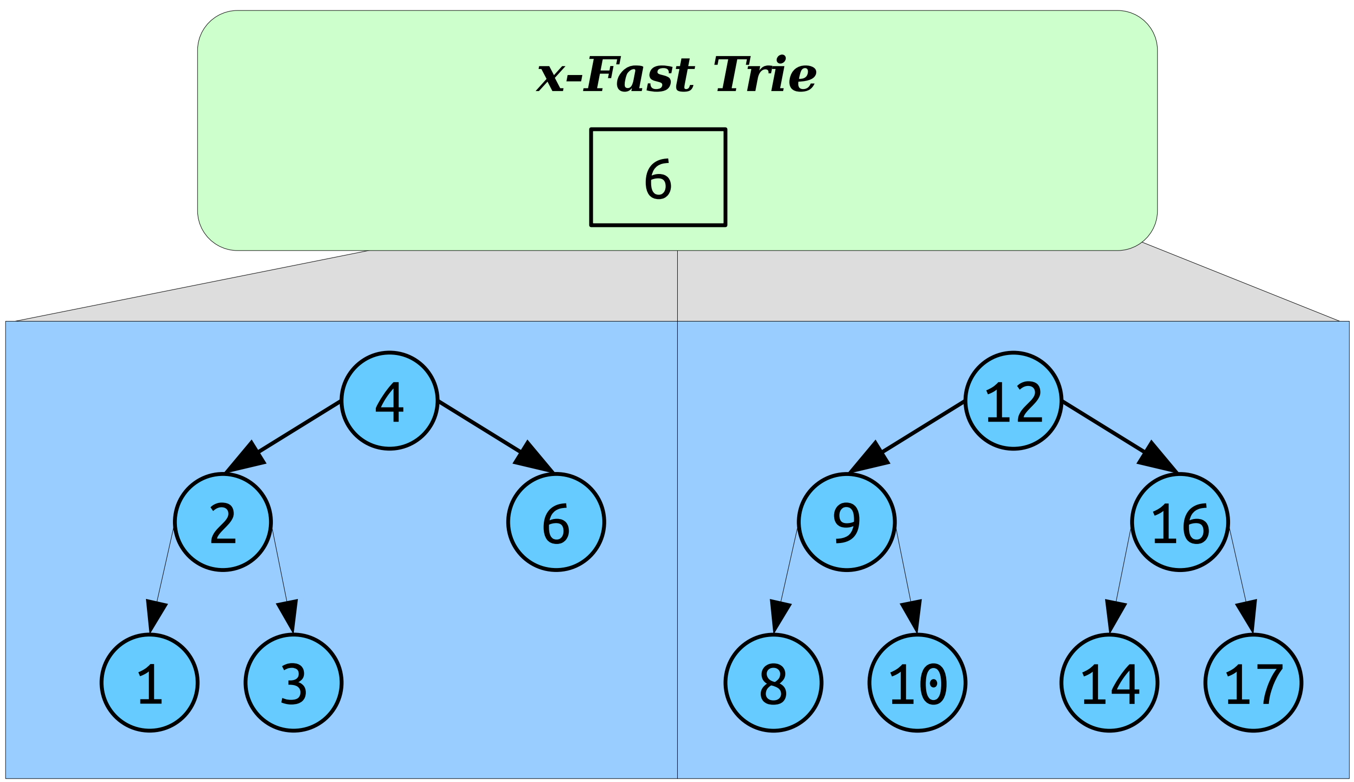
A Nice Side-Effect

- We can now abandon our assumption that we're given all the keys in sorted order in advance.
- Each insertion takes amortized time O(log *w*), so we can build the structure up from scratch!



