x-Fast and y-Fast Tries

Outline for Today

- ***Bitwise Tries***
  - A simple ordered dictionary for integers.
- ***x-Fast Tries***
  - Tries + Hashing
- ***y-Fast Tries***
  - Tries + Hashing + Subdivision + Balanced Trees + Amortization

Recap from Last Time

Ordered Dictionaries

- An ***ordered dictionary*** is a data structure that maintains a set S of elements drawn from an ordered universe $\mathscr{U}$ and supports these operations:
  - ***insert***(x), which adds x to S.
  - ***is-empty***(), which returns whether $S = \emptyset$.
  - ***lookup***(x), which returns whether $x \in S$.
  - ***delete***(x), which removes x from S.
  - ***max***() / ***min***(), which returns the maximum or minimum 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.

Integer Ordered Dictionaries

- Suppose that $\mathscr{U} = [U] = \{0, 1, \ldots, U - 1\}$.
- A ***van Emde Boas tree*** is an ordered dictionary for $[U]$ where
  - ***min***, ***max***, and ***is-empty*** run in time O(1).
  - All other operations run in time $O(\log \log U)$.
  - Space usage is $\Theta(U)$ if implemented deterministically, and $O(n)$ if implemented using hash tables.
- ***Question:*** Is there a simpler data structure meeting these bounds?

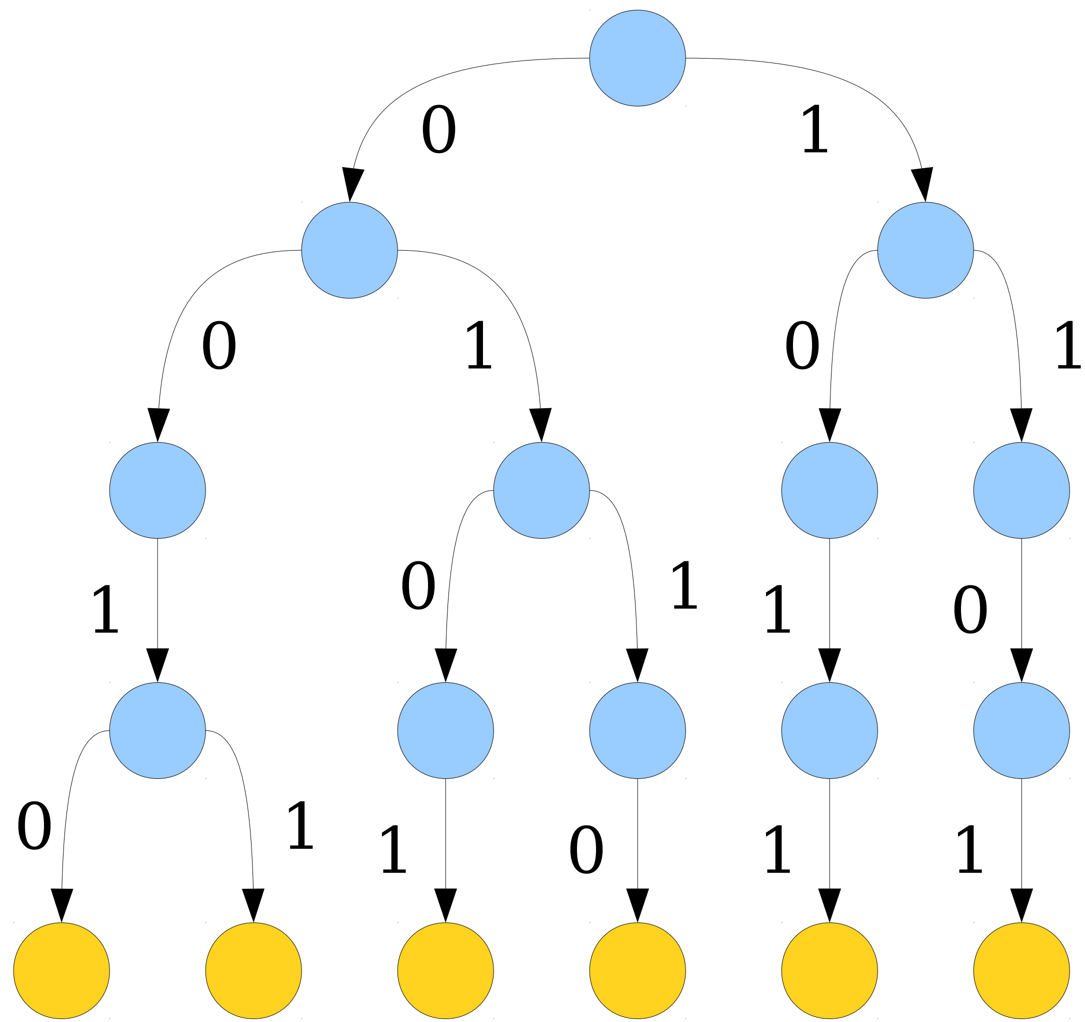
The Machine Model

- We assume a ***transdichotomous machine model***:
  - Memory is composed of words of w bits each.
  - Basic arithmetic and bitwise operations on words take time O(1) each.
  - $w = \Omega(\log n)$.

A Start: ***Bitwise Tries***

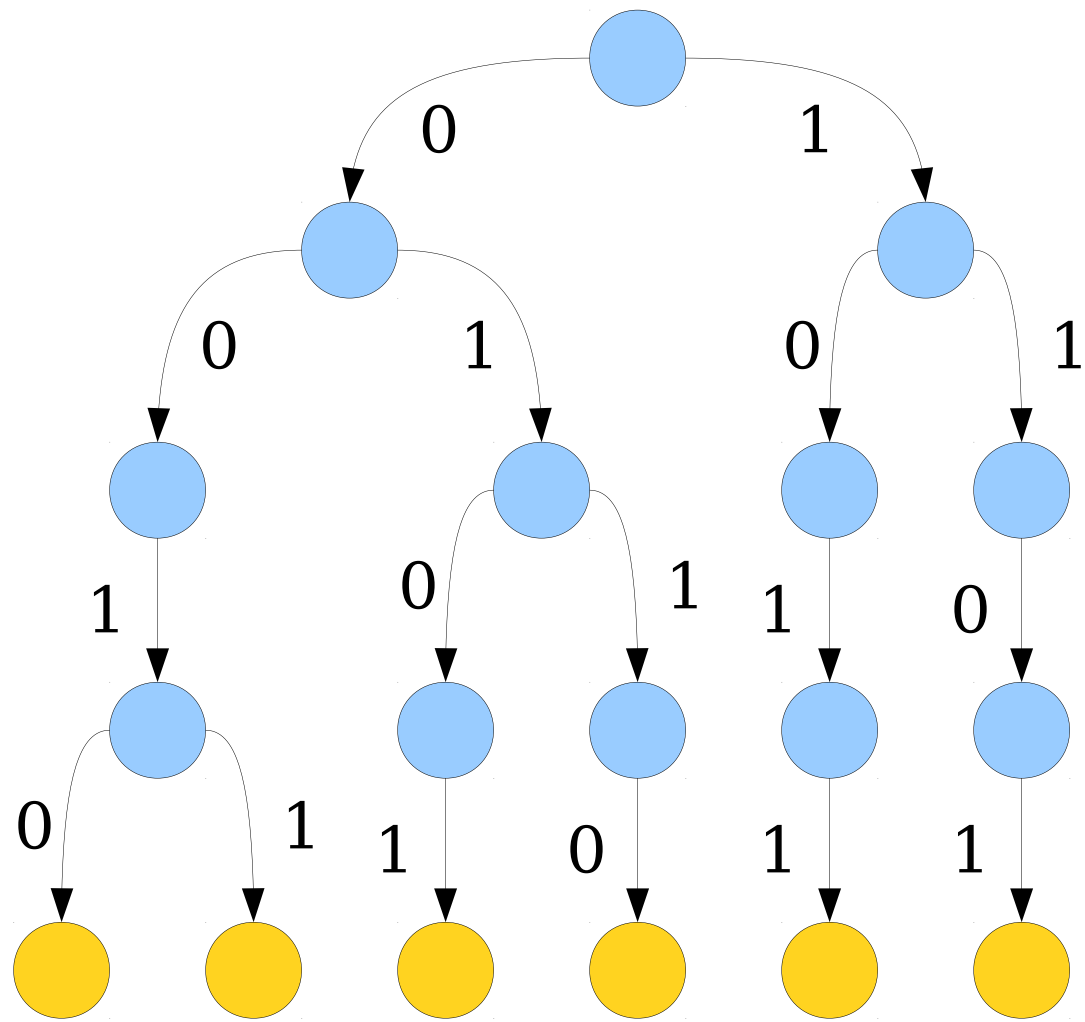
Tries Revisited

- ***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.

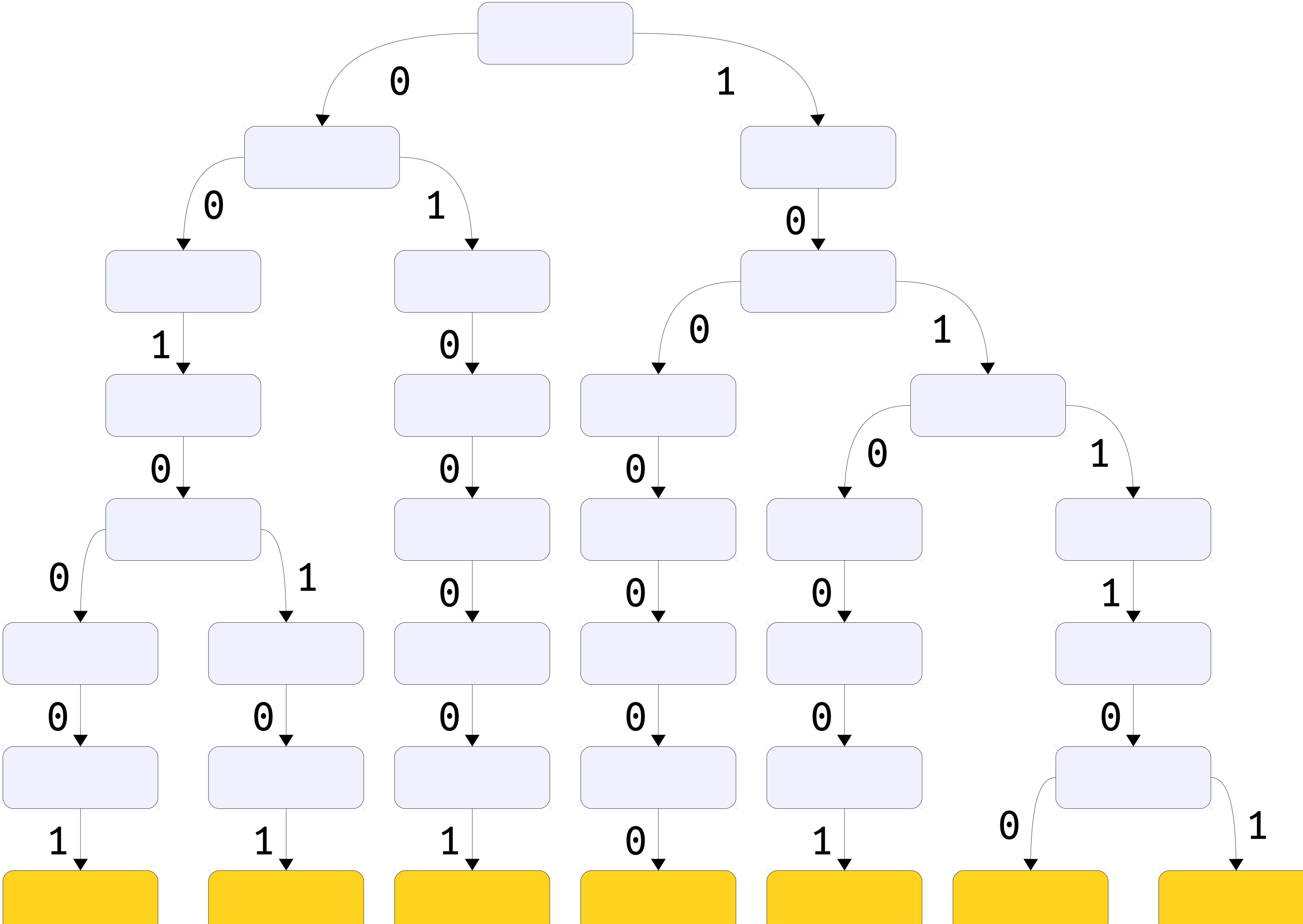


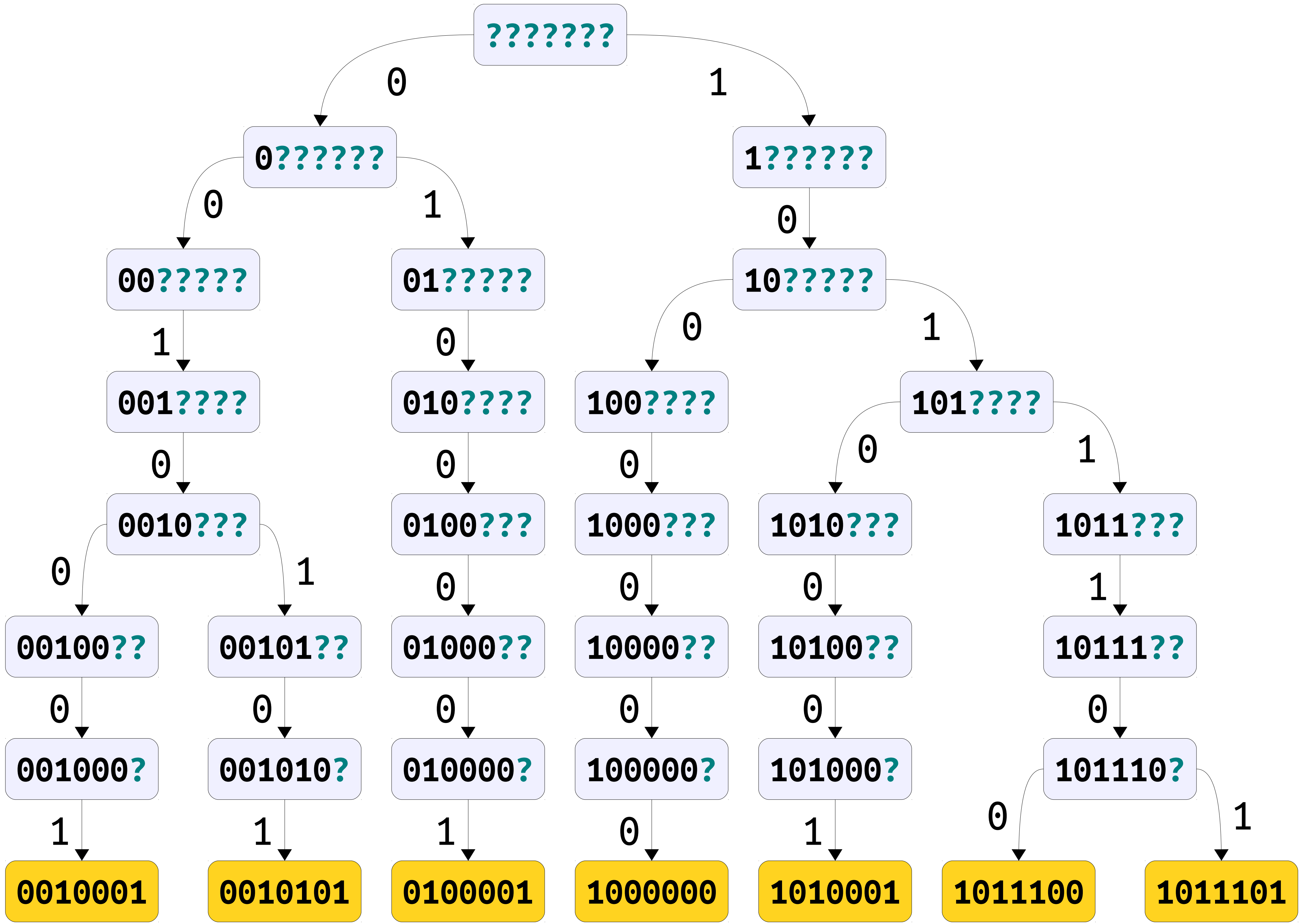
Bitwise Tries

- When storing integers in $[U]$, each integer will have $\Theta(\log U)$ bits.
- Time for any of the ordered dictionary operations: $\mathbf{O(\log \mathit{U})}$.
- In order to match the time bounds of a van Emde Boas tree, we will need to speed this up exponentially.

Speeding up Successors

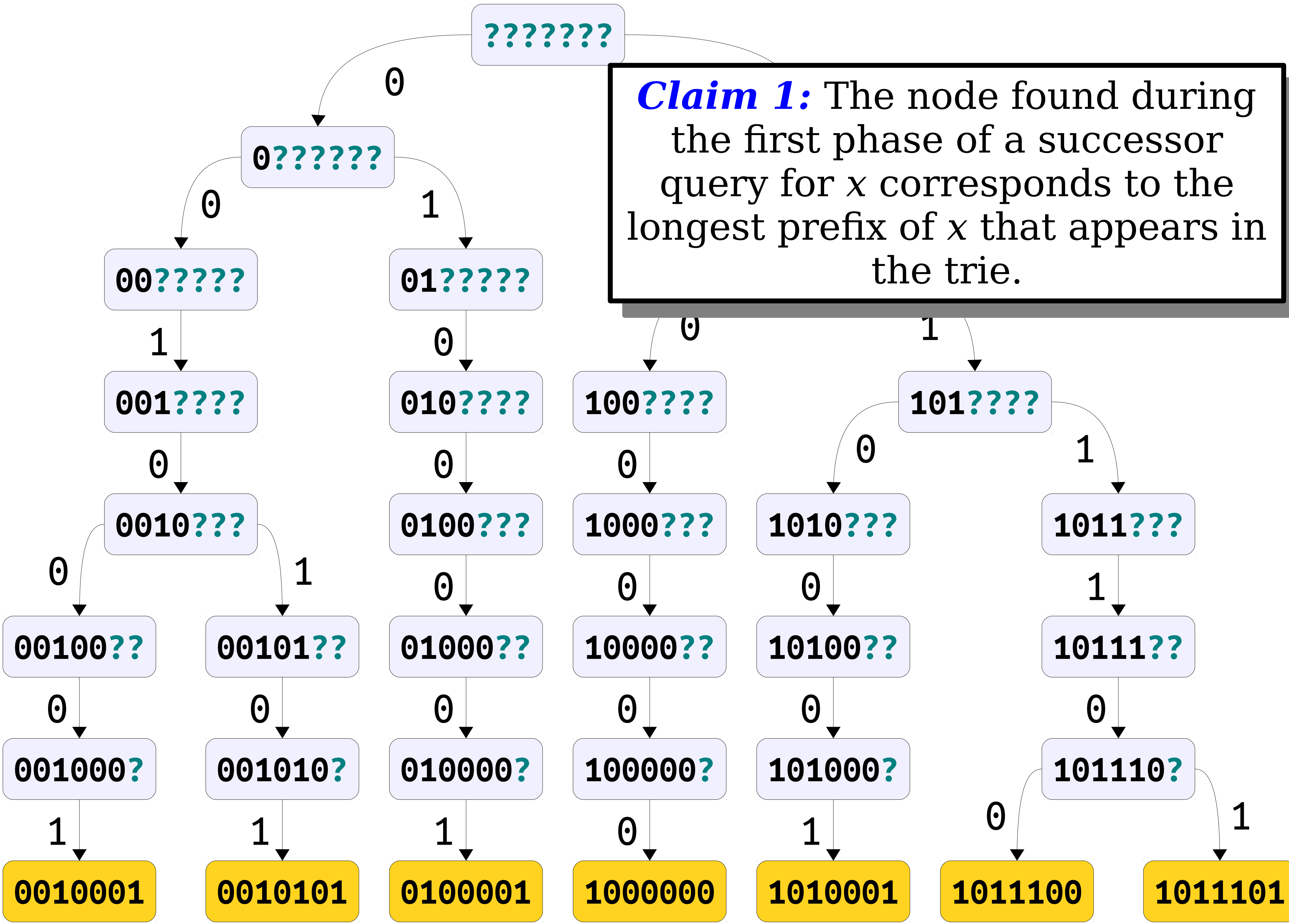
- There are two independent pieces that contribute to the O(log U) runtime:
  - Need to search for the deepest node matching x that we can.
  - From there, need to back up to node with an unfollowed 1 child and then descend to the next leaf.
- To speed this up to O(log log U), we'll need to work around each of these issues.