A Nice Side-Effect

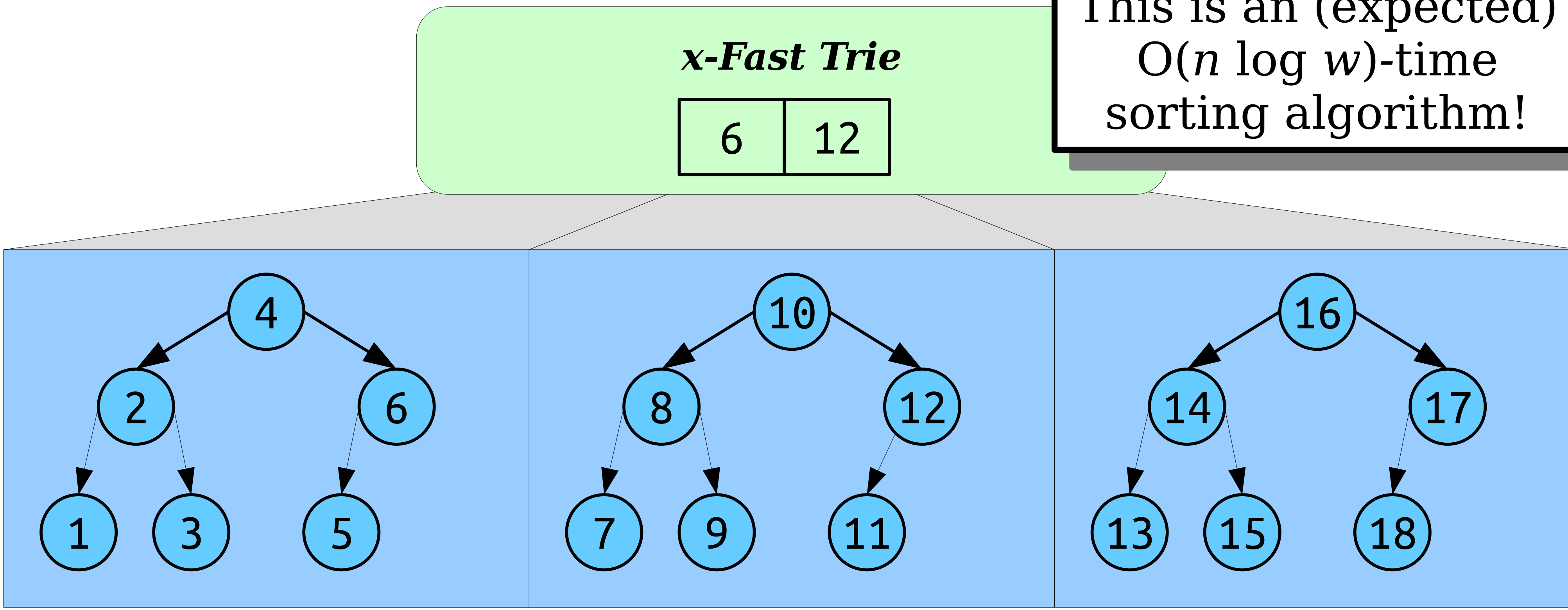
- We can now abandon our assumption that we're given all the keys in sorted order in advance.
- Each insertion takes amortized time O(log *w*), so we can build the structure up from scratch!



A Nice Side-Effect

- We can now abandon our assumption that we're given all the keys in sorted order in advance.
- Each insertion takes amortized time O(log *w*), so we can build the structure up from scratch!

This is an (expected) O(*n* log *w*)-time sorting algorithm!