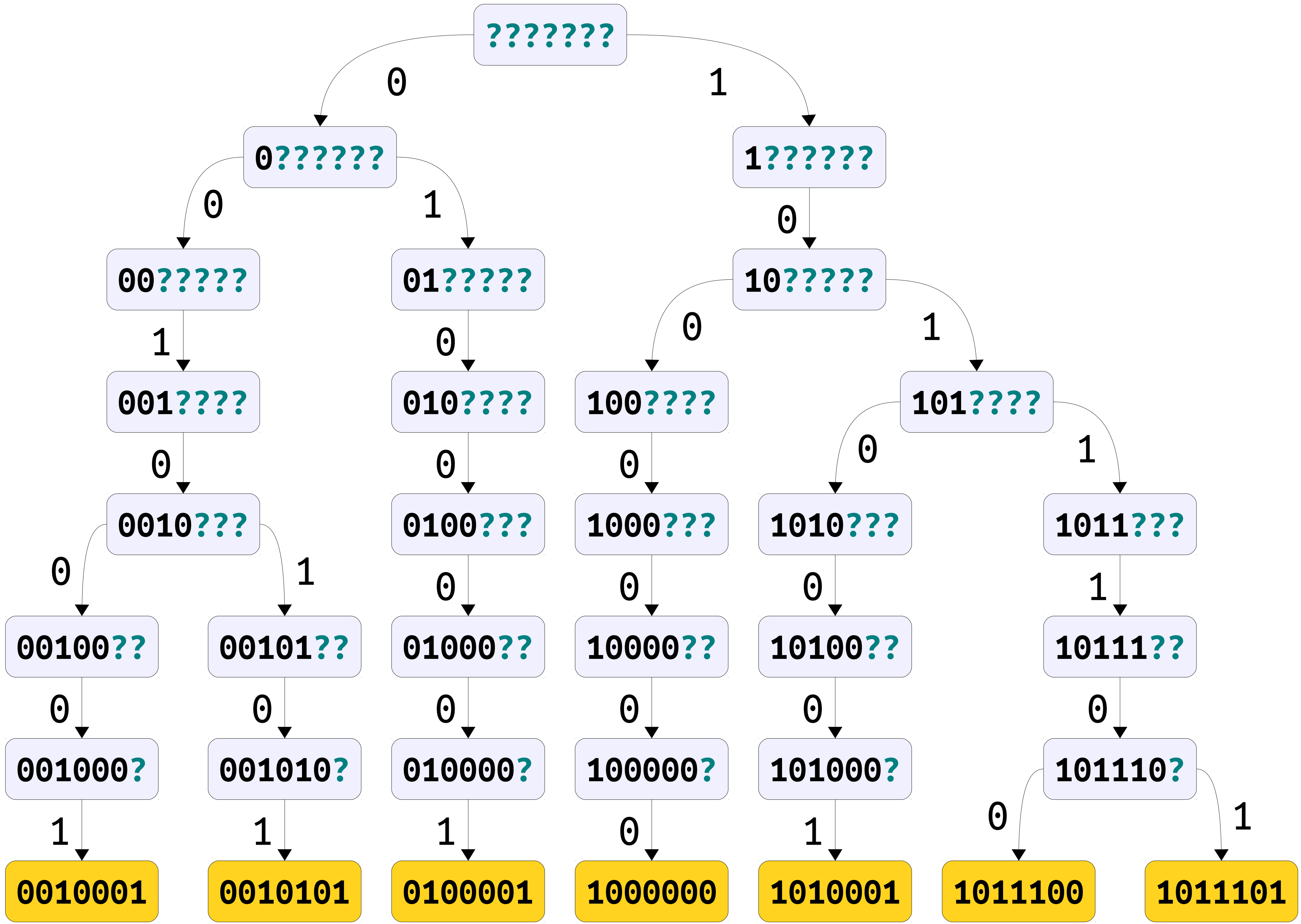

0
1
0
1
0
1
0
0
1
0
0
0
0
1
0
1
0
0
0
1
0
0
0
0
0
0
1
1
1
0
1
0
1



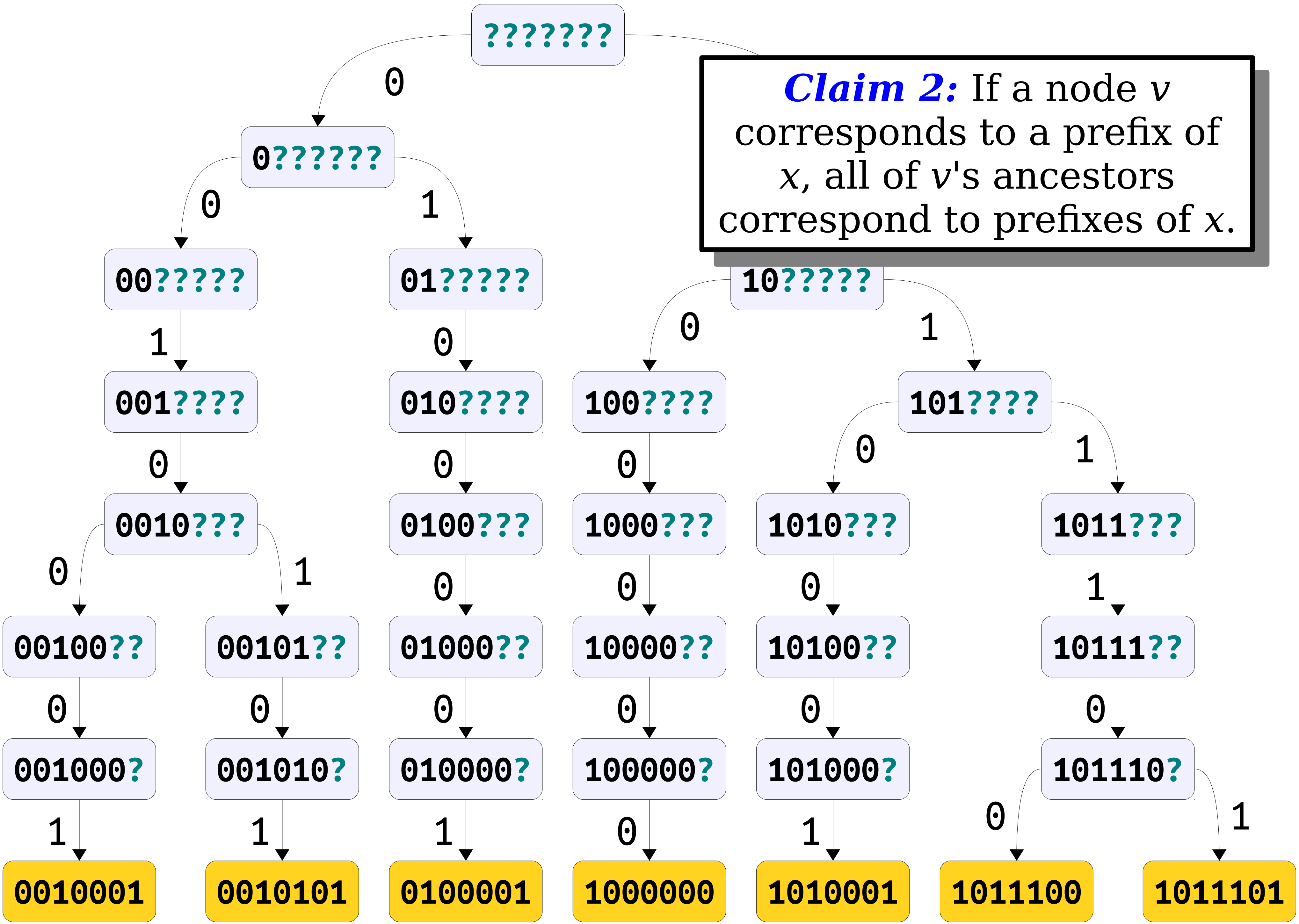
???????
0
1
0??????
1??????
0
1
0
00?????
01?????
10?????
1
0
0
1
001????
010????
100????
101????
0
0
0
0
1
0010???
0100???
1000???
1010???
1011???
0
1
0
0
0
1
00100??
00101??
01000??
10000??
10100??
10111??
0
0
0
0
0
0
001000?
001010?
010000?
100000?
101000?
101110?
1
1
1
0
1
0
1
0010001
0010101
0100001
1000000
1010001
1011100
1011101

Claim 1: The node found during the first phase of a successor query for x corresponds to the longest prefix of x that appears in the trie.

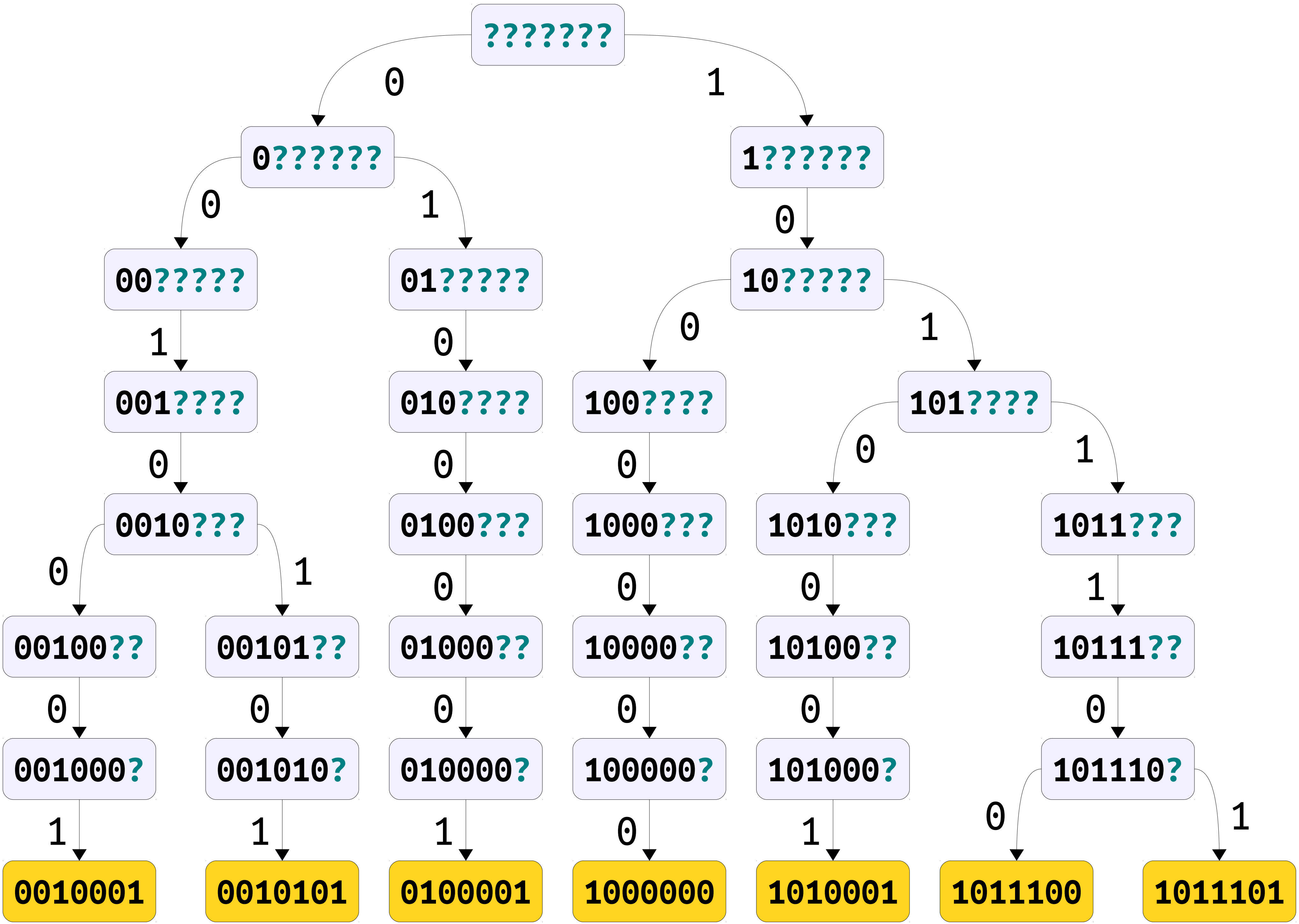


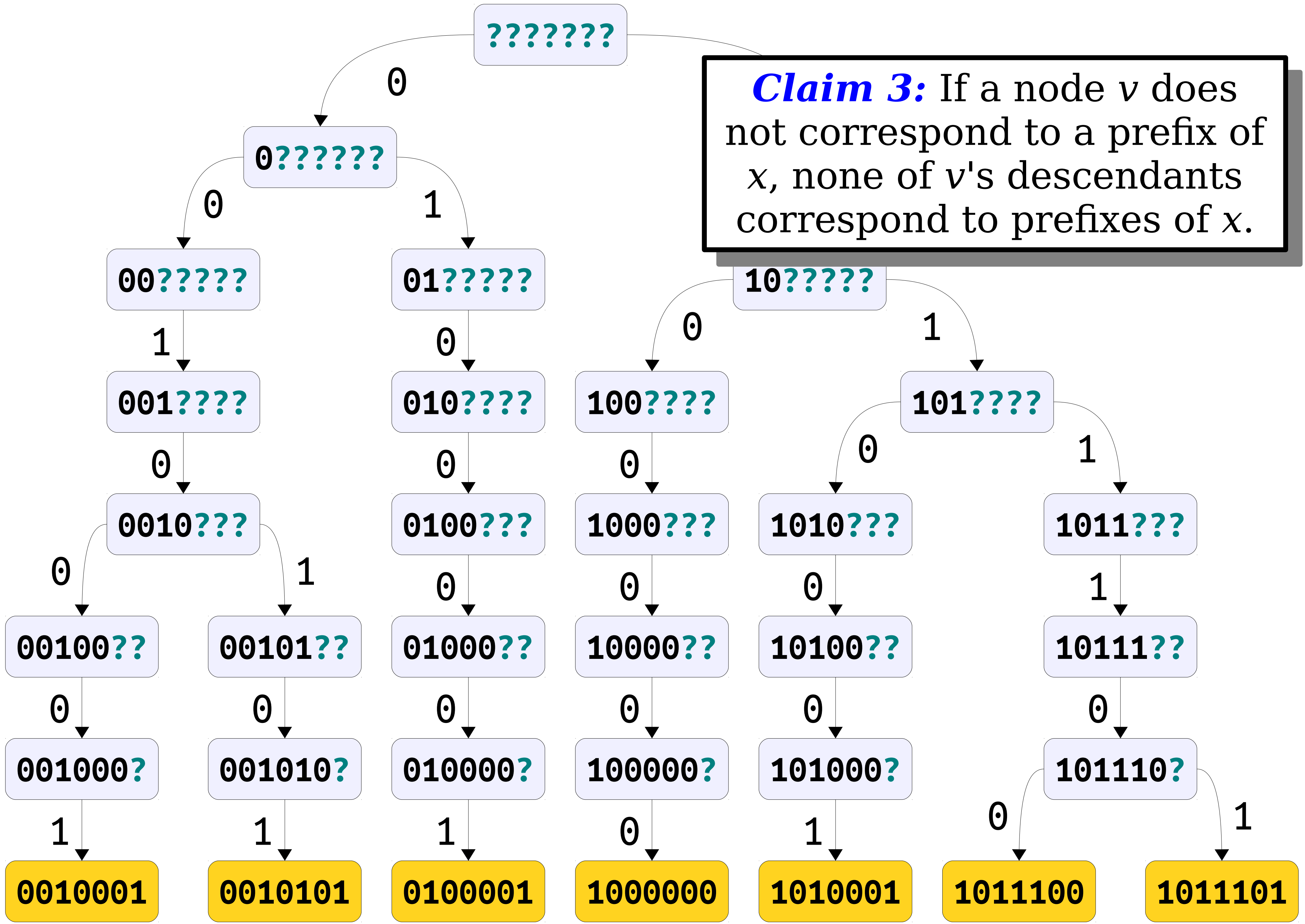


???????
0
1
0??????
1??????
0
1
0
00?????
01?????
10?????
1
0
0
1
001????
010????
100????
101????
0
0
0
0
1
0010???
0100???
1000???
1010???
1011???
0
1
0
0
0
1
00100??
00101??
01000??
10000??
10100??
10111??
0
0
0
0
0
0
001000?
001010?
010000?
100000?
101000?
101110?
1
1
1
0
1
0
1
0010001
0010101
0100001
1000000
1010001
1011100
1011101

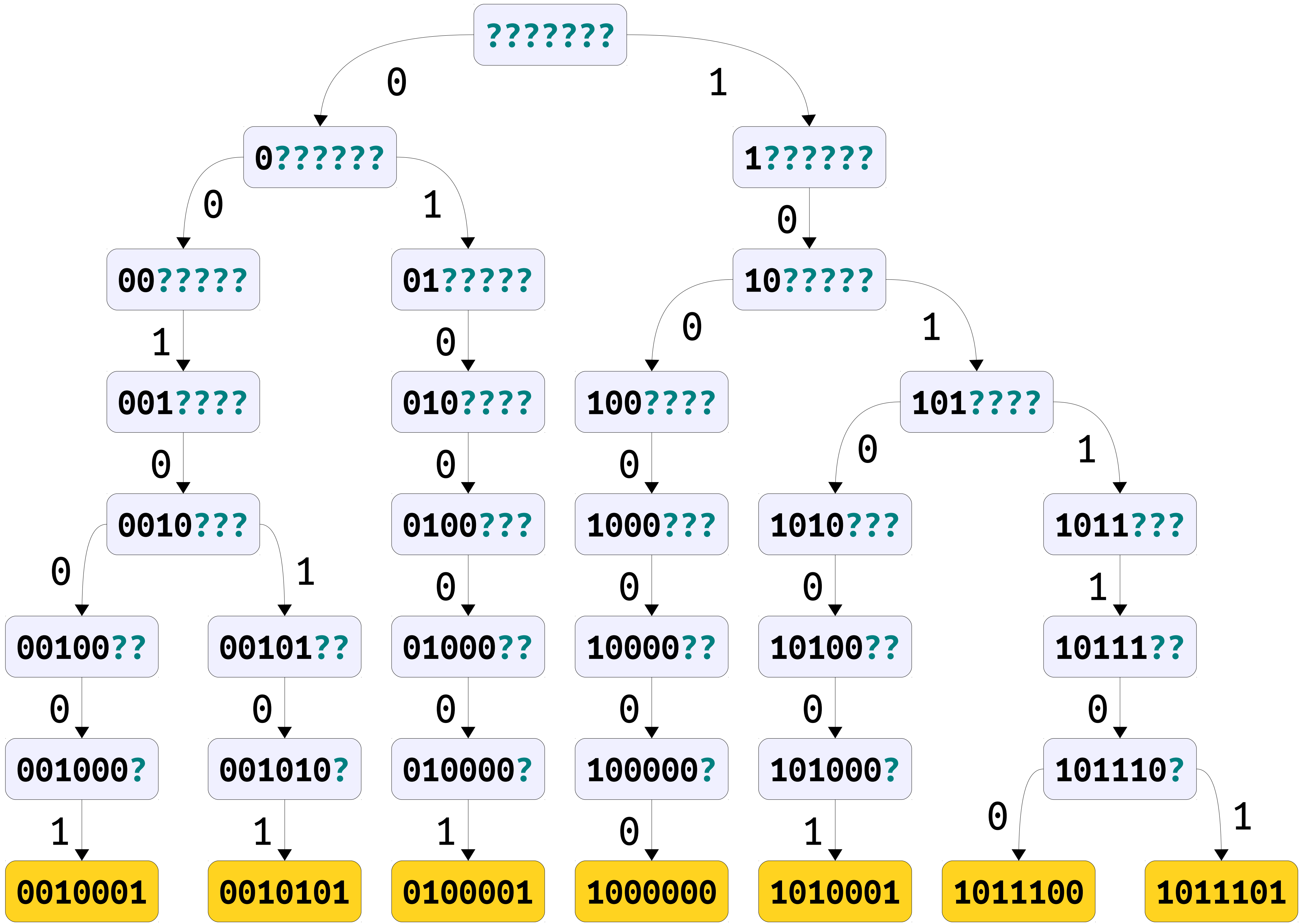


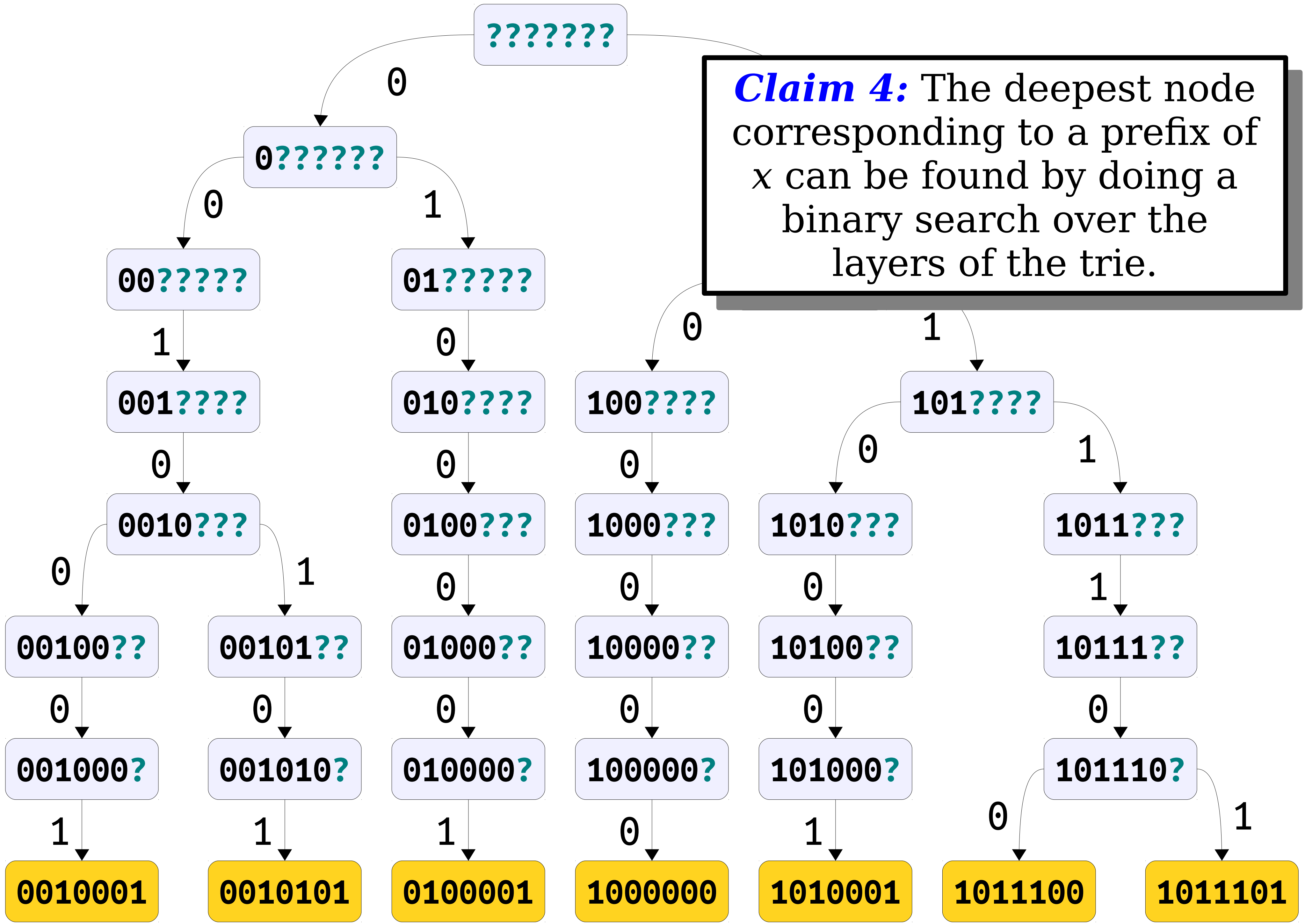
Claim 2: If a node v corresponds to a prefix of x, all of v's ancestors correspond to prefixes of x.
???????
0??????
00?????
01?????
10?????
001????
010????
100????
101????
0010???
0100???
1000???
1010???
1011???
00100??
00101??
01000??
10000??
10100??
10111??
001000?
001010?
010000?
100000?
101000?
101110?
0010001
0010101
0100001
1000000
1010001
1011100
1011101


???????
0??????
1??????
00?????
01?????
10?????
001????
010????
100????
101????
0010???
0100???
1000???
1010???
1011???
00100??
00101??
01000??
10000??
10100??
10111??
001000?
001010?
010000?
100000?
101000?
101110?
0010001
0010101
0100001
1000000
1010001
1011100
1011101



Claim 3: If a node v does not correspond to a prefix of x, none of v's descendants correspond to prefixes of x.


???????
0??????
1??????
00?????
01?????
10?????
001????
010????
100????
101????
0010???
0100???
1000???
1010???
1011???
00100??
00101??
01000??
10000??
10100??
10111??
001000?
001010?
010000?
100000?
101000?
101110?
0010001
0010101
0100001
1000000
1010001
1011100
1011101



Claim 4: The deepest node corresponding to a prefix of x can be found by doing a binary search over the layers of the trie.
???????
0??????
00?????
01?????
001????
010????
100????
101????
0010???
0100???
1000???
1010???
1011???
00100??
00101??
01000??
10000??
10100??
10111??
001000?
001010?
010000?
100000?
101000?
101110?
0010001
0010101
0100001
1000000
1010001
1011100
1011101

One Speedup

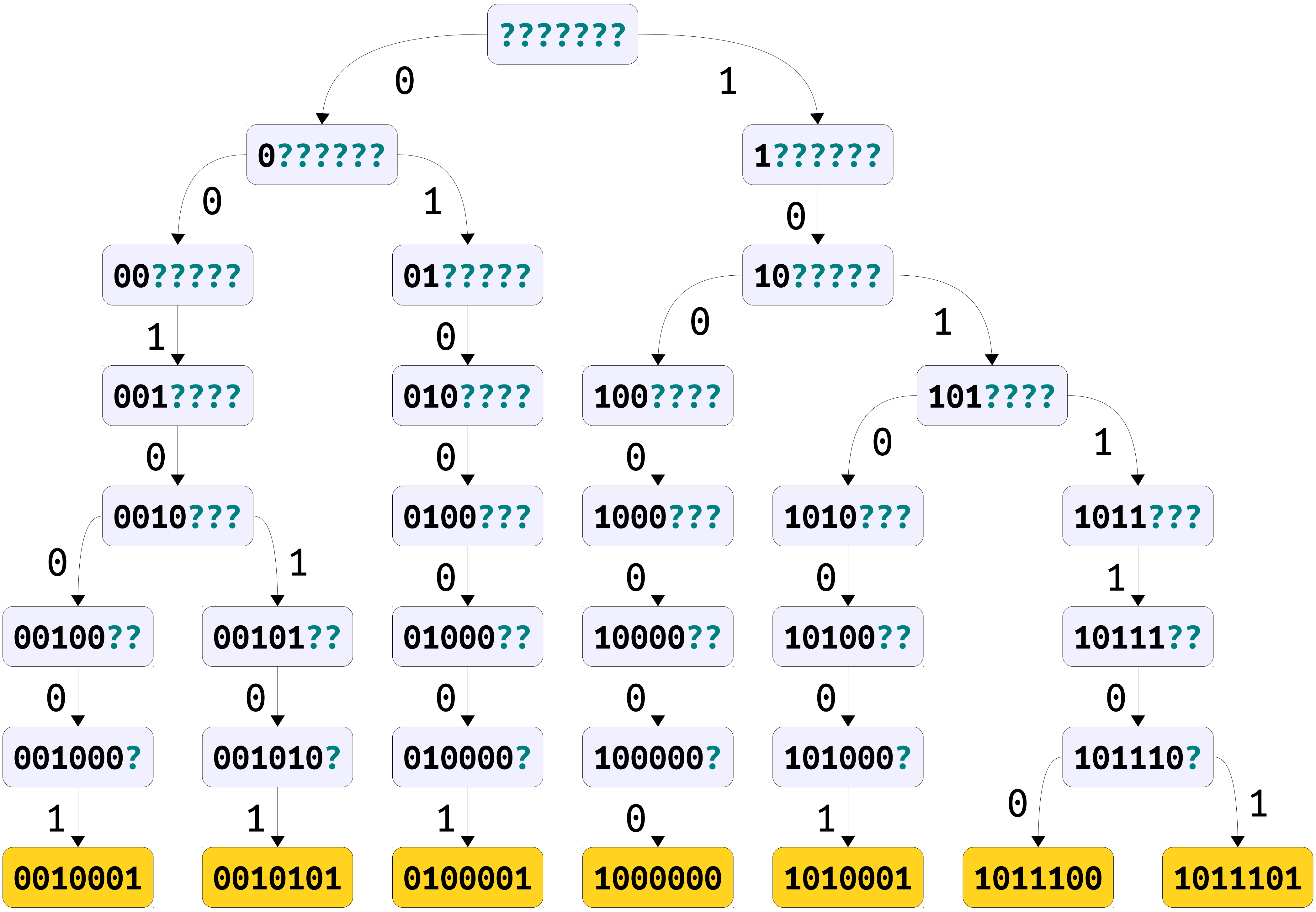
- ***Goal:*** Encode the trie so that we can do a binary search over its layers.
- ***One Solution:*** Store an array of cuckoo hash tables, one per layer of the trie, that stores all the nodes in that layer.
- Can now query, in worst-case time O(1), whether a node's prefix is present on a given layer.
- There are O(log U) layers in the trie.
- Binary search will take worst-case time **O(log log U)**.
- ***Nice side-effect:*** Queries are now worst-case O(1), since we can just check the hash table at the bottom layer.

The Next Issue

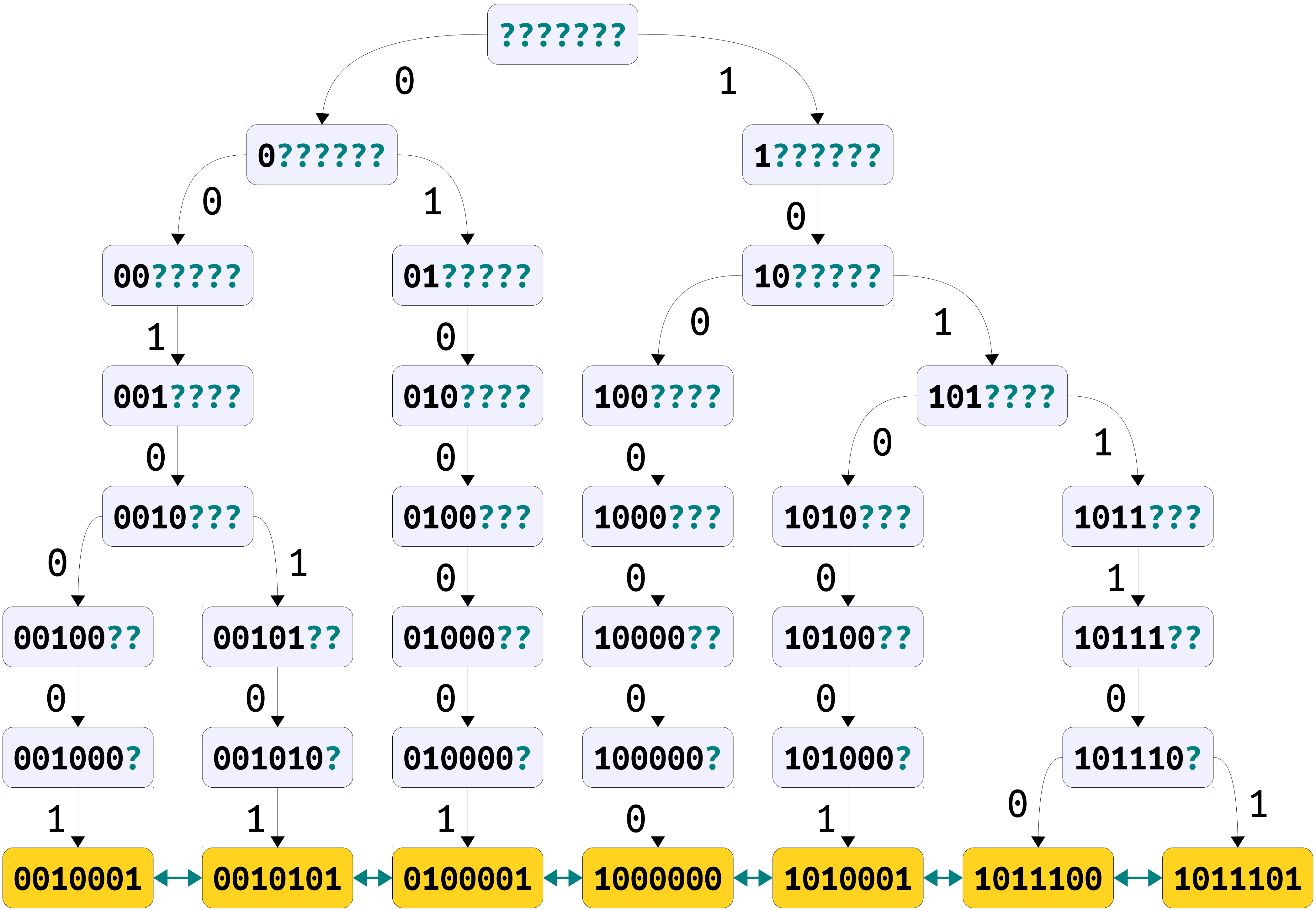
- We can now find the node where the successor search would initially arrive.
- However, after arriving there, we have to back up to a node with a 1 child we didn't follow on the path down.
- This will take time O(log U).
- Can we do better?

A Useful Observation

- Our binary search for the longest prefix of x will either stop at
  - a leaf node (so x is present), or
  - an internal node.
- If we stop at a leaf node, the successor will be the next leaf in the trie.
- ***Idea:*** Thread a doubly-linked list through the leaf nodes.



???????
0
1
0??????
1??????
00?????
01?????
10?????
001????
010????
100????
101????
0010???
0100???
1000???
1010???
1011???
00100??
00101??
01000??
10000??
10100??
10111??
001000?
001010?
010000?
100000?
101000?
101110?
0010001
0010101
0100001
1000000
1010001
1011100
1011101

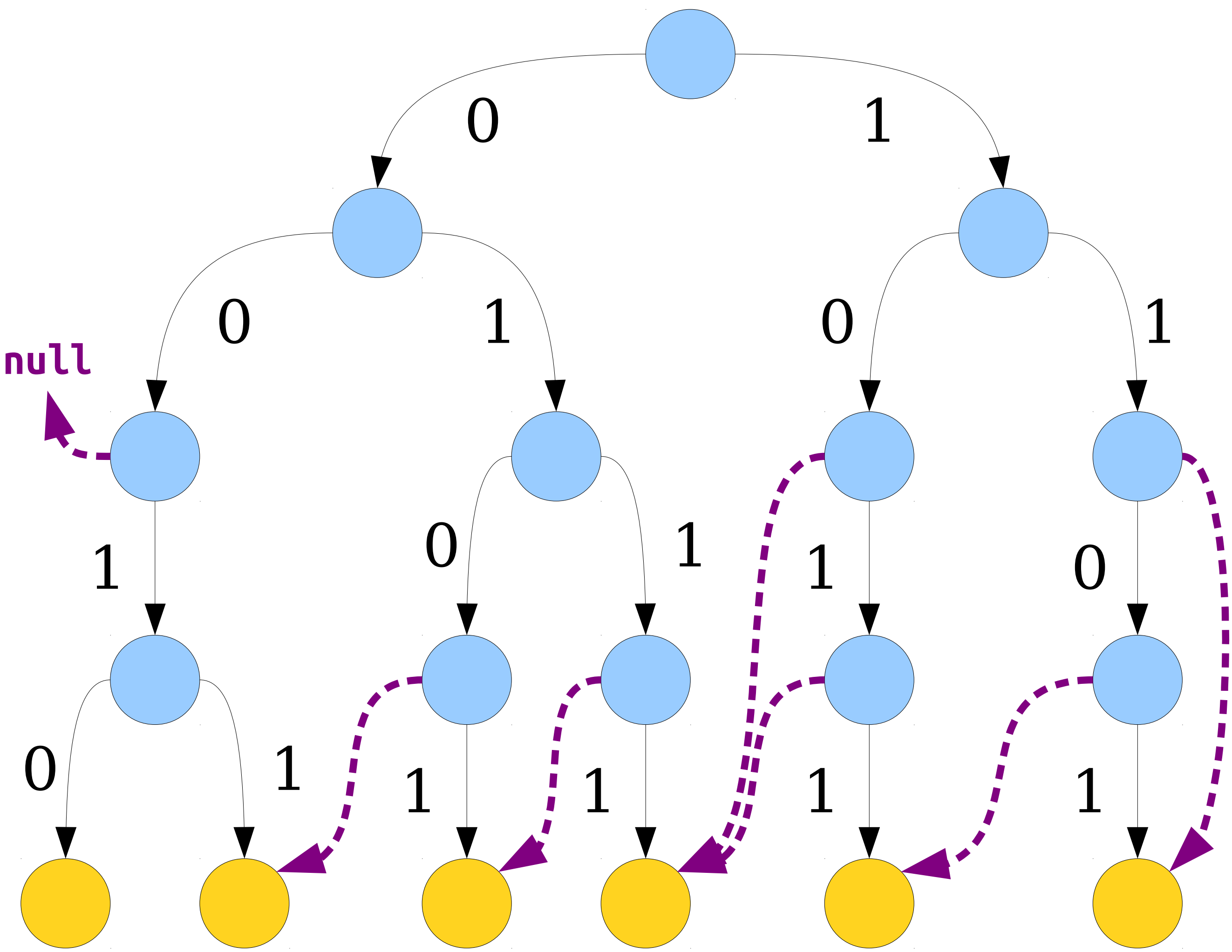

???????
0??????
1??????
00?????
01?????
10?????
001????
010????
100????
101????
0010???
0100???
1000???
1010???
1011???
00100??
00101??
01000??
10000??
10100??
10111??
001000?
001010?
010000?
100000?
101000?
101110?
0010001
0010101
0100001
1000000
1010001
1011100
1011101

Successors of Internal Nodes

- ***Claim:*** If the binary search terminates at an internal node, that node must only have 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 pointer and use it to speed up successor and predecessor searches.

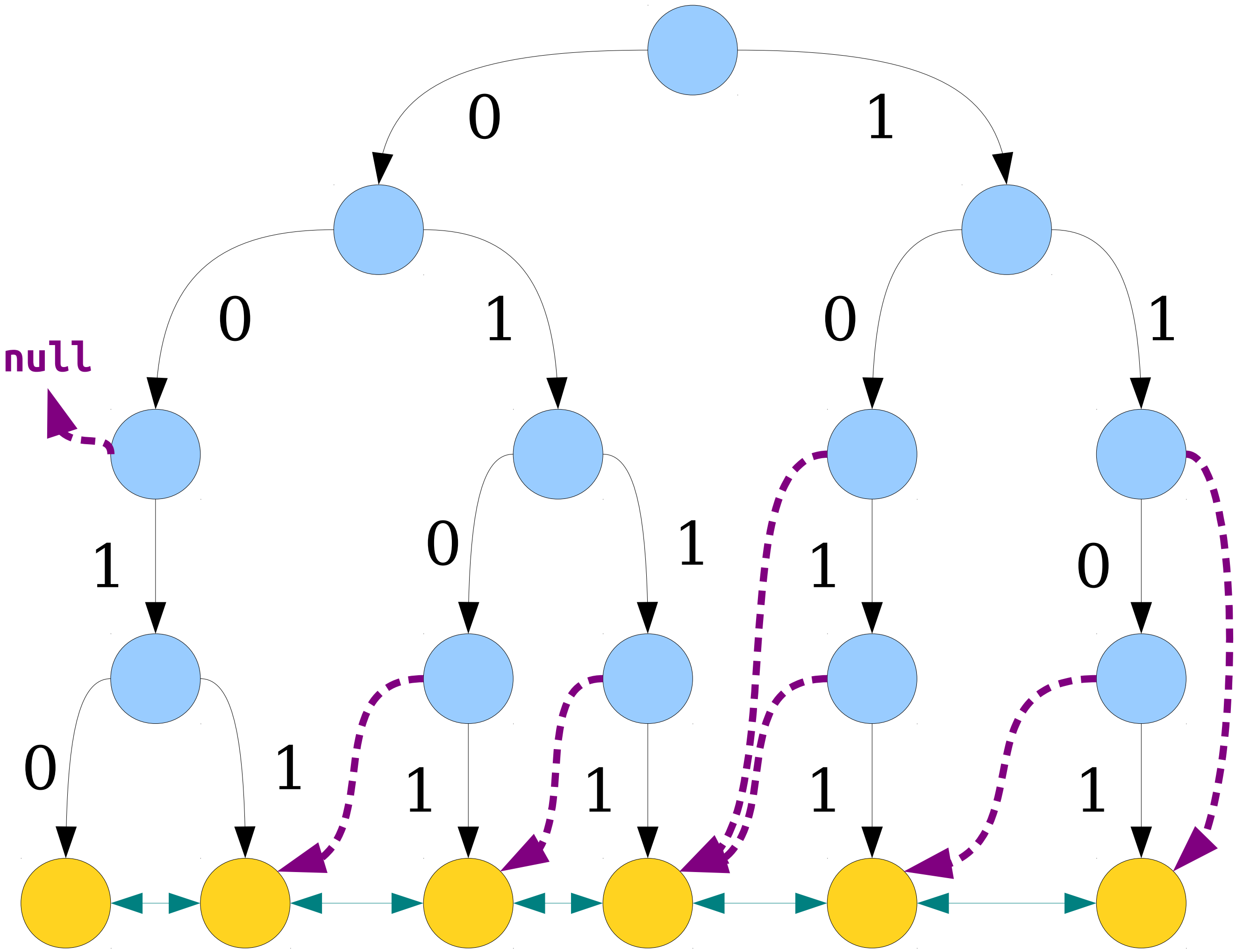
Threaded Binary Tries

- A ***threaded binary trie*** is a binary tree where
  - each missing 0 pointer points to the inorder predecessor of the node and
  - each missing 1 points to the inorder successor of the node.
- Related to threaded binary search trees; read up on them if you're curious!



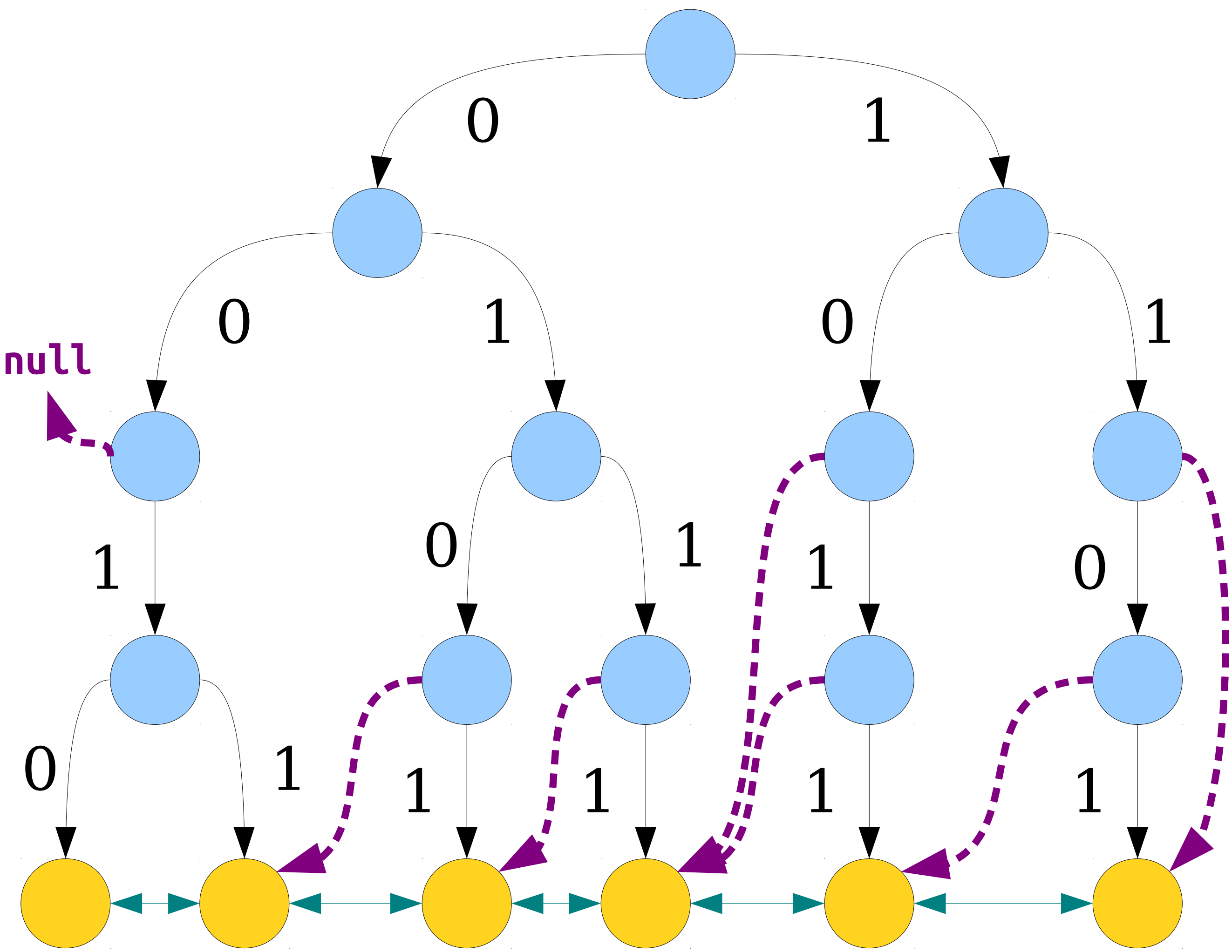
x-Fast Tries

- An ***x-Fast Trie*** is a threaded binary trie where leaves are stored in a doubly-linked list and where all nodes in each level are stored in a hash table.
- Can do lookups in time O(1).



x-Fast Tries

- ***Claim:*** Can determine ***successor***(*x*) in time O(log log *U*).
- Start by binary searching for the longest prefix of *x*.
- If at a leaf node, follow the forward pointer to the successor.
- If at an internal node, follow the thread pointer to a leaf node. Either return that value or the one after it, depending on how it compares to *x*.