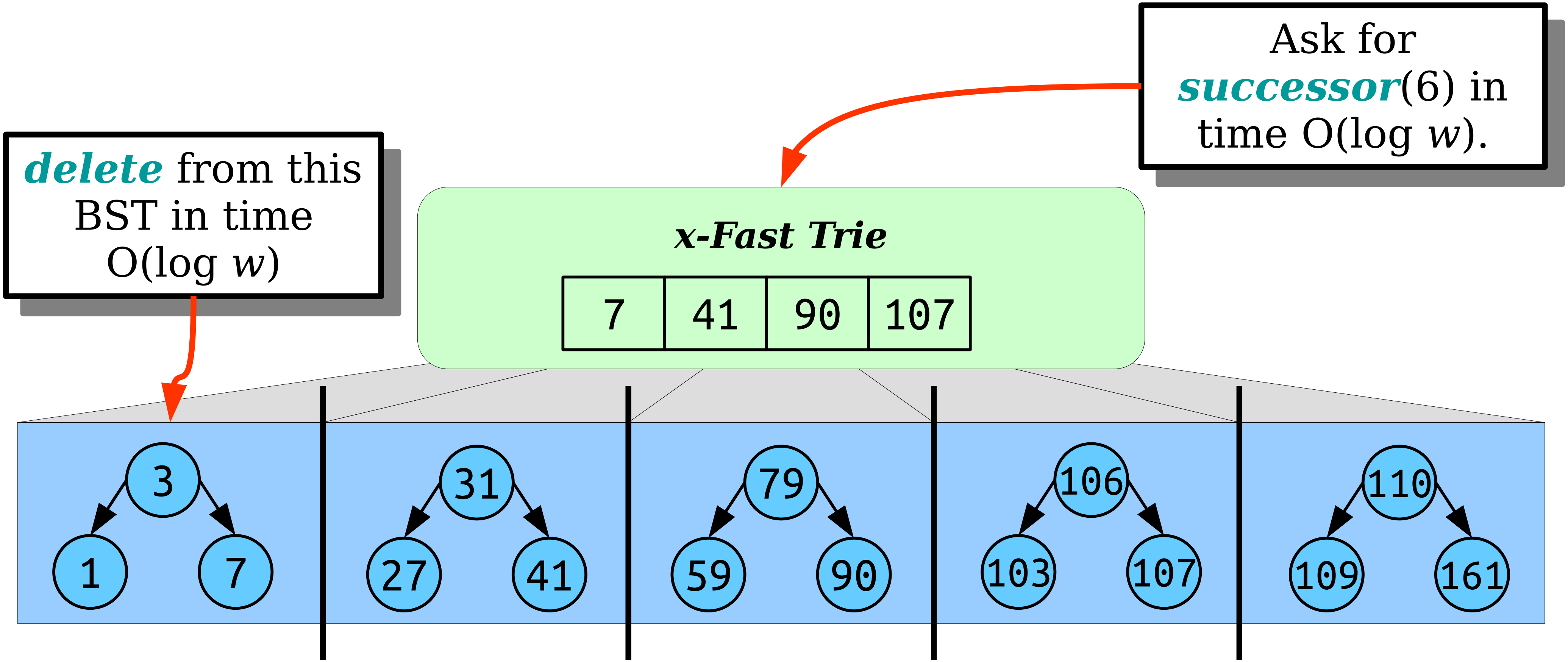


Making Edits

- With a major caveat, deletions follow the same procedure as insertions.
- Here's how we'd ***delete***(7).