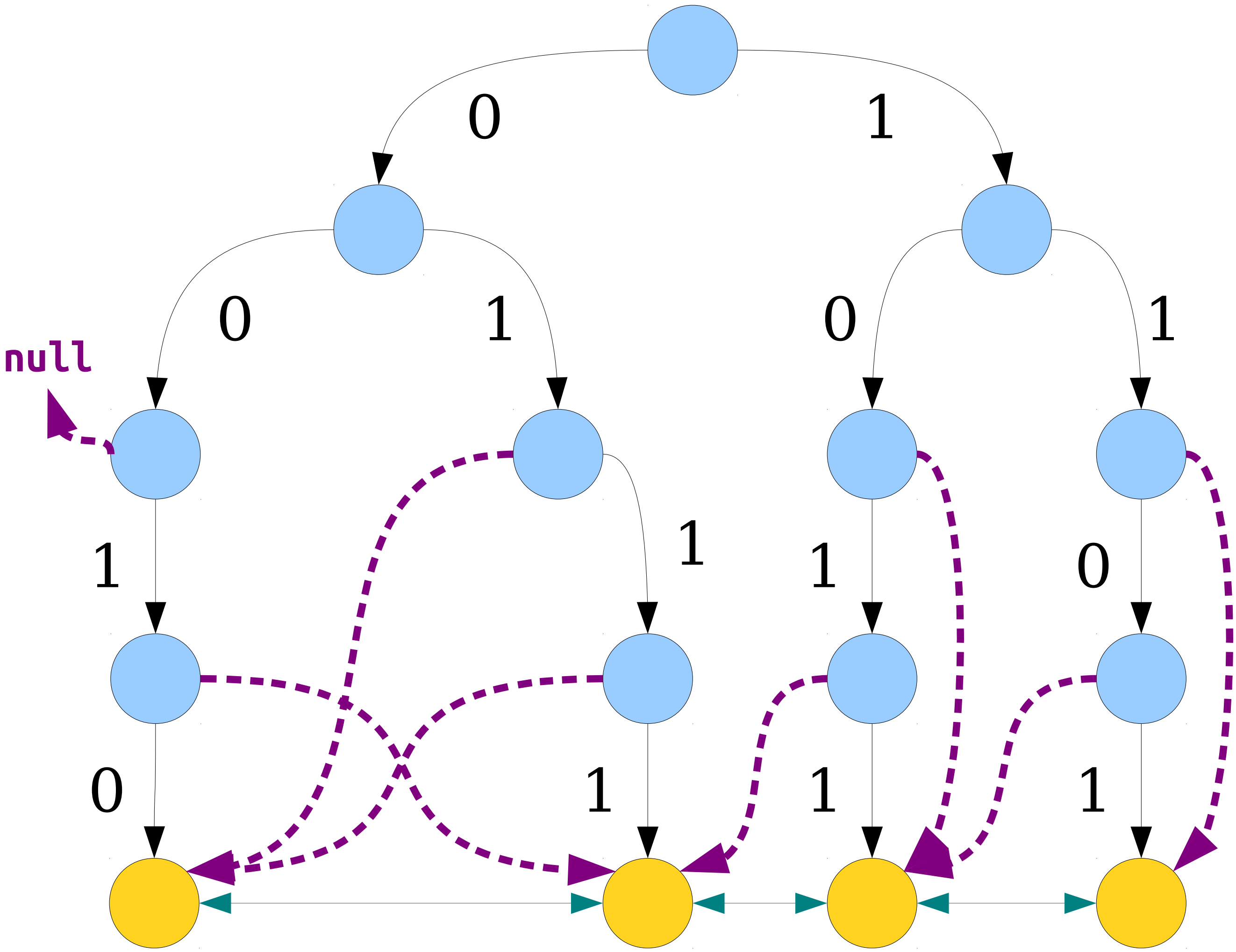


x-Fast Trie Maintenance

- Based on what we've seen:
  - Lookups take worst-case time O(1).
  - Successor and predecessor queries take worst-case time O(log log U).
  - Min and max can be done in time O(log log U) by finding the predecessor of ∞ or the successor of -∞.
- How efficiently can we support insertions and deletions?

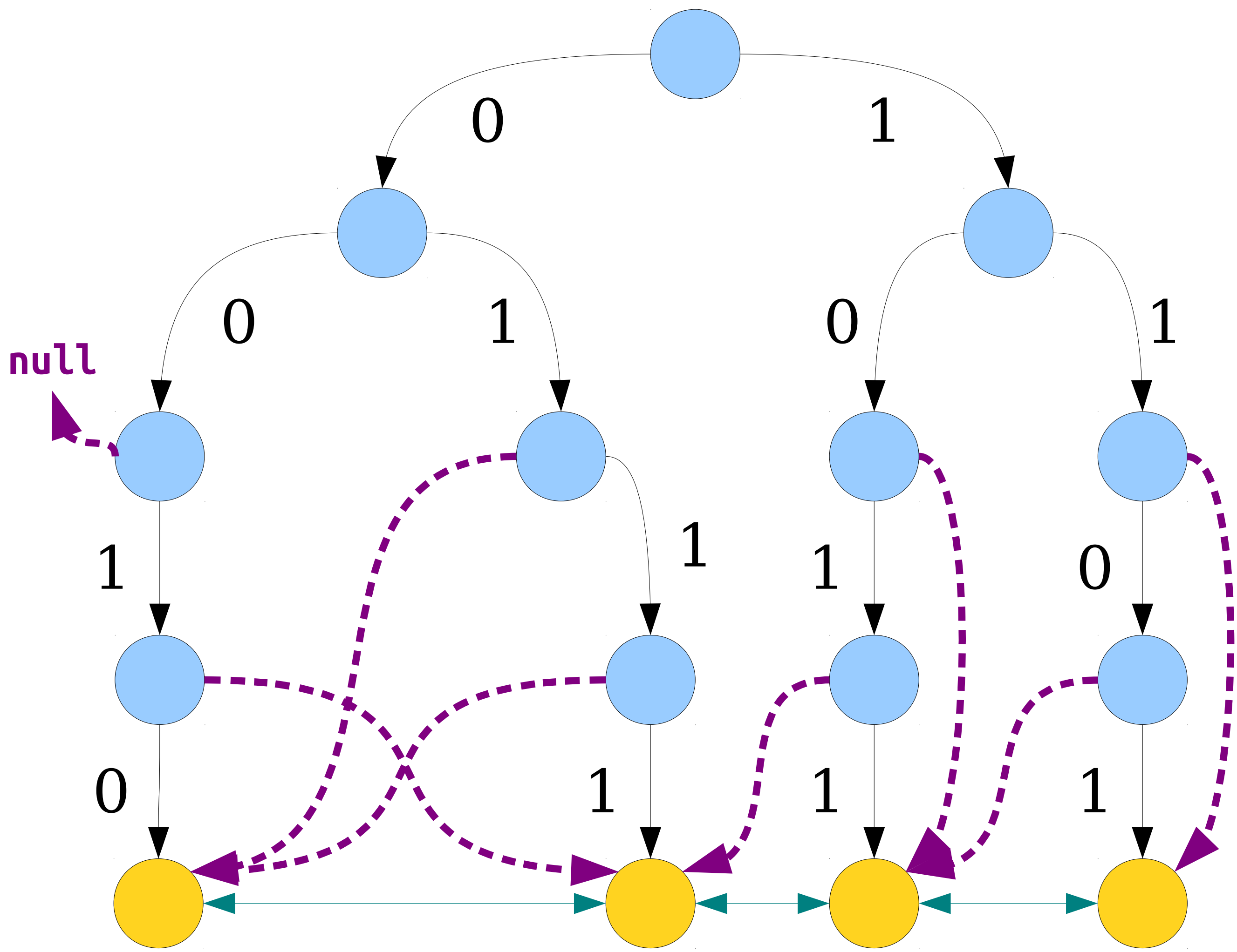
x-Fast Tries

- If we ***insert***(x), we need to
  - Add some new nodes to the trie.
  - Wire x into the doubly-linked list of leaves.
  - Update the thread pointers to include x.
- Worst-case will be $\Omega(\log U)$ due to the first and third steps.



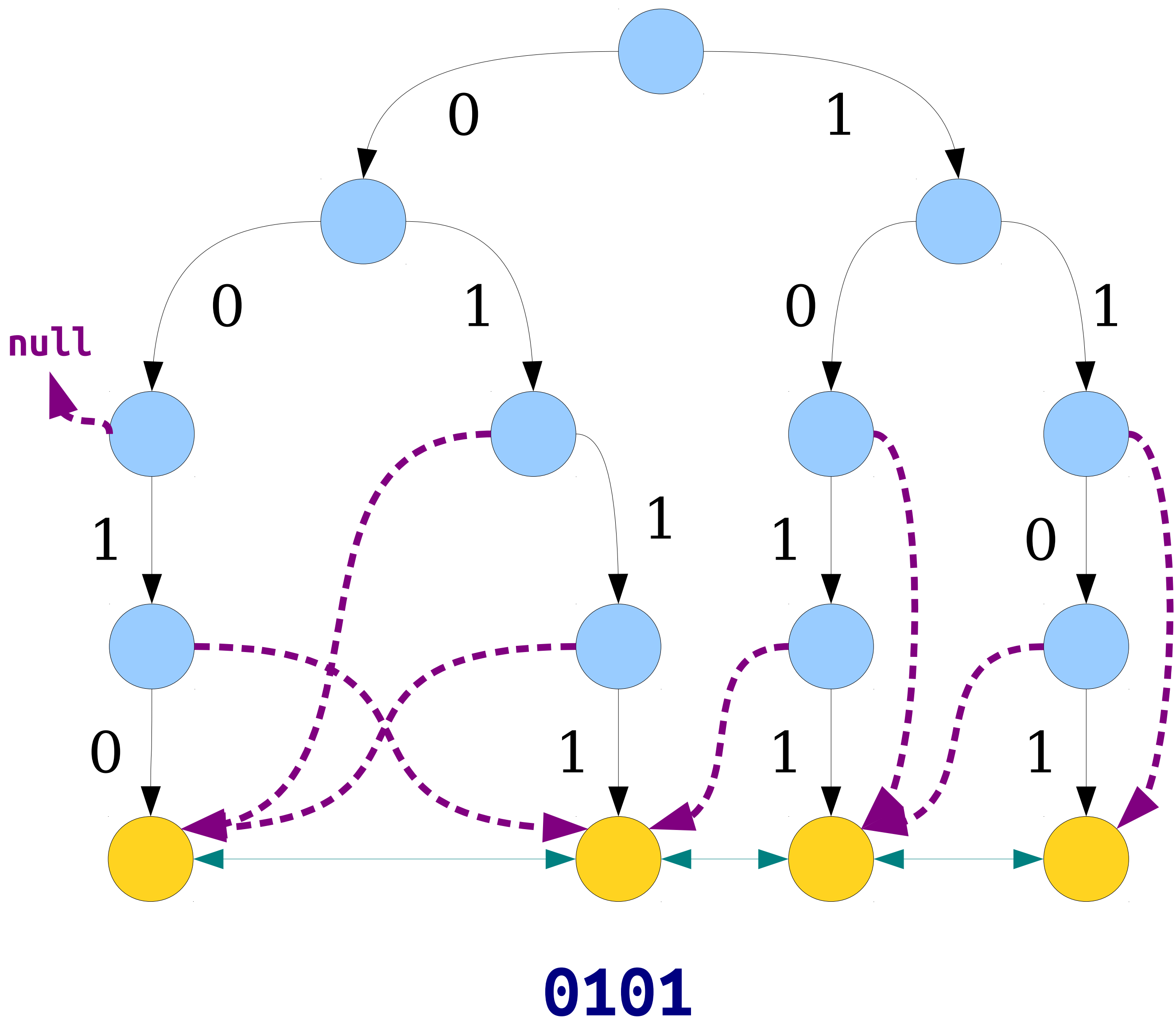
x-Fast Tries

- Here is an (amortized, expected) O(log *U*) 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.



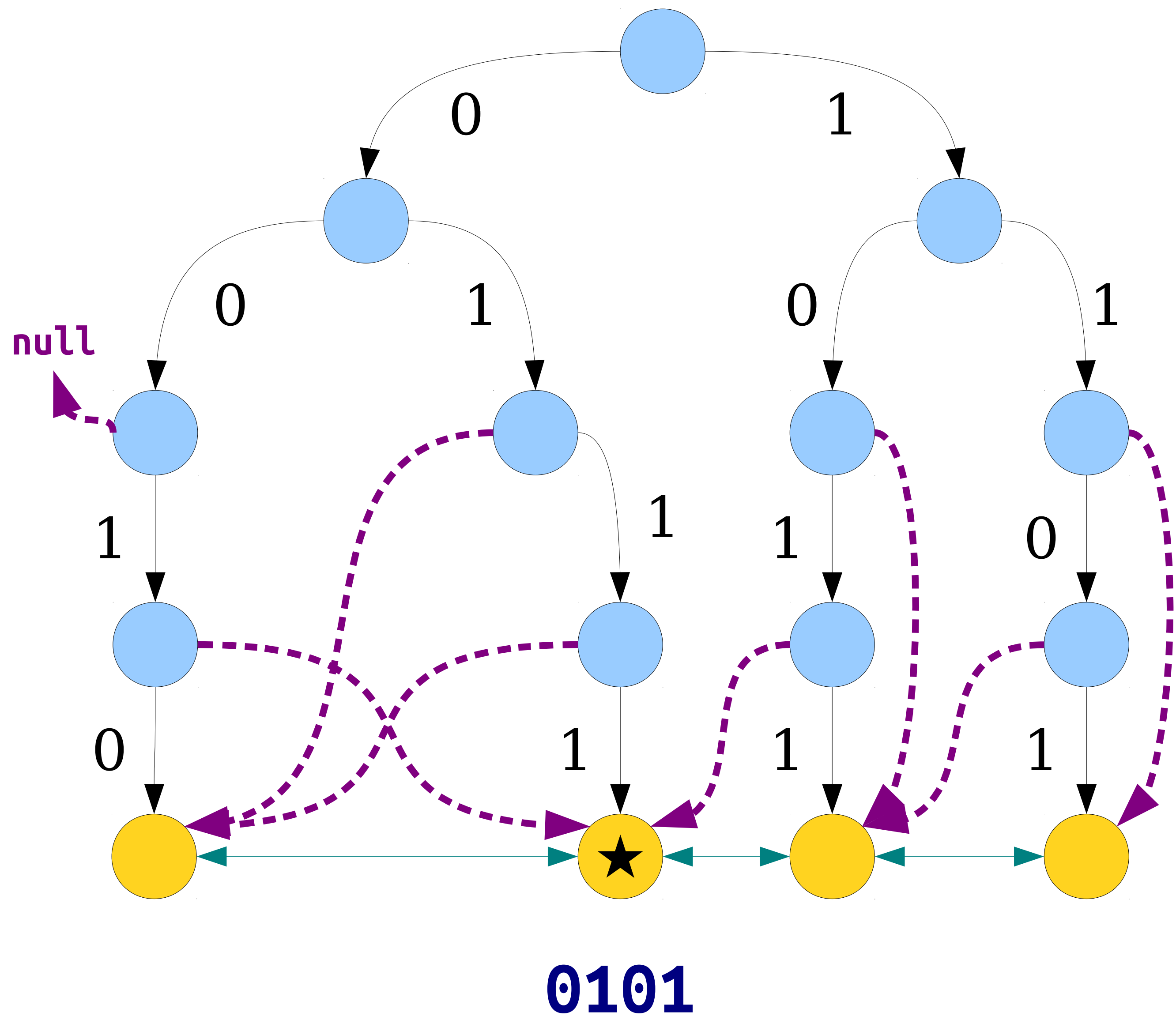
x-Fast Tries

- Here is an (amortized, expected) O(log *U*) 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.



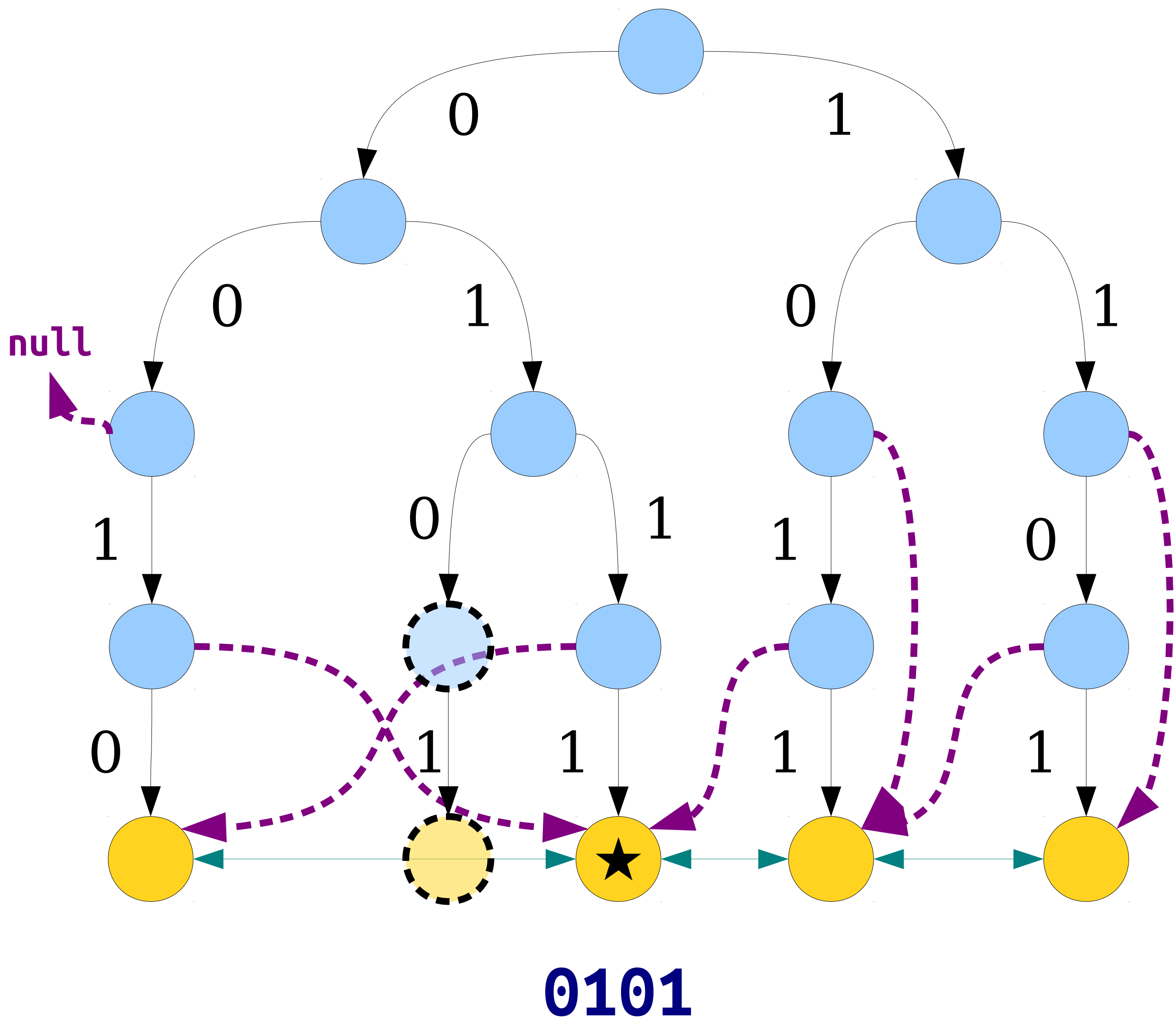
x-Fast Tries

- Here is an (amortized, expected) O(log *U*) 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.



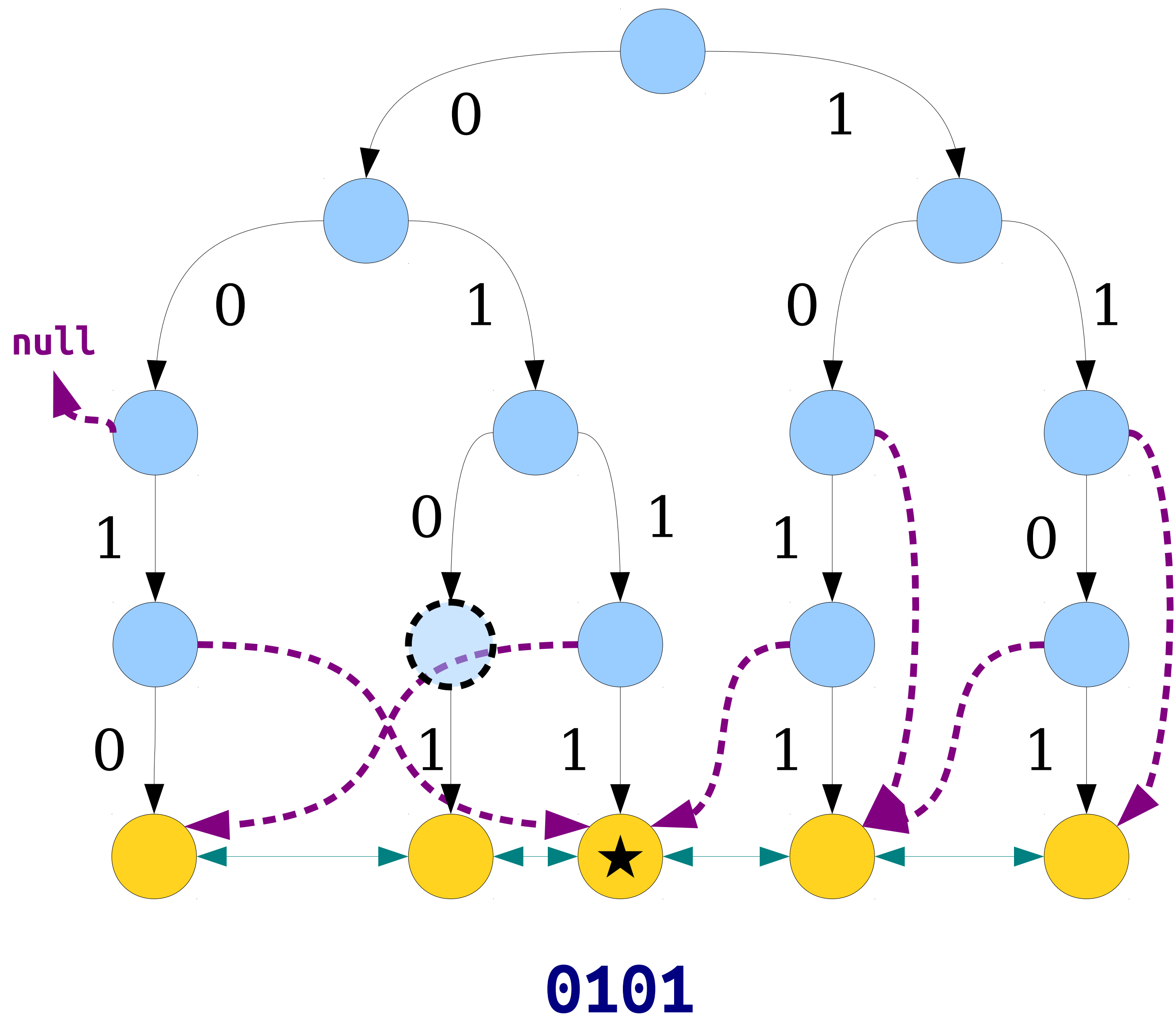
x-Fast Tries

- Here is an (amortized, expected) O(log *U*) 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.



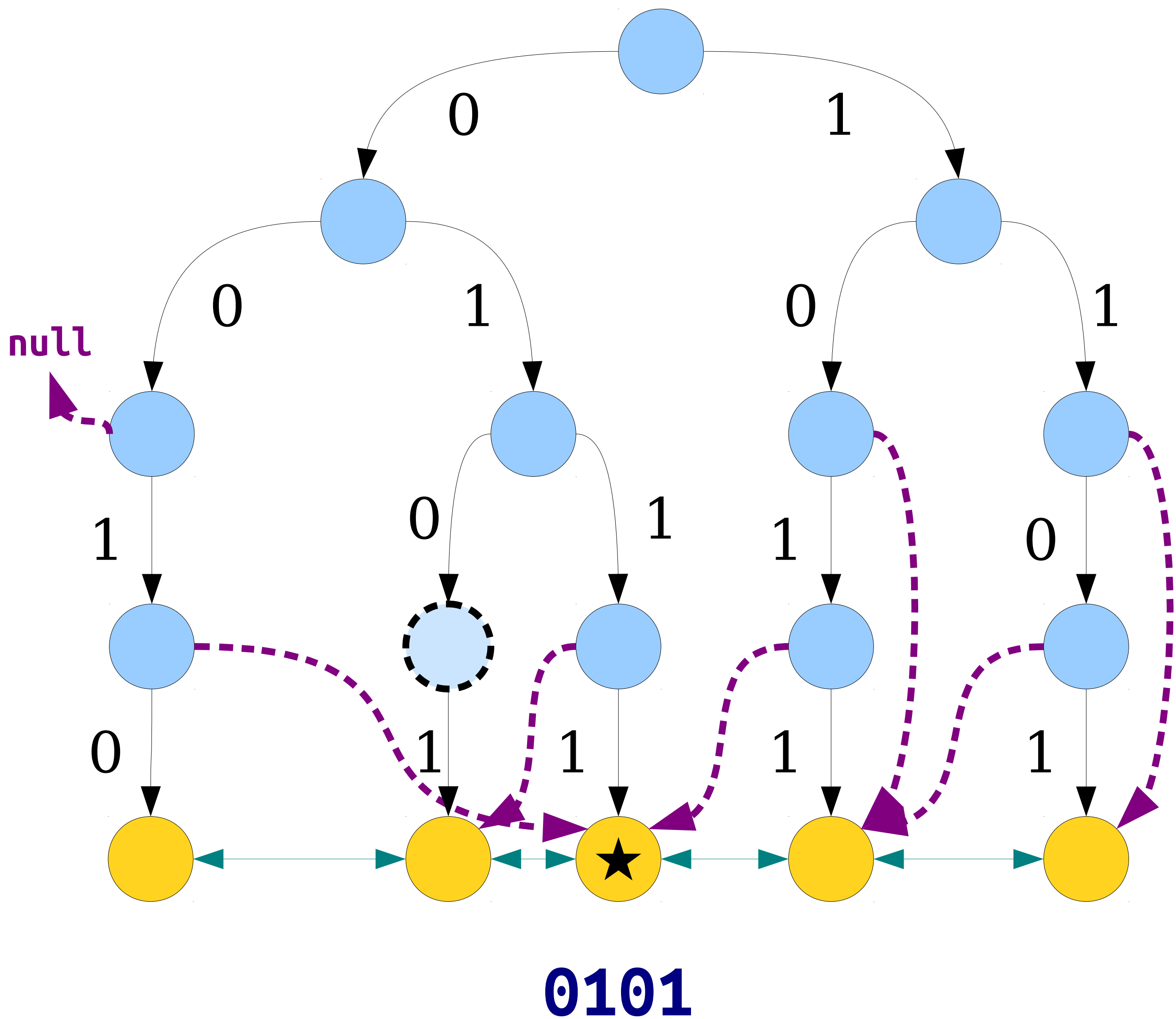
x-Fast Tries

- Here is an (amortized, expected) O(log *U*) 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.



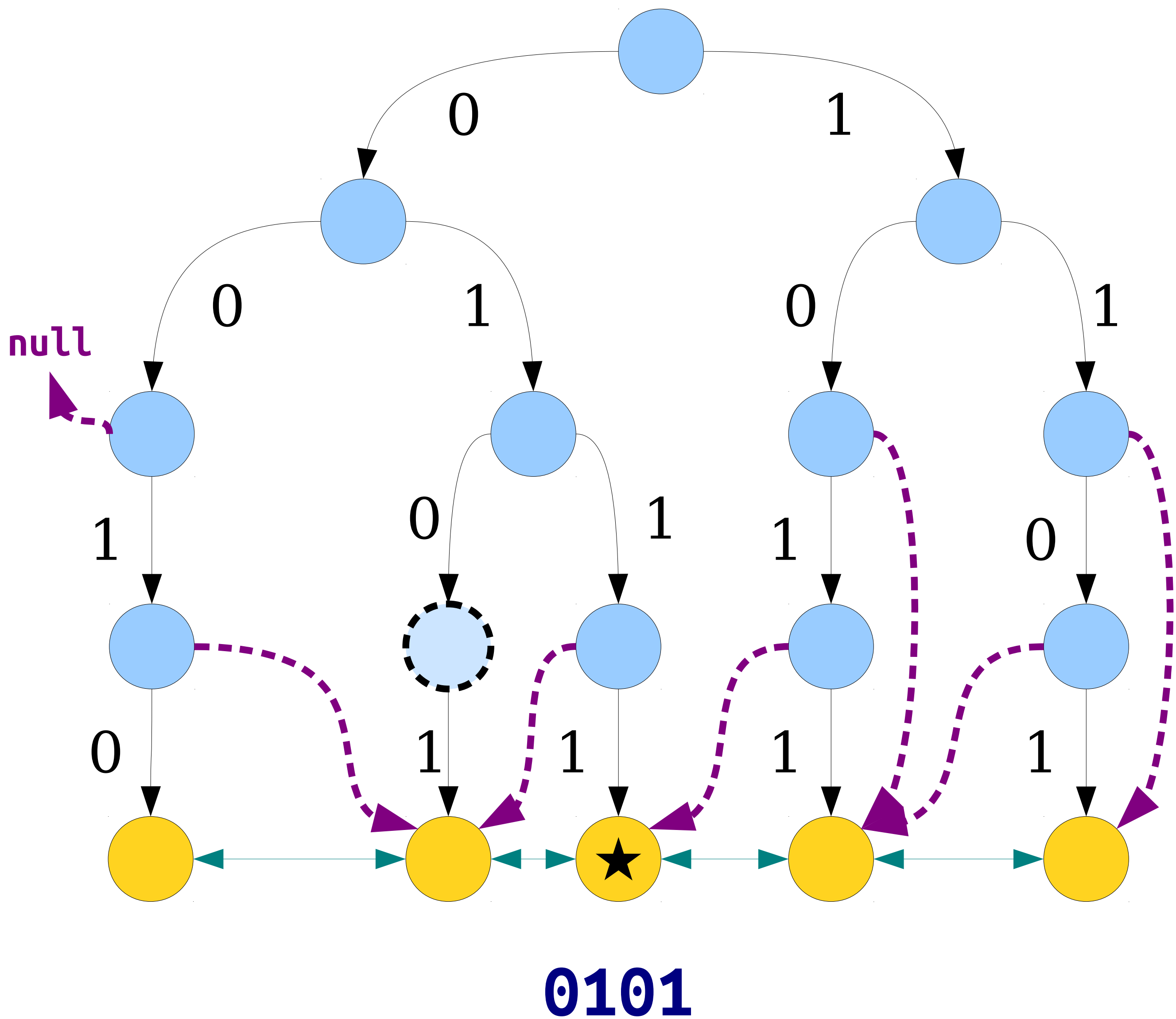
x-Fast Tries

- Here is an (amortized, expected) O(log U) 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.



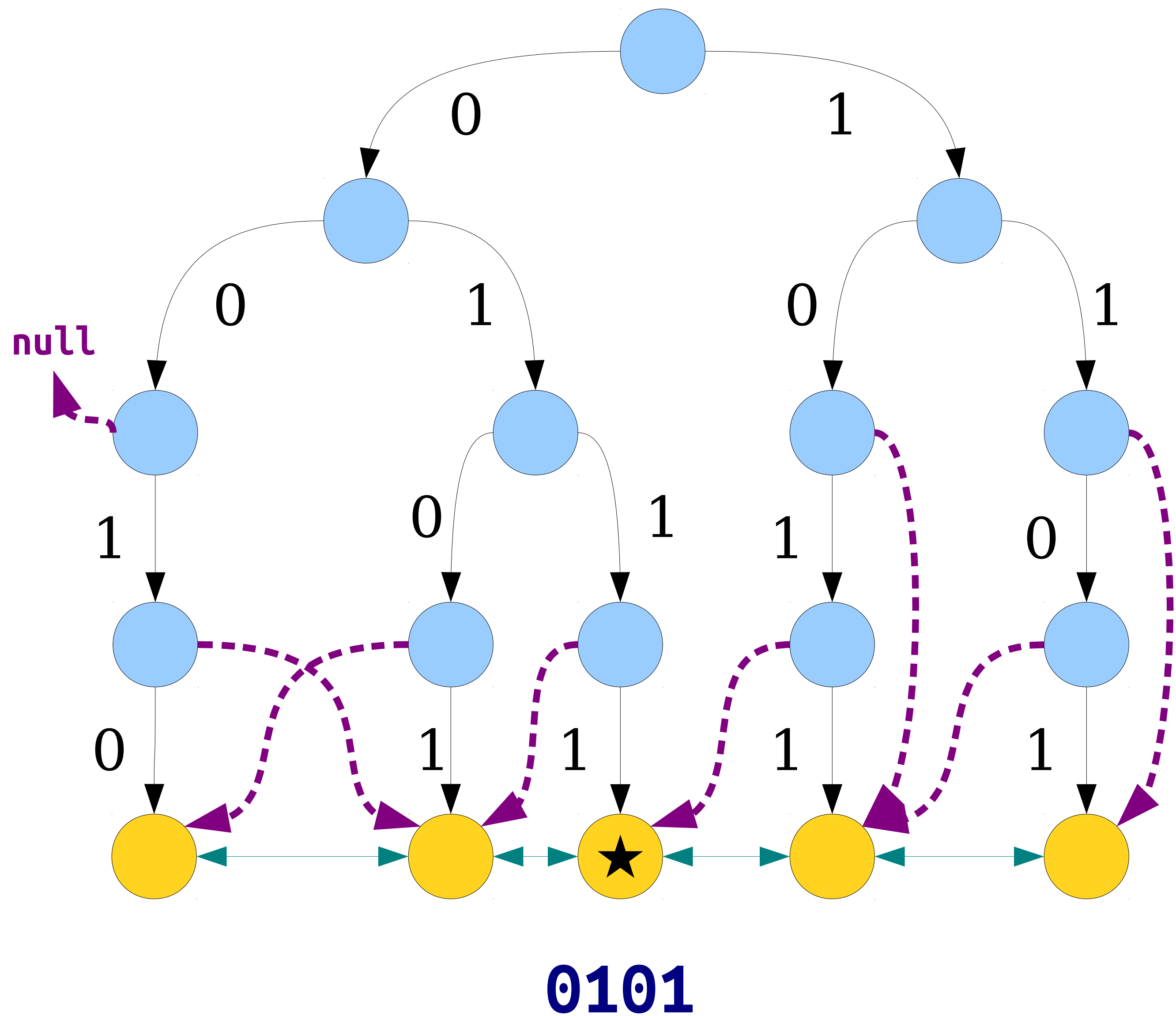
x-Fast Tries

- Here is an (amortized, expected) O(log *U*) 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(log *U*) 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(log *U*)**.
- Full details are left as a proverbial Exercise to the Reader. ☺

Space Usage

- How much space is required in an x-fast trie?
- Each leaf node contributes at most O(log U) nodes in the trie.
- Total space usage for hash tables is proportional to total number of trie nodes.
- Total space: **O(n log U)**.

For Reference

- van Emde Boas tree
  - ***insert***: O(log log U)
  - ***delete***: O(log log U)
  - ***lookup***: O(log log U)
  - ***max***: O(1)
  - ***succ***: O(log log U)
  - ***is-empty***: O(1)
  - Space: O(U)
- x-Fast Trie
  - ***insert***: O(log U)*
  - ***delete***: O(log U)*
  - ***lookup***: O(1)
  - ***max***: O(log log U)
  - ***succ***: O(log log U)
  - ***is-empty***: O(1)
  - Space: O(n log U)

* Expected, amortized

What Remains

- We need to speed up ***insert*** and ***delete*** to run in time O(log log U).
- We'd like to drop the space usage down to O(n).
- How can we do this?

- x-Fast Trie
  - ***insert***: O(log U)*
  - ***delete***: O(log U)*
  - ***lookup***: O(1)
  - ***max***: O(log log U)
  - ***succ***: O(log log U)
  - ***is-empty***: O(1)
  - Space: O(n log U)

  * Expected, amortized

Time-Out for Announcements!

Problem Set Five

- Problem Set Five was due today at 3:00PM.
  - If you use all your remaining late days, it's due at Saturday at 3:00PM.
- We're going to aim to get this graded before the midterm.
- Solutions will go out on Monday. We'll put them in the filing cabinet in the Gates building.

Midterm Logistics

- As a reminder, the midterm is next Tuesday from 7:00PM – 10:00PM in **320-105**.
- Closed-book, closed-computer, and limited-note. You can bring a double-sided 8.5” × 11” sheet of notes with you to the exam.
- Solutions to the practice problems are available up front. They'll be in Gates if you missed class today.
  - ***Gates is locked over the weekend***, so please stop by to pick them up before then. Otherwise, you'll have to wait until Monday unless you have a Gates key.

Final Project Presentations

- Final project presentations will run from **Tuesday, May 31** to **Thursday, June 2**.
- The following link will let you sign up for time slots:

  http://www.slottr.com/sheets/1197528
- This will be open from noon on Monday, May 23 until noon on Friday, May 27. It's first-come, first-served.
- Presentations will be 10-15 minutes, plus five minutes for questions. Please arrive five minutes early to get set up.
- Presentations are open to the public, so feel free to stop by any of the presentations you're interested in.

Back to CS166!

y-Fast Tries

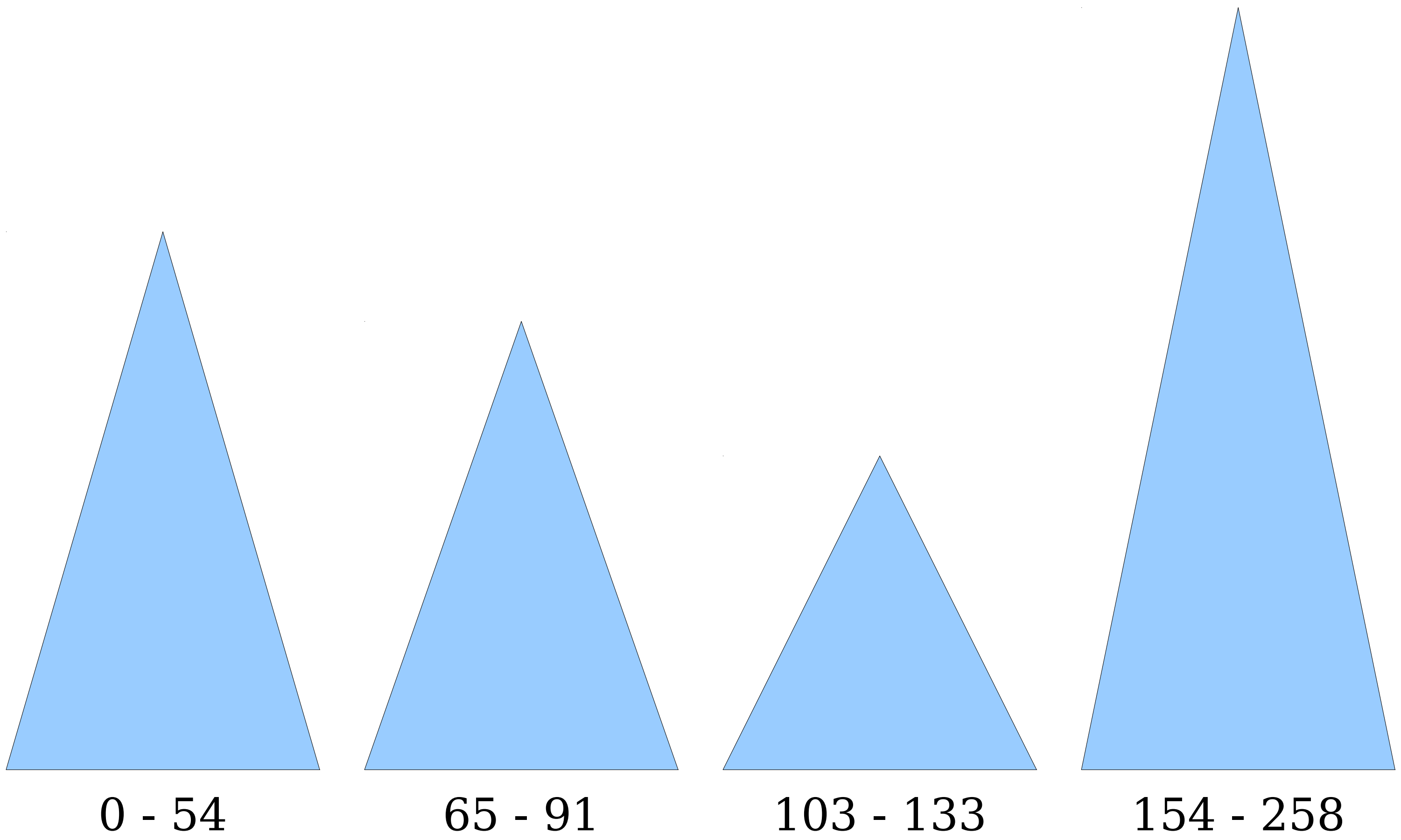
y-Fast Tries

- The ***y-Fast Trie*** is a data structure that will match the vEB time bounds in an expected, amortized sense while requiring only O(n) space.
- It's built out of an x-fast trie and a collection of red/black trees.

The Motivating Idea

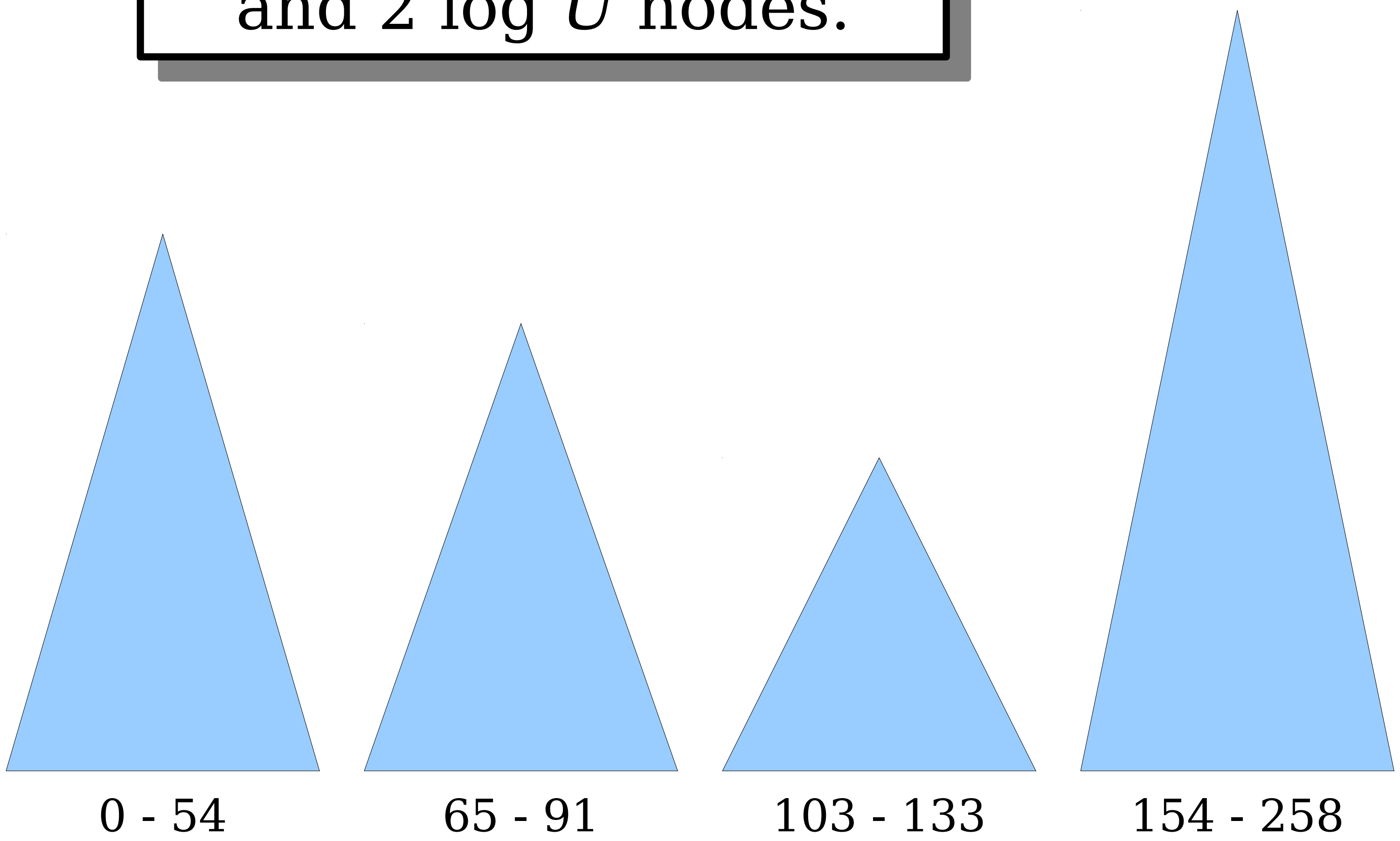
- Suppose we have a red/black tree with $\Theta(\log U)$ nodes.
- Any ordered dictionary operation on the tree will then take time $O(\log \log U)$.
- ***Idea:*** Store the elements in the ordered dictionary in a collection of red/black trees with $\Theta(\log U)$ elements each.