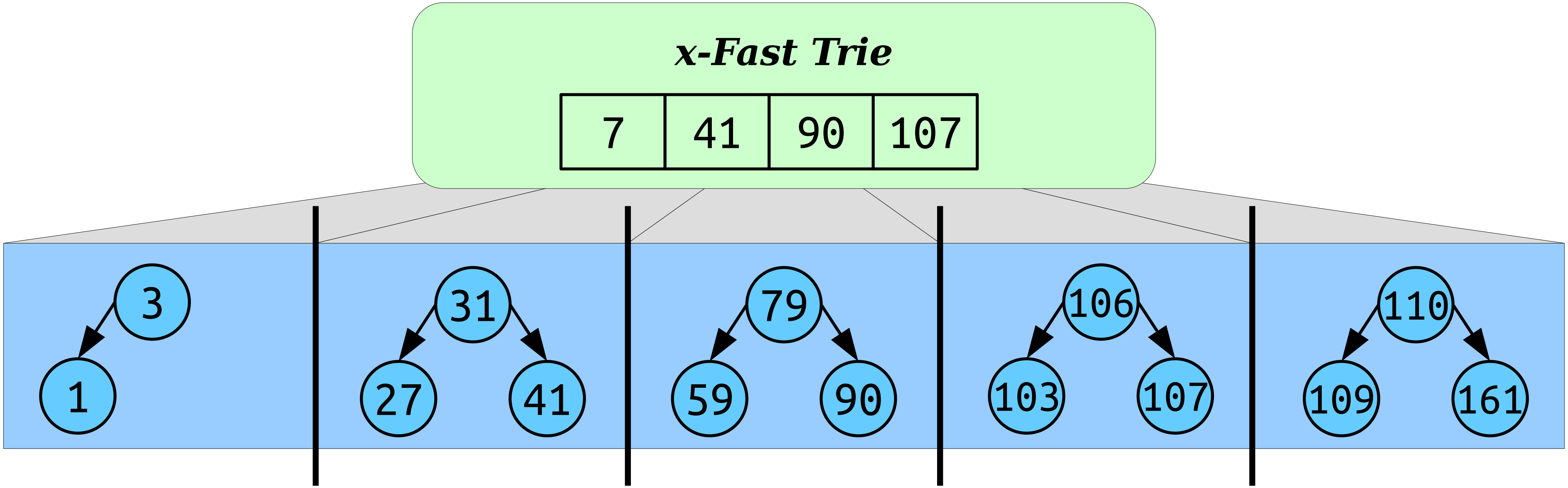
Making Edits

- With a major caveat, deletions follow the same procedure as insertions.
- Here's how we'd ***delete***(7).



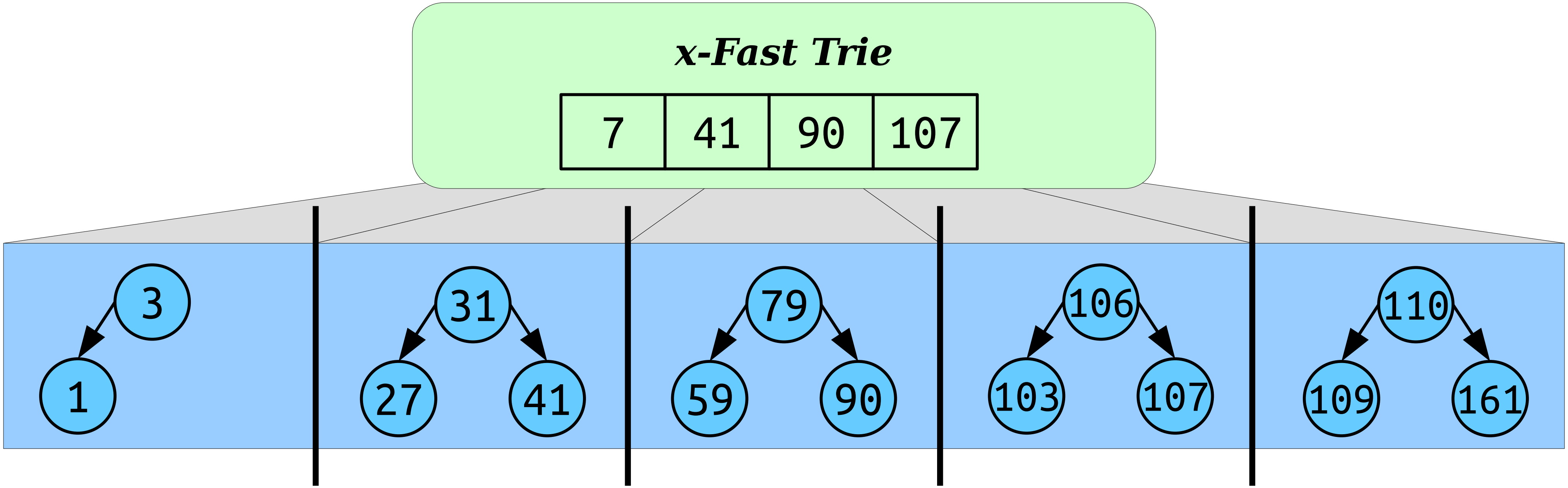
Making Edits

- Our *x*-fast trie still holds 7, even though 7 is no longer present.
- That's not a problem – those keys just serve as "routing information" to tell us which BSTs to look at.
- ***Intuition:*** The *x*-fast trie keys act as partitions between BSTs. They don't need to actually be present in our data structure.



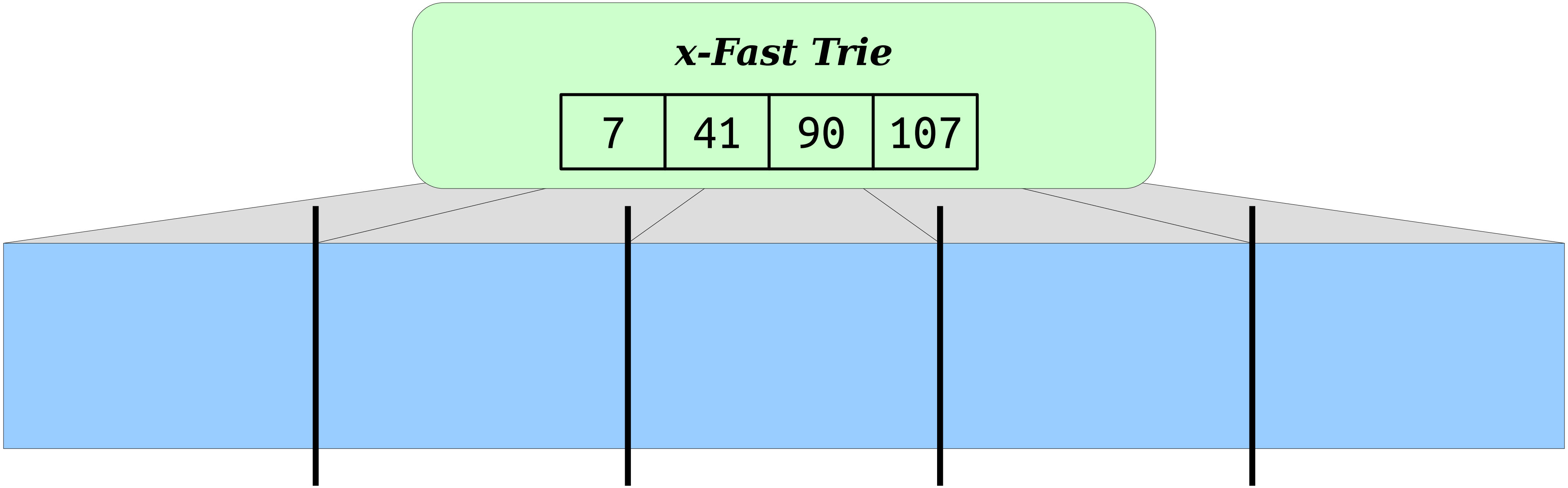
Shrinking our Structure

- What happens if we remove all the elements from our structure without touching the *x*-fast trie?