The Idea



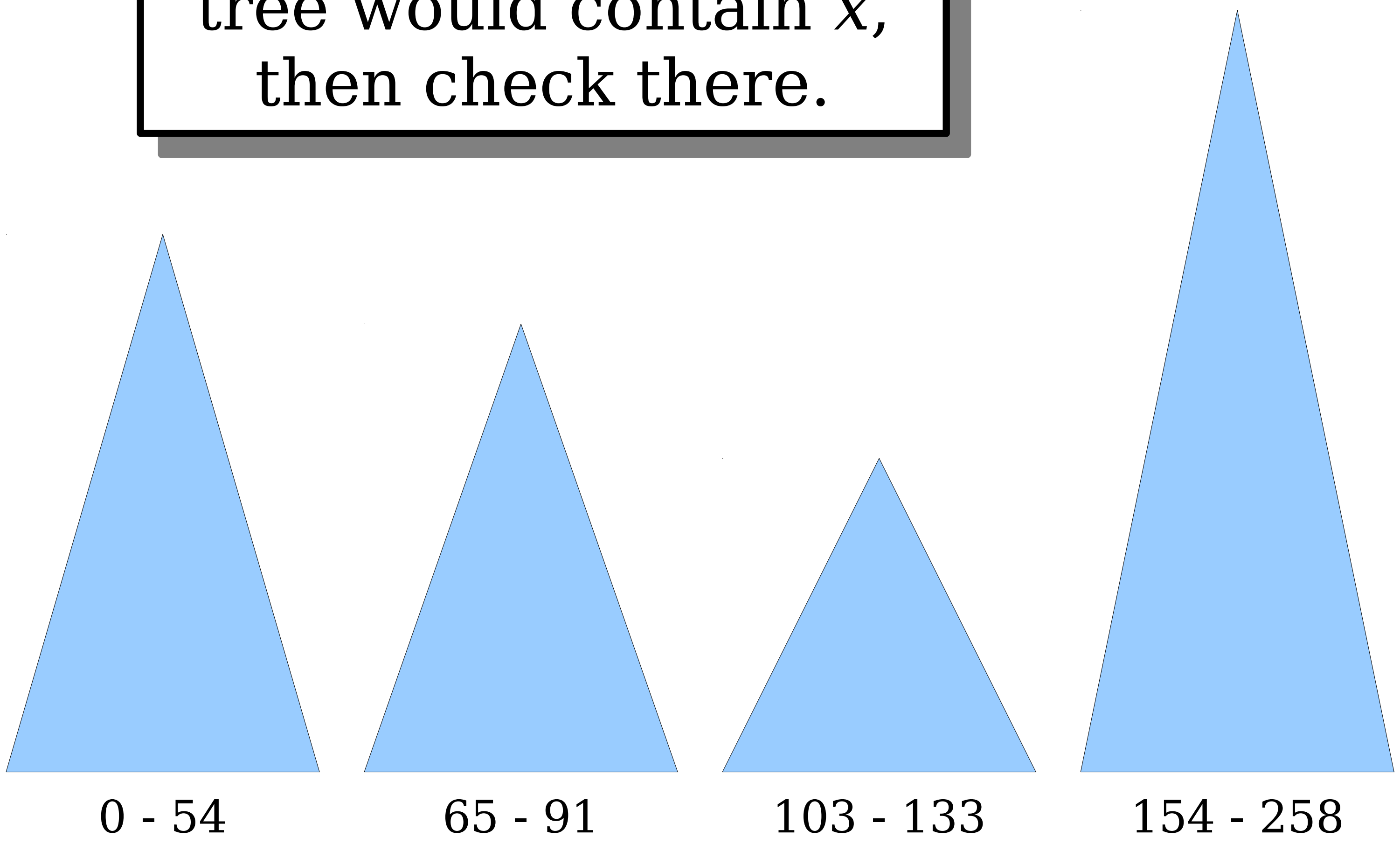
The Idea

Each of these trees has between ½ log U and 2 log U nodes.



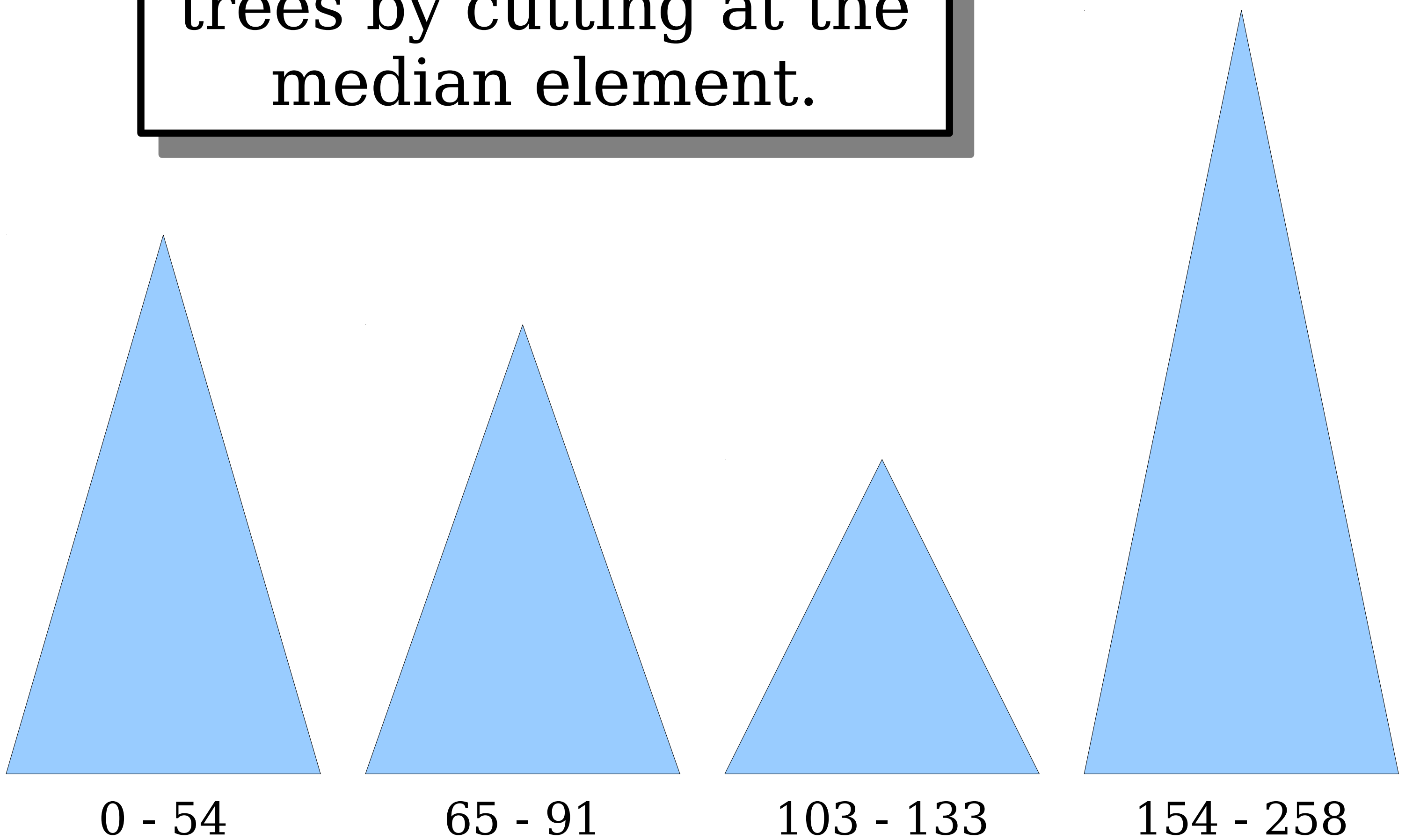
The Idea

To perform ***lookup***(x), we determine which tree would contain x, then check there.



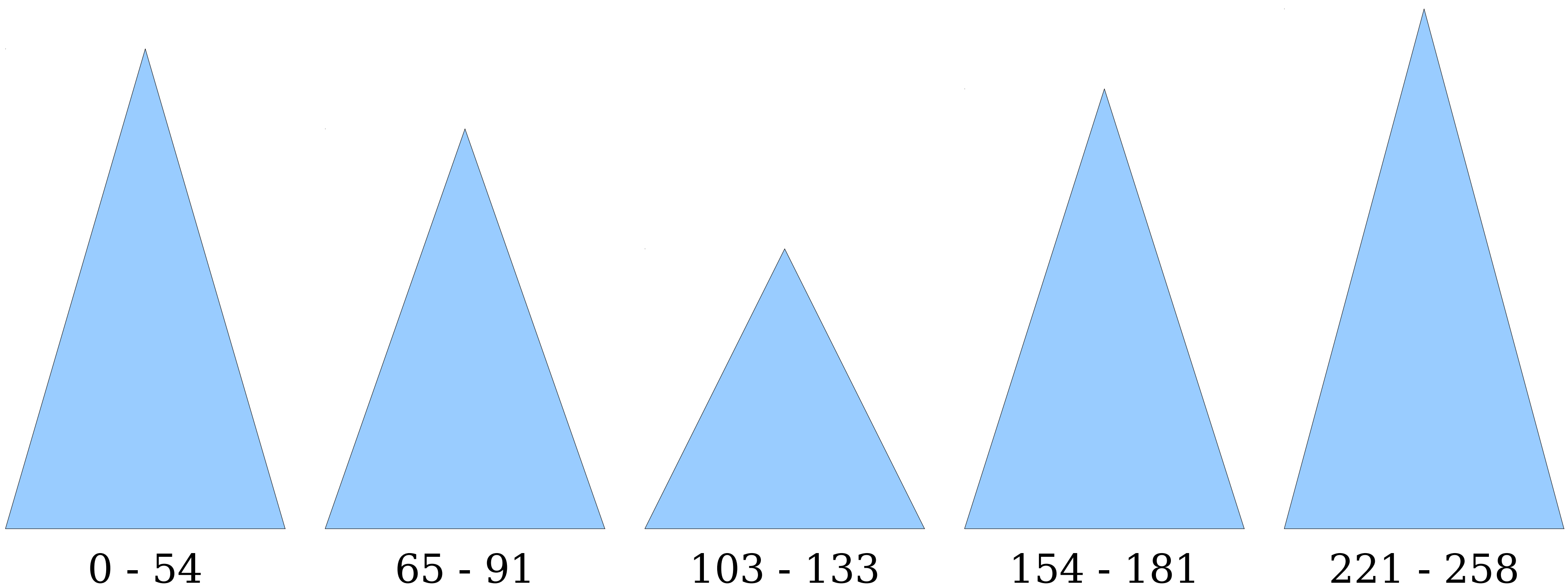
The Idea

If a tree gets too big, we can split it into two trees by cutting at the median element.



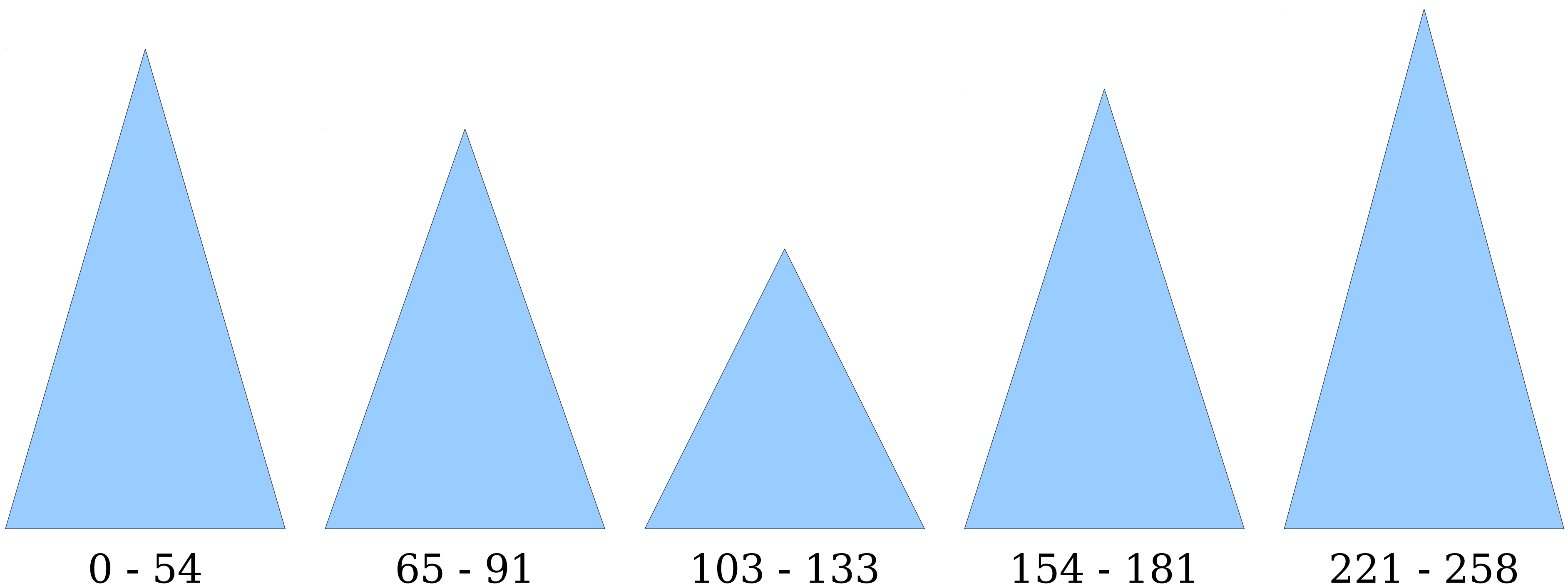
The Idea

If a tree gets too big, we can split it into two trees by cutting at the median element.



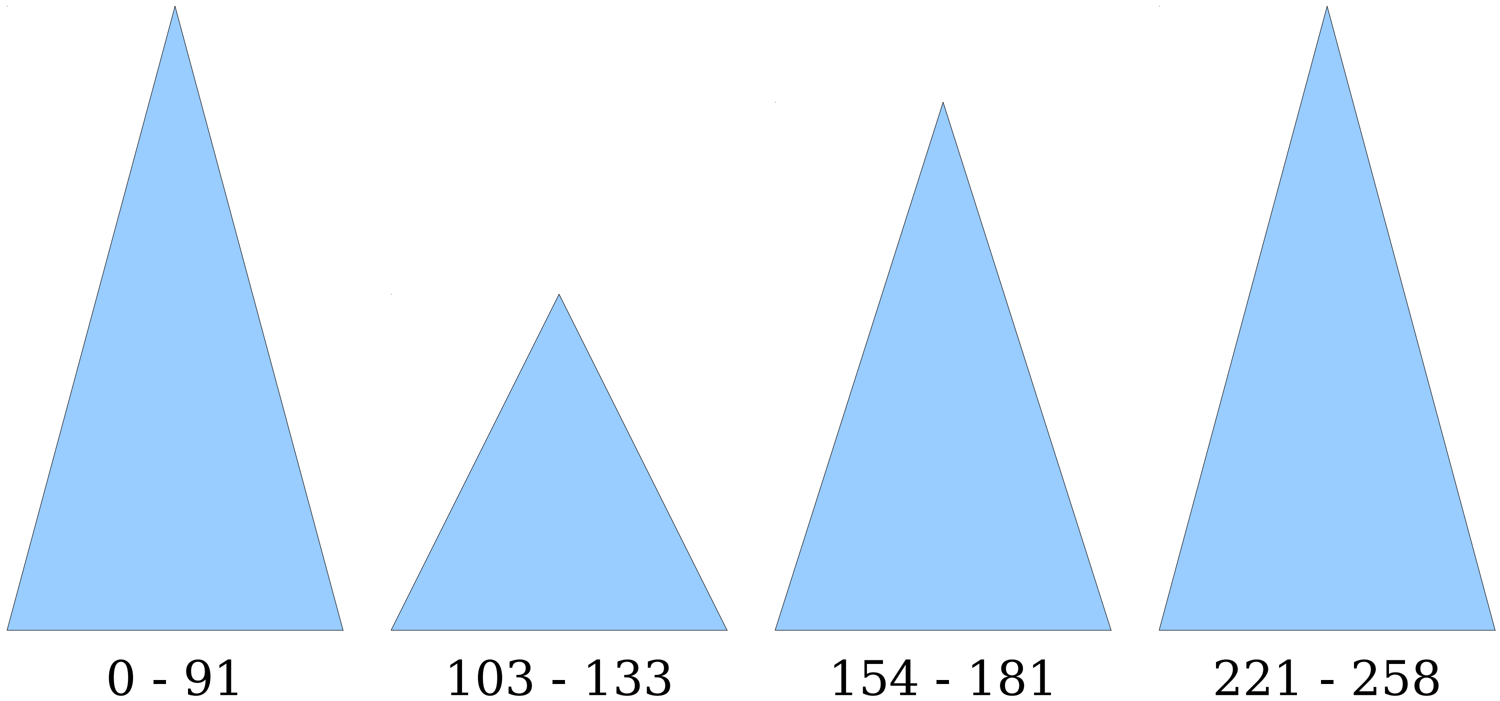
The Idea

Similarly, if trees get too small, we can concatenate the tree with a neighbor.



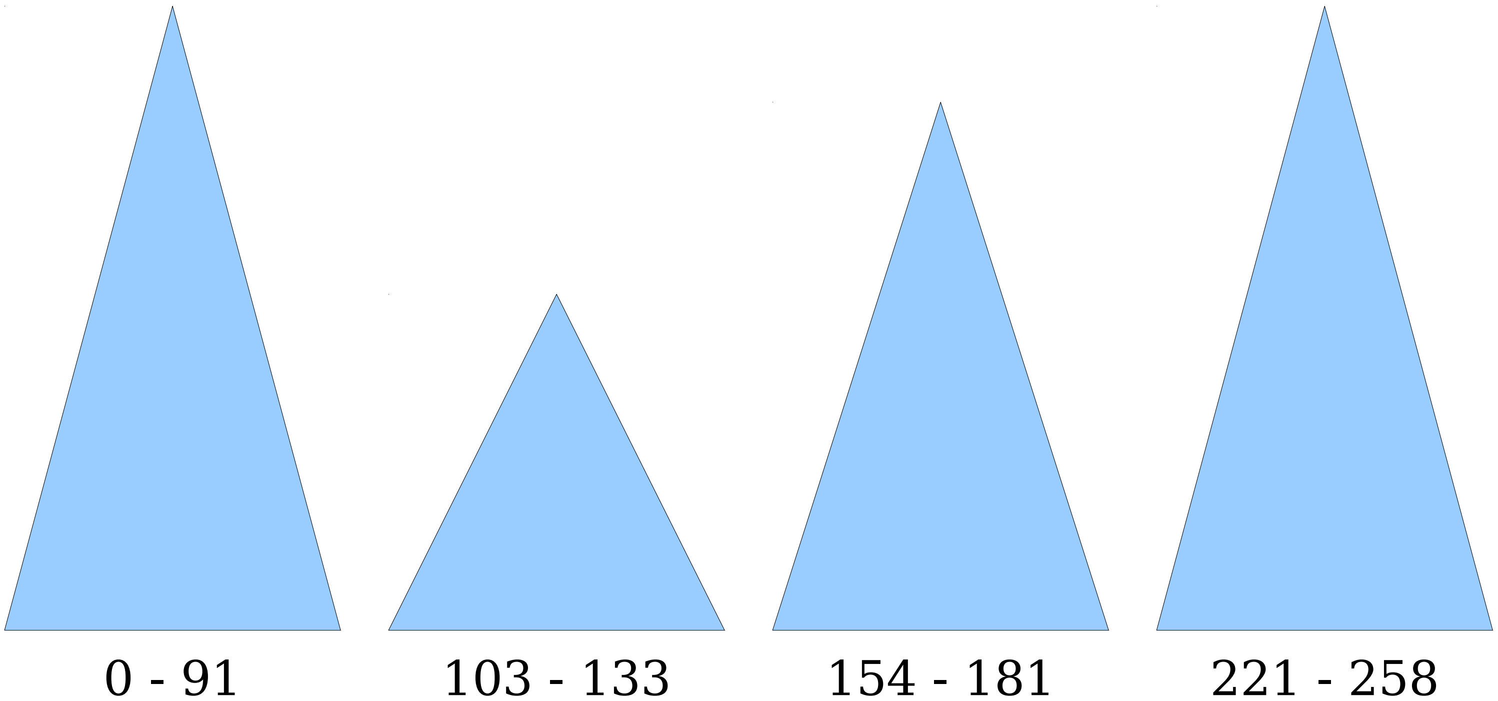
The Idea

Similarly, if trees get too small, we can concatenate the tree with a neighbor.