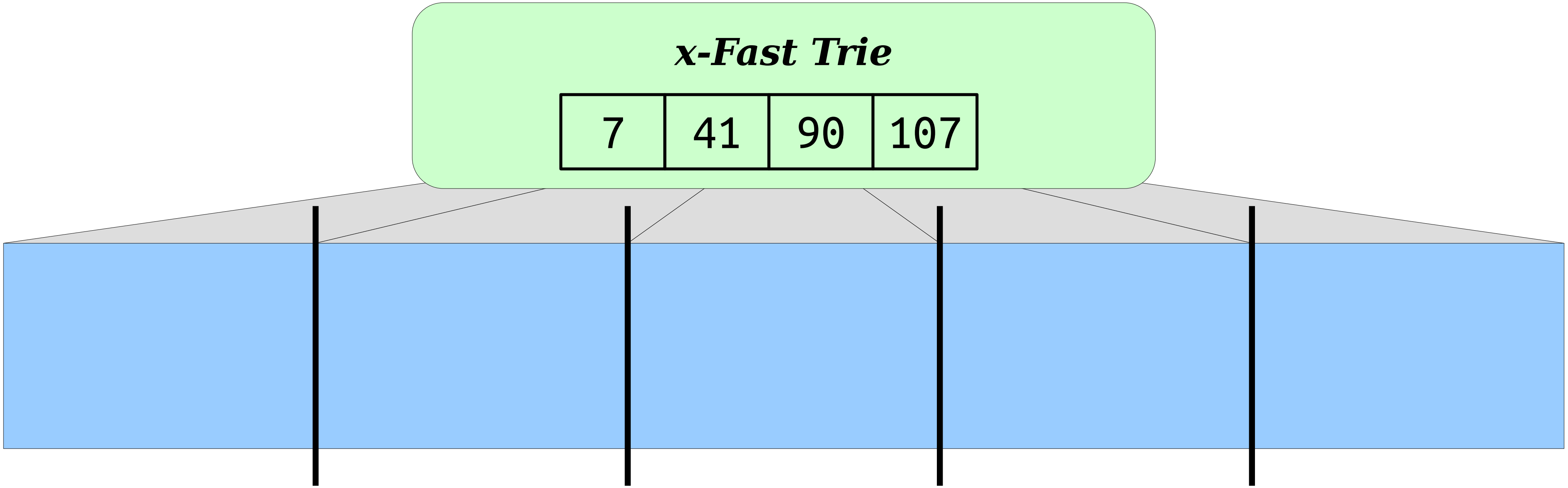
Shrinking our Structure

- What happens if we remove all the elements from our structure without touching the *x*-fast trie?



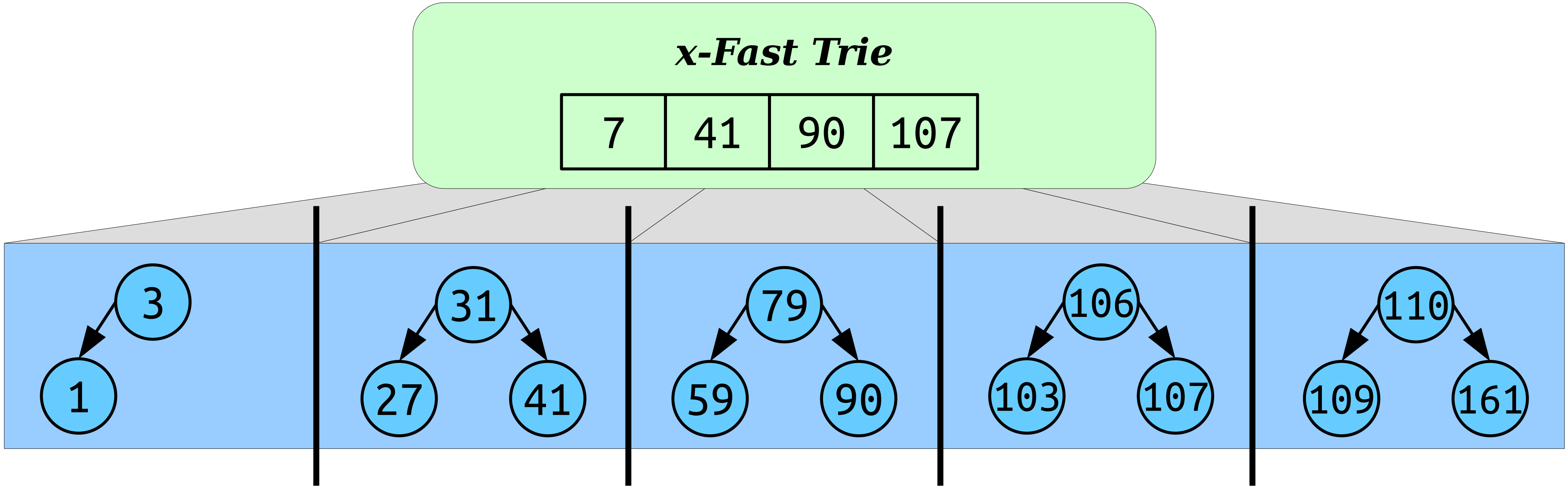
Shrinking our Structure

- What happens if we remove all the elements from our structure without touching the *x*-fast trie?
- Each operation still takes time **O(log *w*)**.
- But now our space usage depends on the maximum size we reached, not the current size!



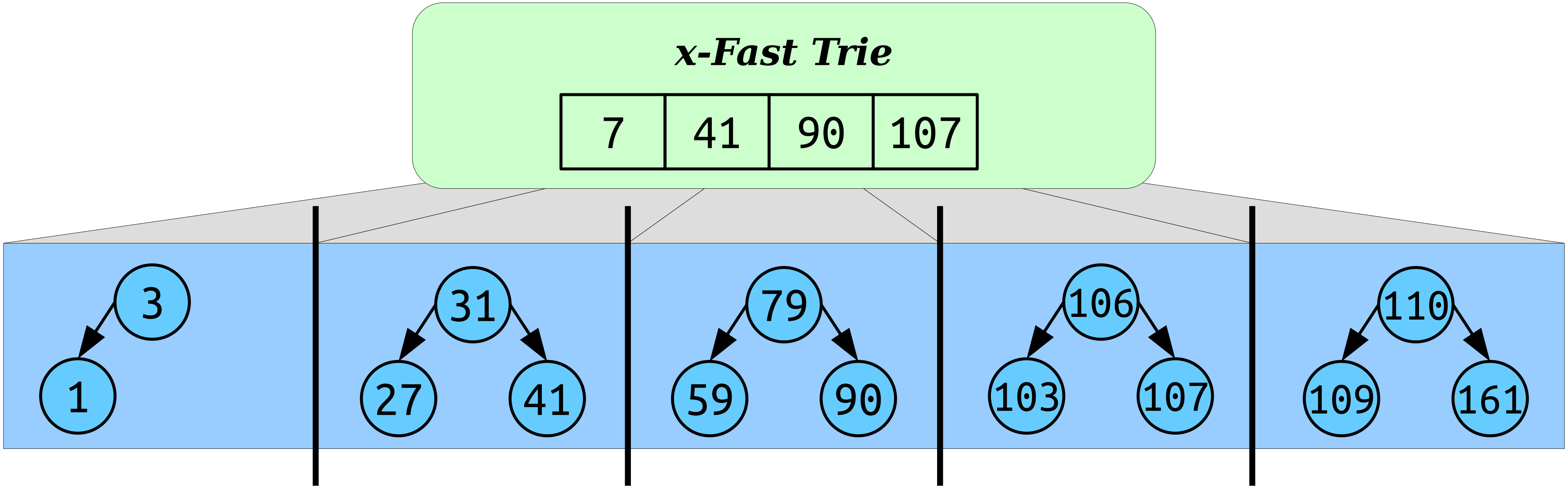
Achieving a Balance

- ***Invariant:*** Require each tree to have between ½w and 2w elements.
- If a tree gets too small:
  - Merge with the tree next to you, editing the x-fast trie as appropriate.
  - If the resulting tree is too big, split it in half, editing the x-fast trie.
- This does O(w) on the x-fast trie only once every Θ(w) operations, so this amortizes out to O(log w) work per operation.



Achieving a Balance

- If each tree has $\Theta(w)$ elements in it, then our space usage is
  - $\Theta(n)$ for all the trees, plus
  - $\Theta((n / w)\ w) = \Theta(n)$ for the *x*-fast trie,
- This uses **$\Theta(n)$** total memory.



What We've Seen

- Here's the final scorecard for the y-fast trie.
- Assuming $n = \omega(w)$, which it probably is, this is faster than a binary search tree!
- And it gives rise to an $O(n \log w)$-expected-time sorting algorithm!

The y-Fast Trie:

- ***lookup***: $O(\log w)$
- ***insert***: $O(\log w)^*$
- ***delete***: $O(\log w)^*$
- ***max***: $O(\log w)$
- ***succ***: $O(\log w)$
- Space: $\Theta(n)$

* Expected, amortized.

What We Needed

- An *x*-fast trie requires ***tries*** and ***cuckoo hashing***.
- The *y*-fast trie requires ***amortized analysis*** and ***split/join*** on ***balanced BSTs***.
- *y*-fast tries also use the “blocking” technique from ***RMQ*** we used to shave off log factors.

What's Missing

- There's still a little gap between where BSTs dominate and where y-fast tries take over.
  - Specifically, what if $n = \Theta(w)$?
- Our solution still involves randomness.
  - We need that in the cuckoo hash tables at each level.
- ***Question:*** Can we build a solution with neither of these weaknesses?

Next Time

- ***Word-Level Parallelism***
  - Treating arithmetic as parallel computation.
- ***Sardine Trees***
  - A fast ordered dictionary for truly tiny integers.
- ***Finding the Most Significant Bit***
  - An astonishing algorithm for a deceptively tricky problem.