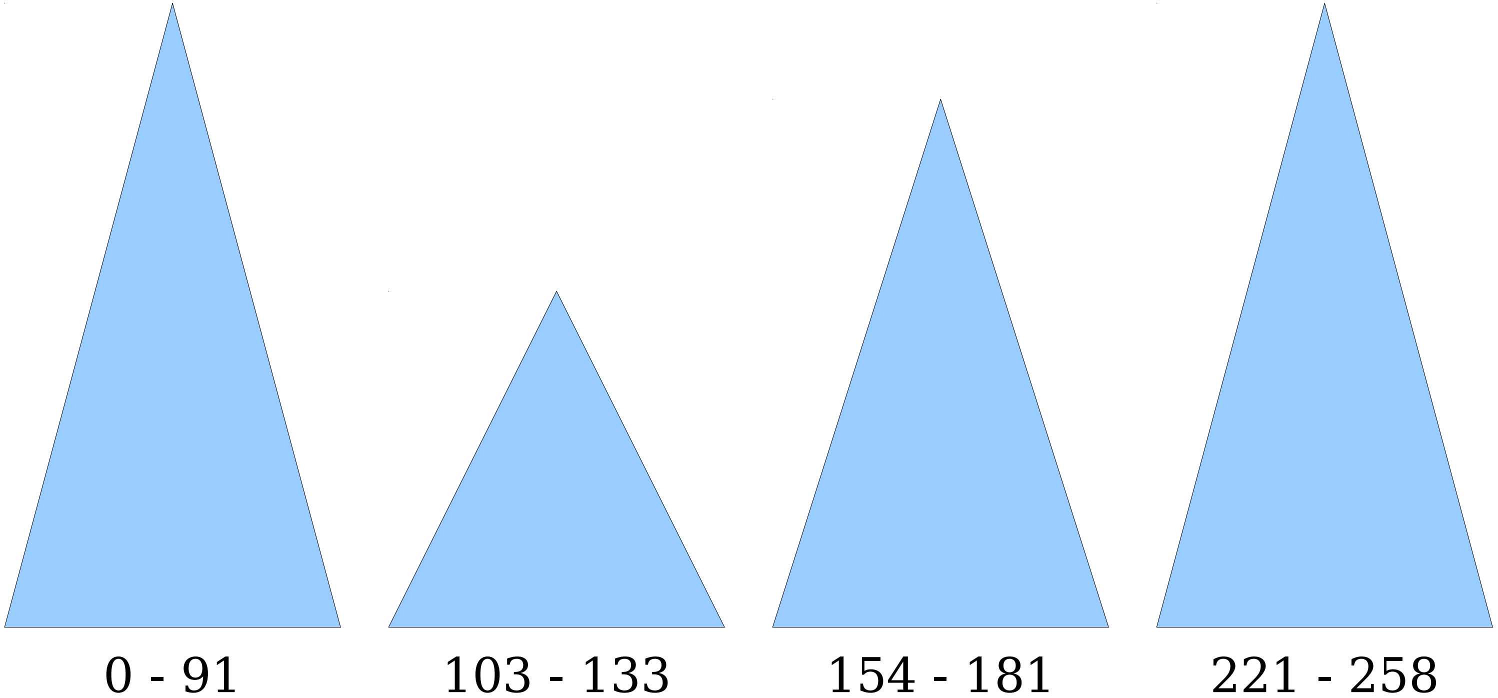
The Idea

That might create a tree that's too big, in which case we split it in half.

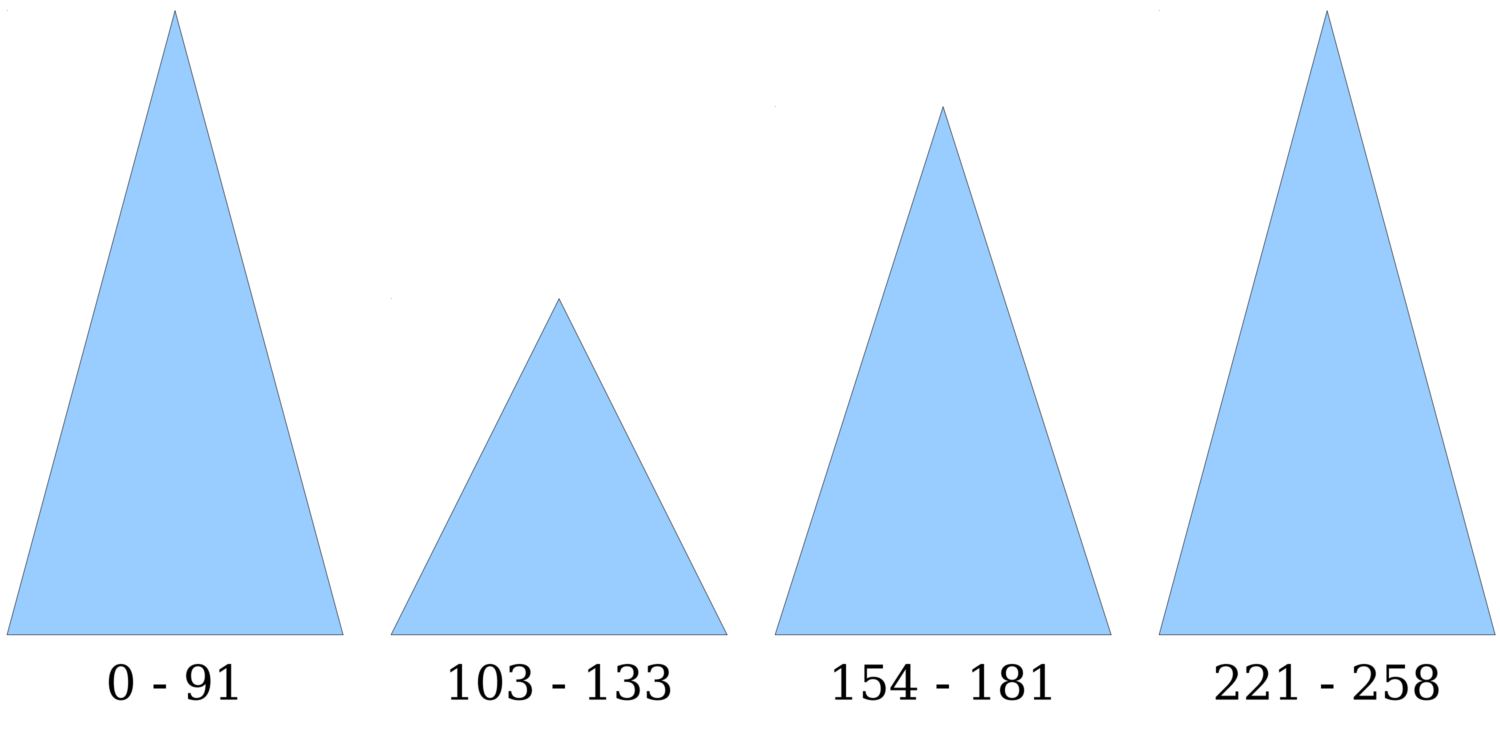


The Idea

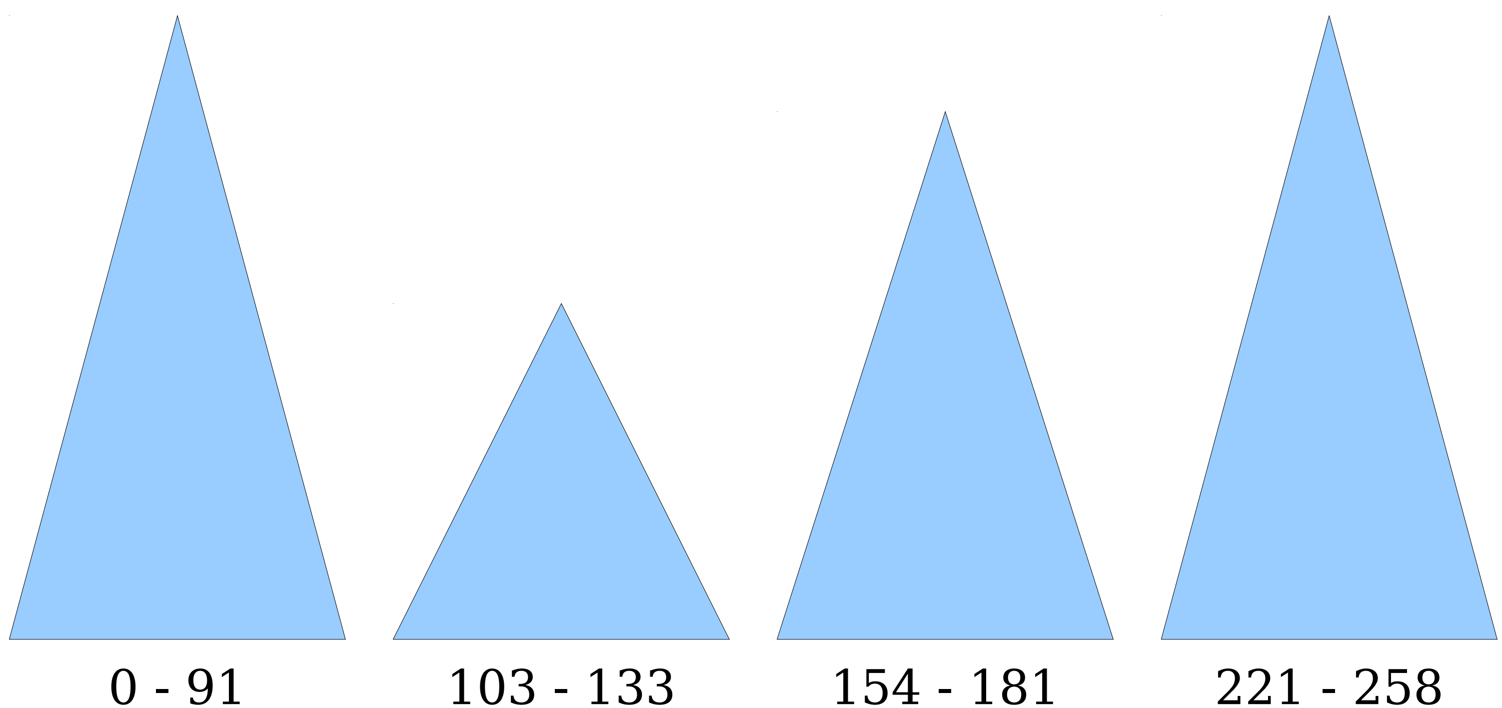
To determine ***successor***(x), we find the tree that would contain x, and take its successor there or the minimum value from the next tree.



The Idea

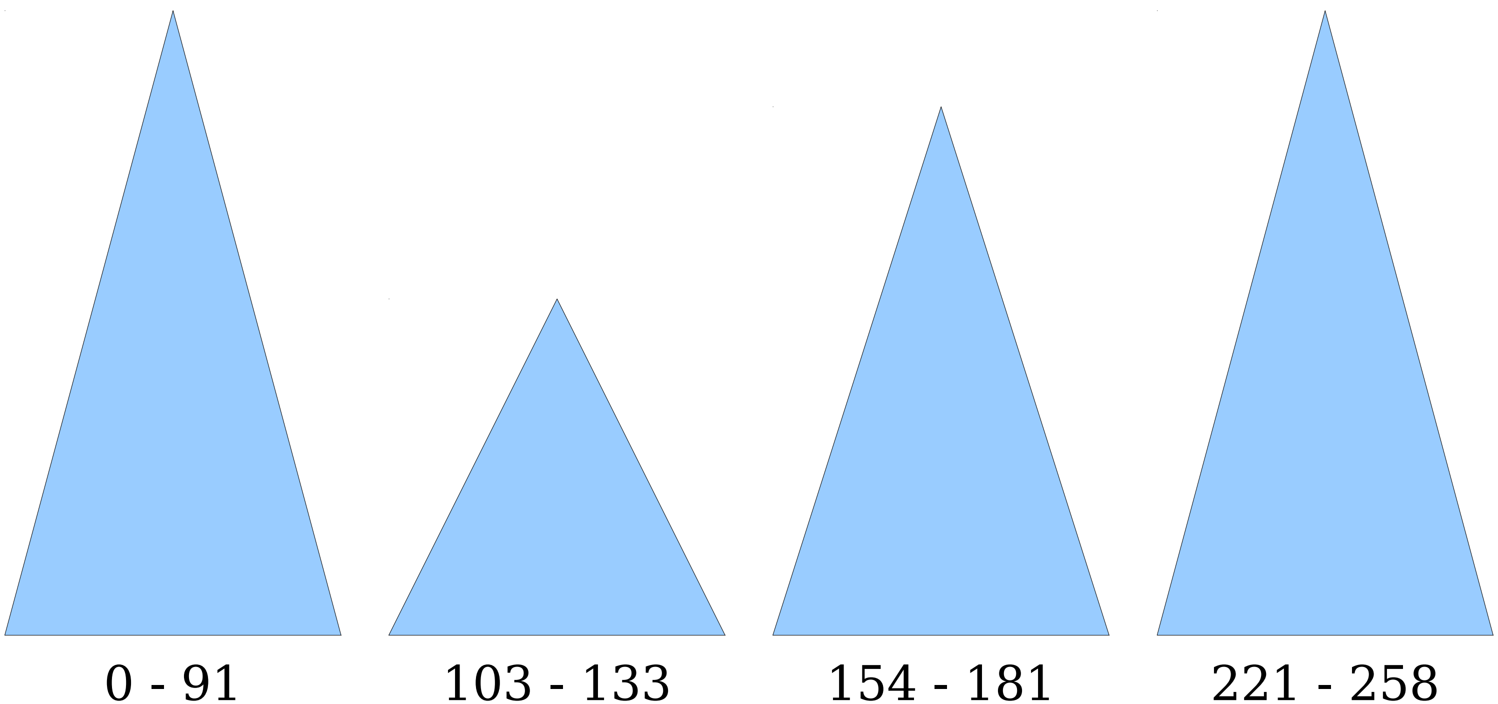


The Idea

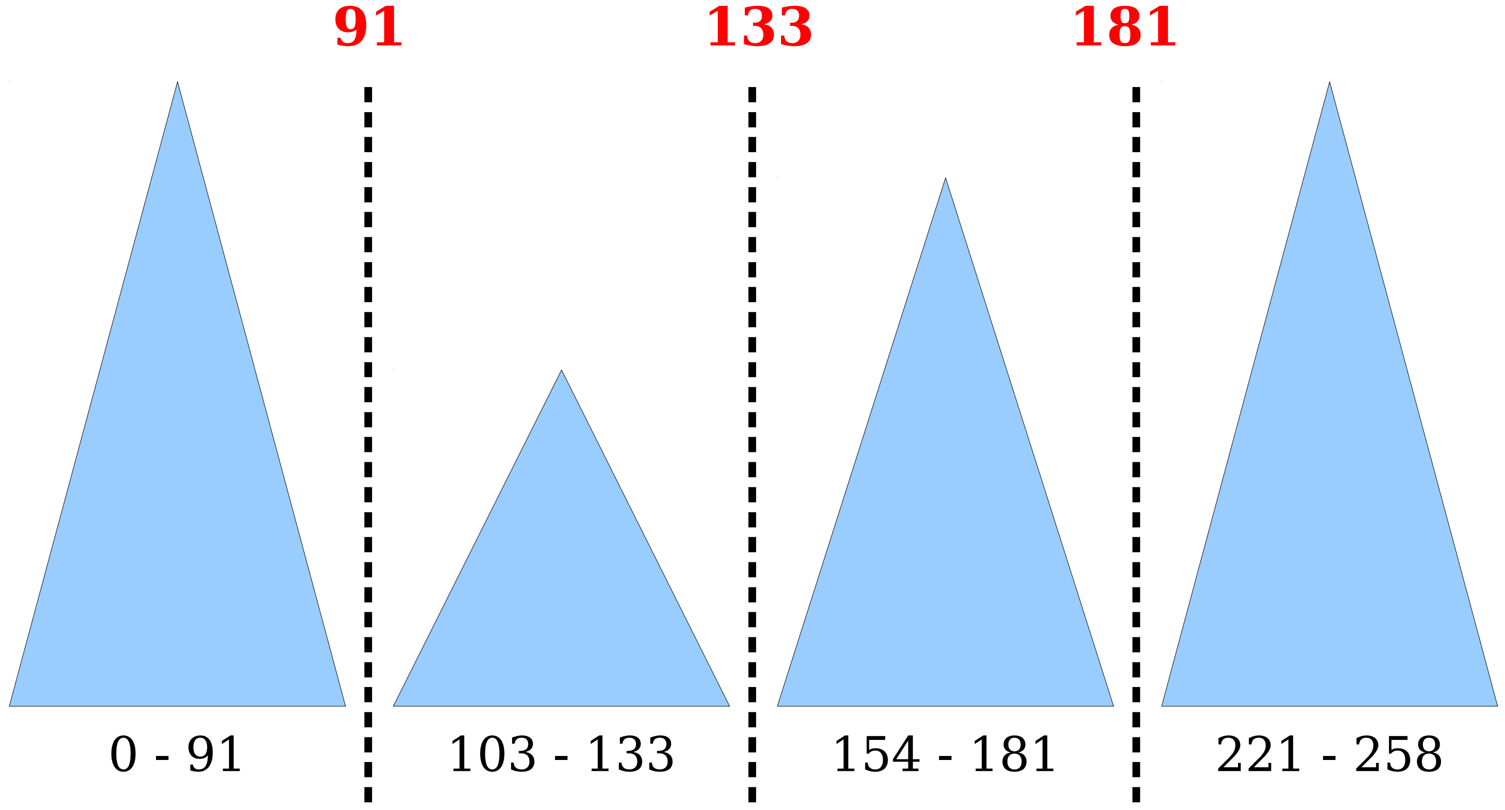


The Idea

How do we efficiently determine which tree a given element belongs to?

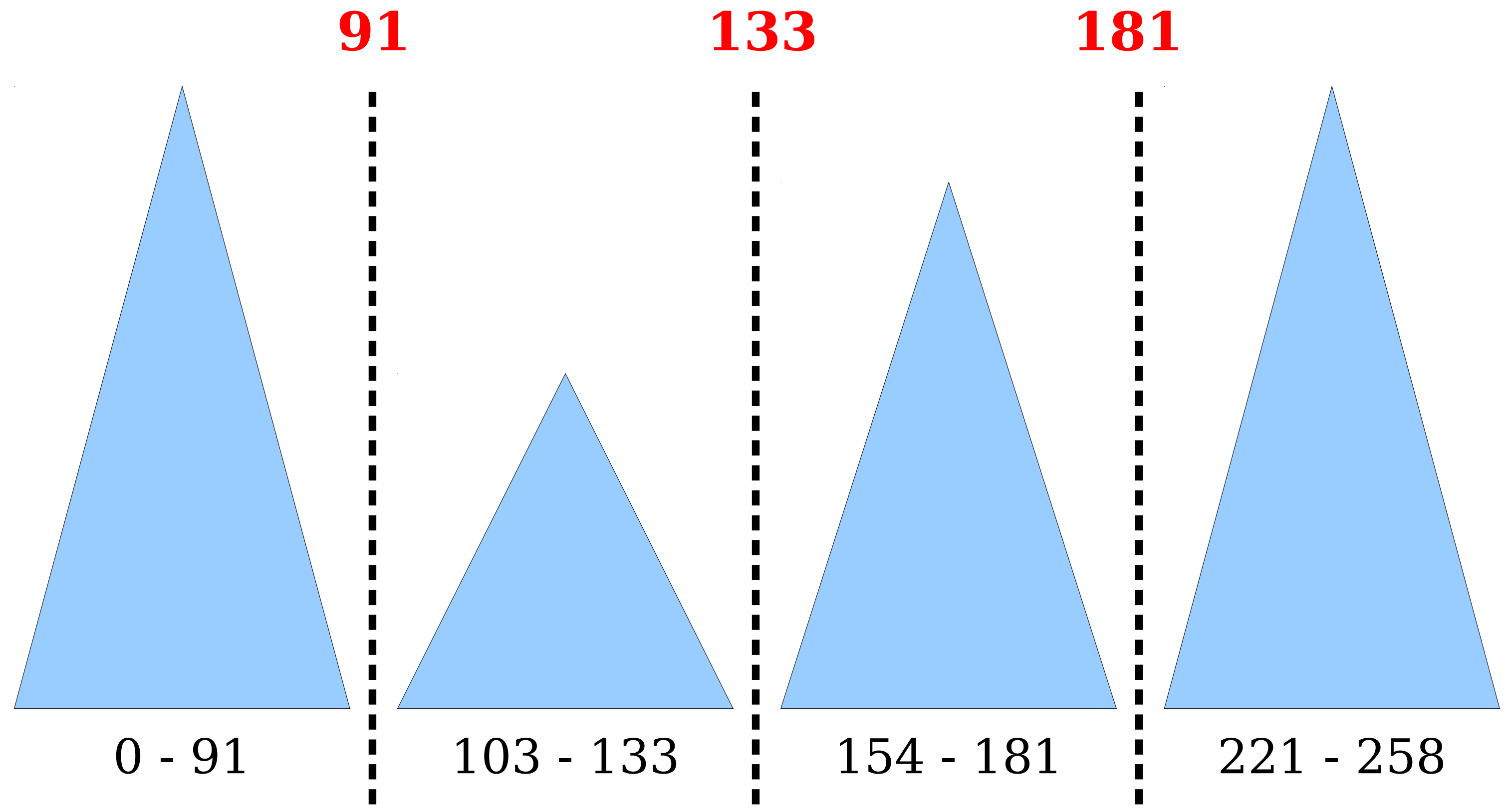


The Idea



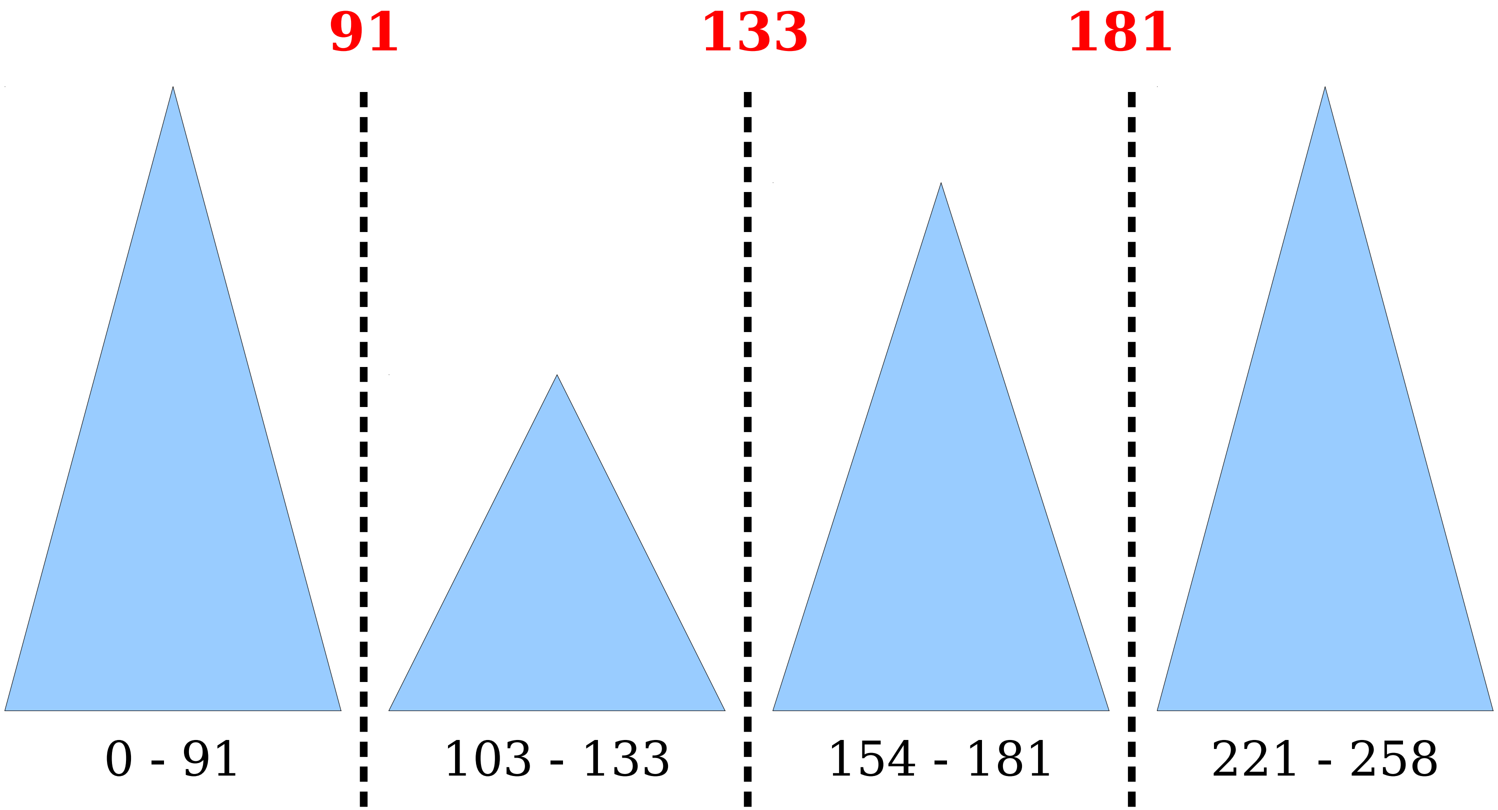
The Idea

These partition points are given by taking the maximum element in each tree at the time it's created.



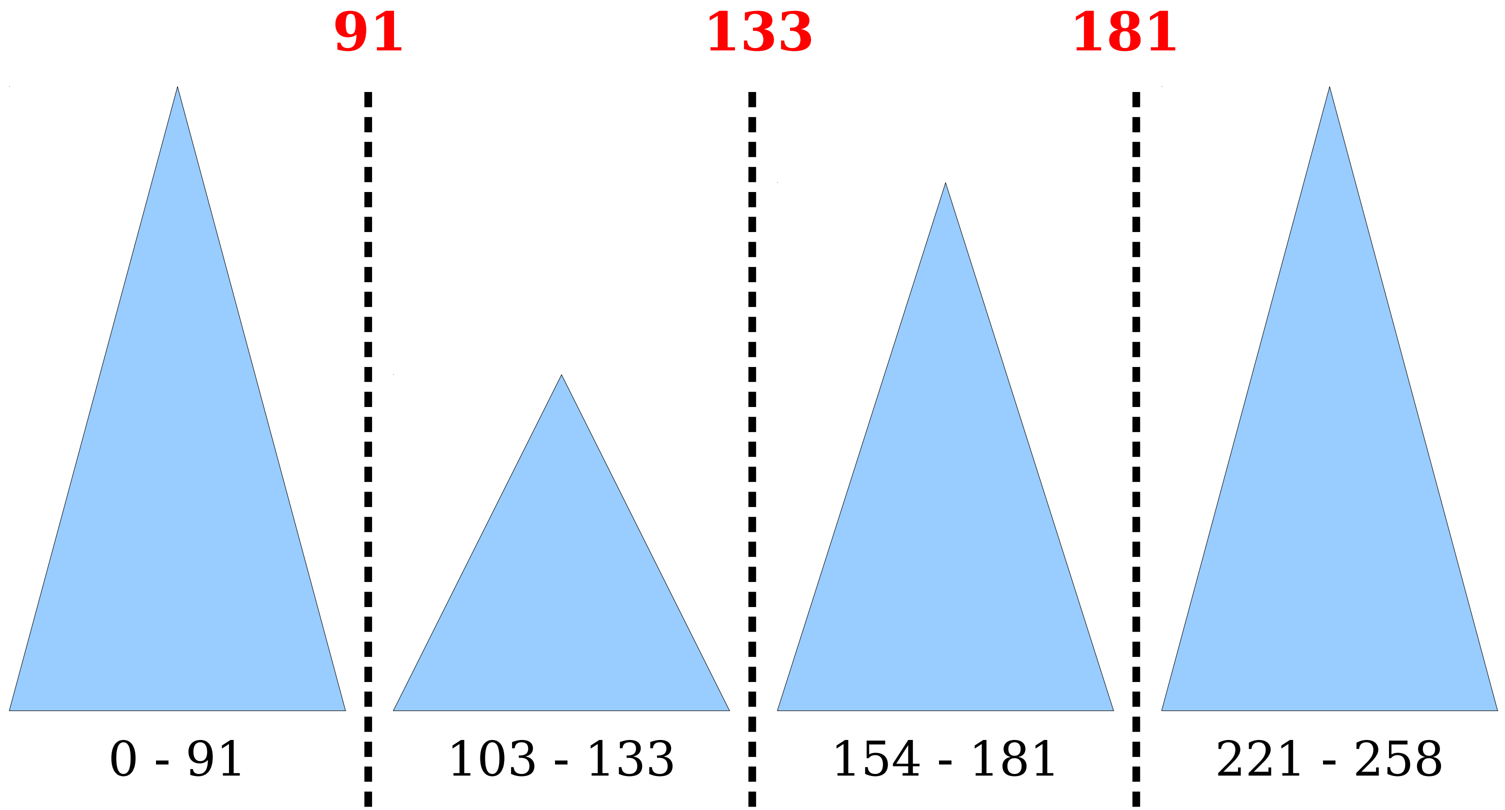
The Idea

To do ***lookup***(x), find the smallest max value that's at least x, then go into the preceding tree.



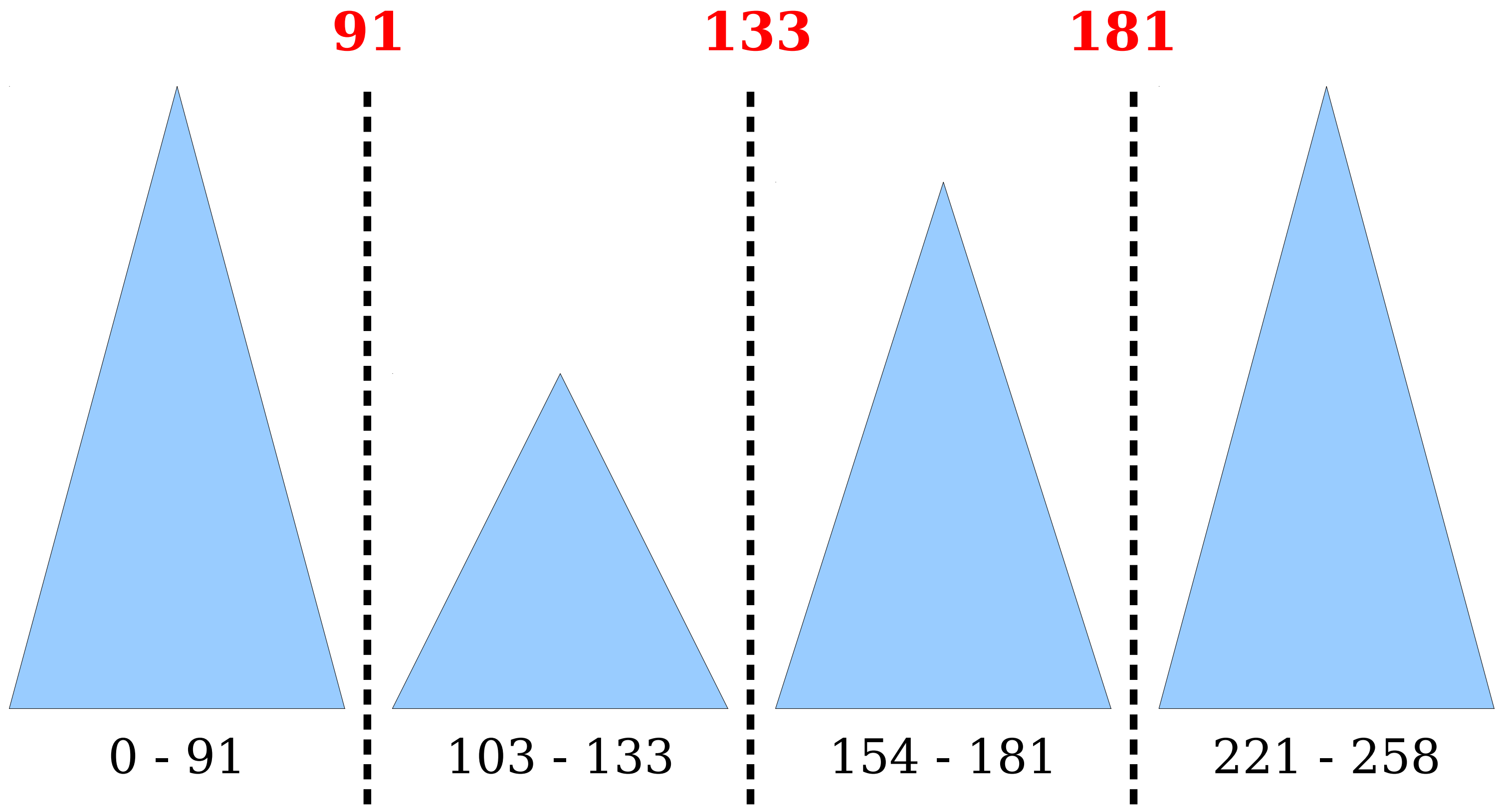
The Idea

To do ***lookup***(x), find ***successor***(x) in the set of maxes, then go into the preceding tree.



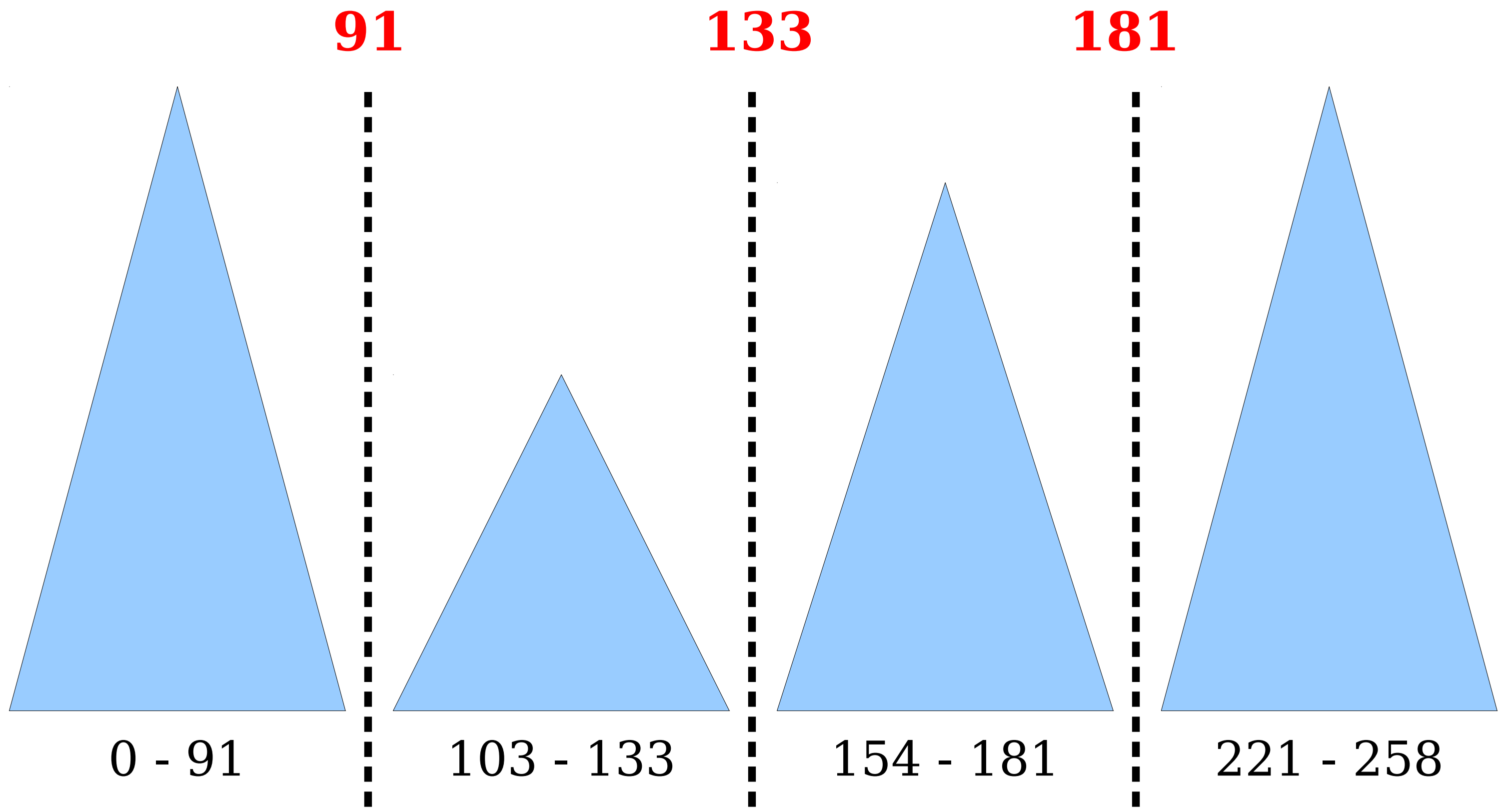
The Idea

To determine ***successor***(x), find ***successor***(x) in the maxes, then return the successor of x in that subtree or the min of the next subtree.



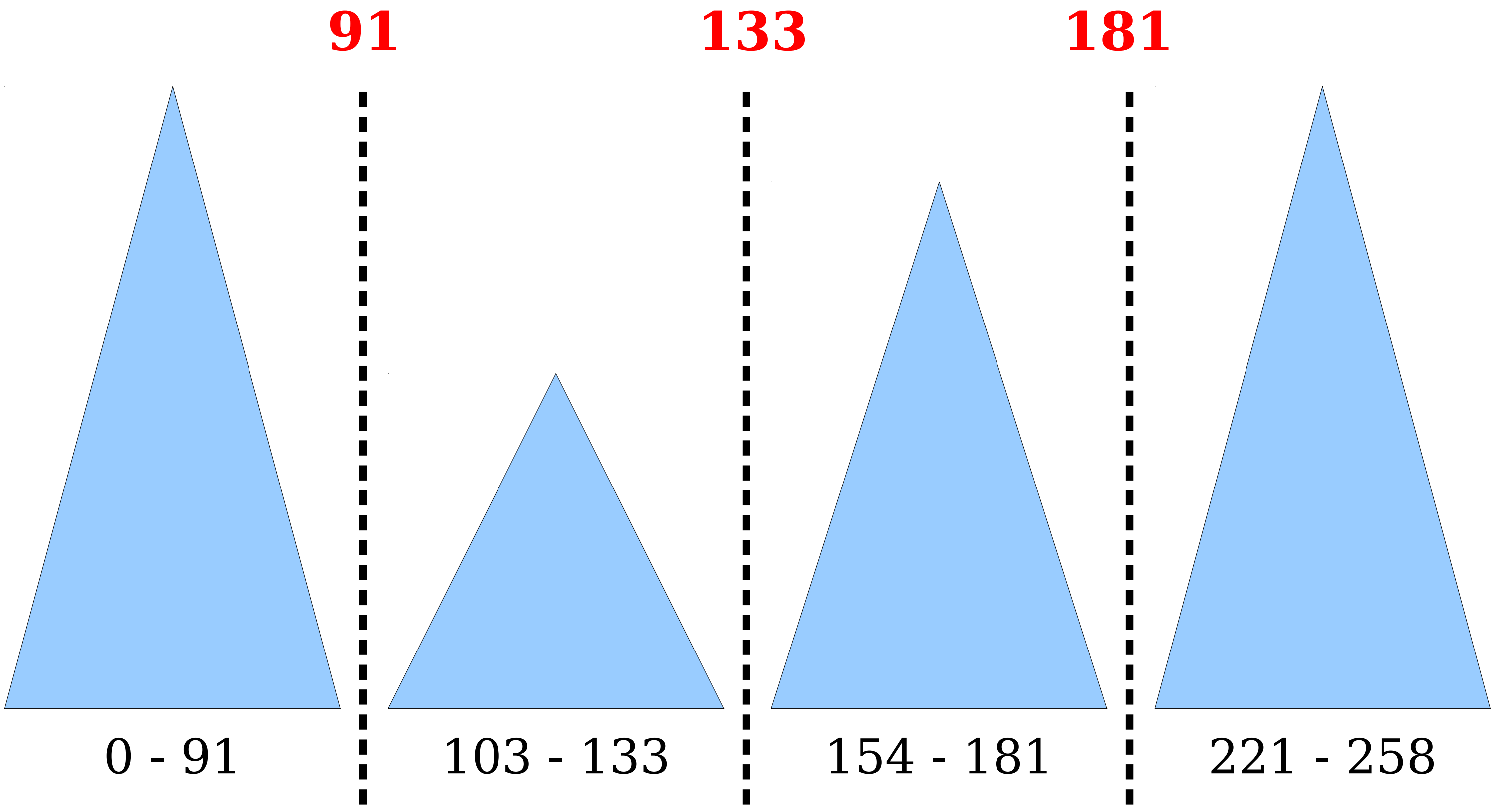
The Idea

To ***insert***(x), compute ***successor***(x) and insert x into the tree before it. If the tree splits, insert a new max into the top list.

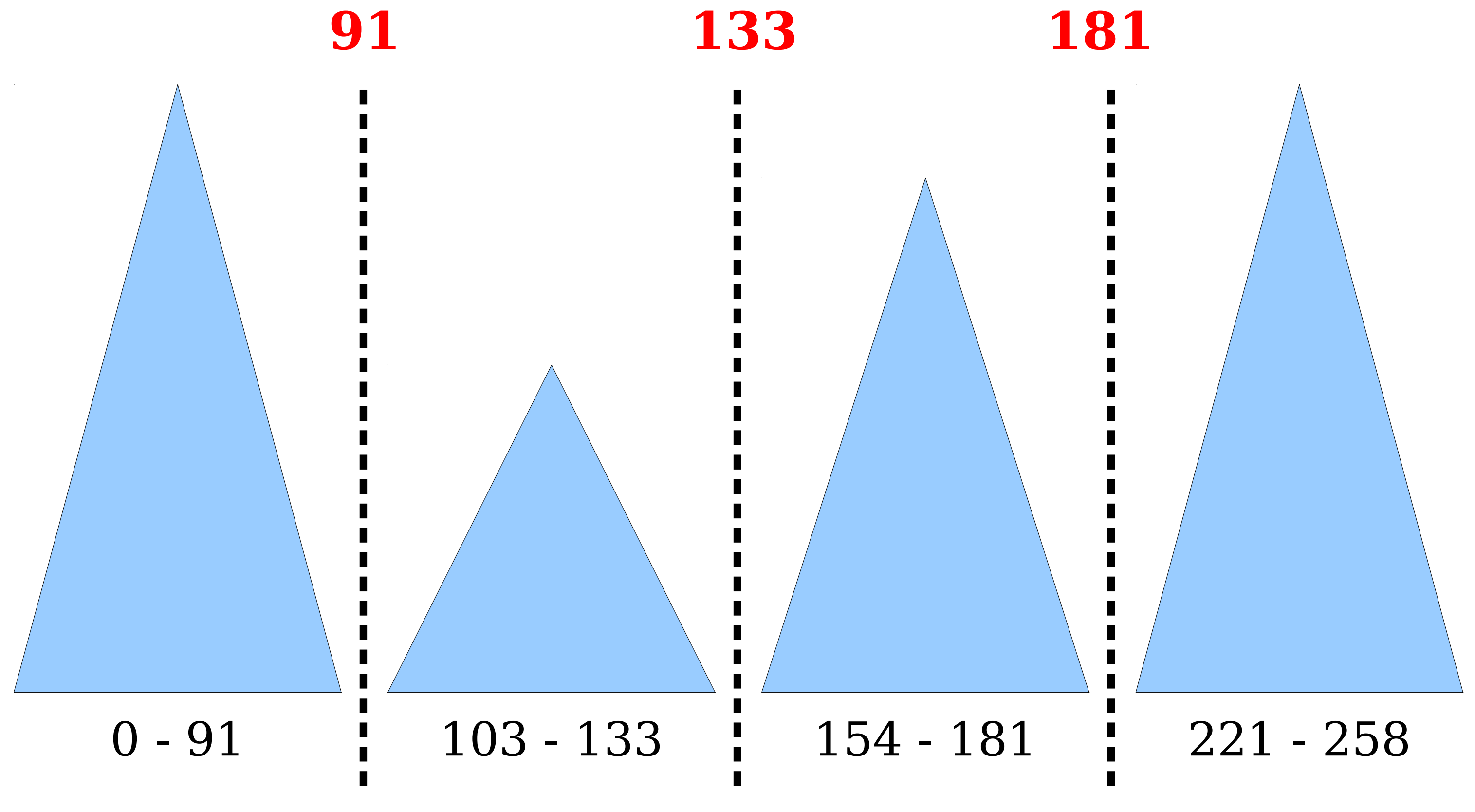


The Idea

To ***delete***(*x*), do a lookup for *x* and delete it from that tree. If *x* was the max of a tree, *don't delete it from the top list*. Contract trees if necessary.

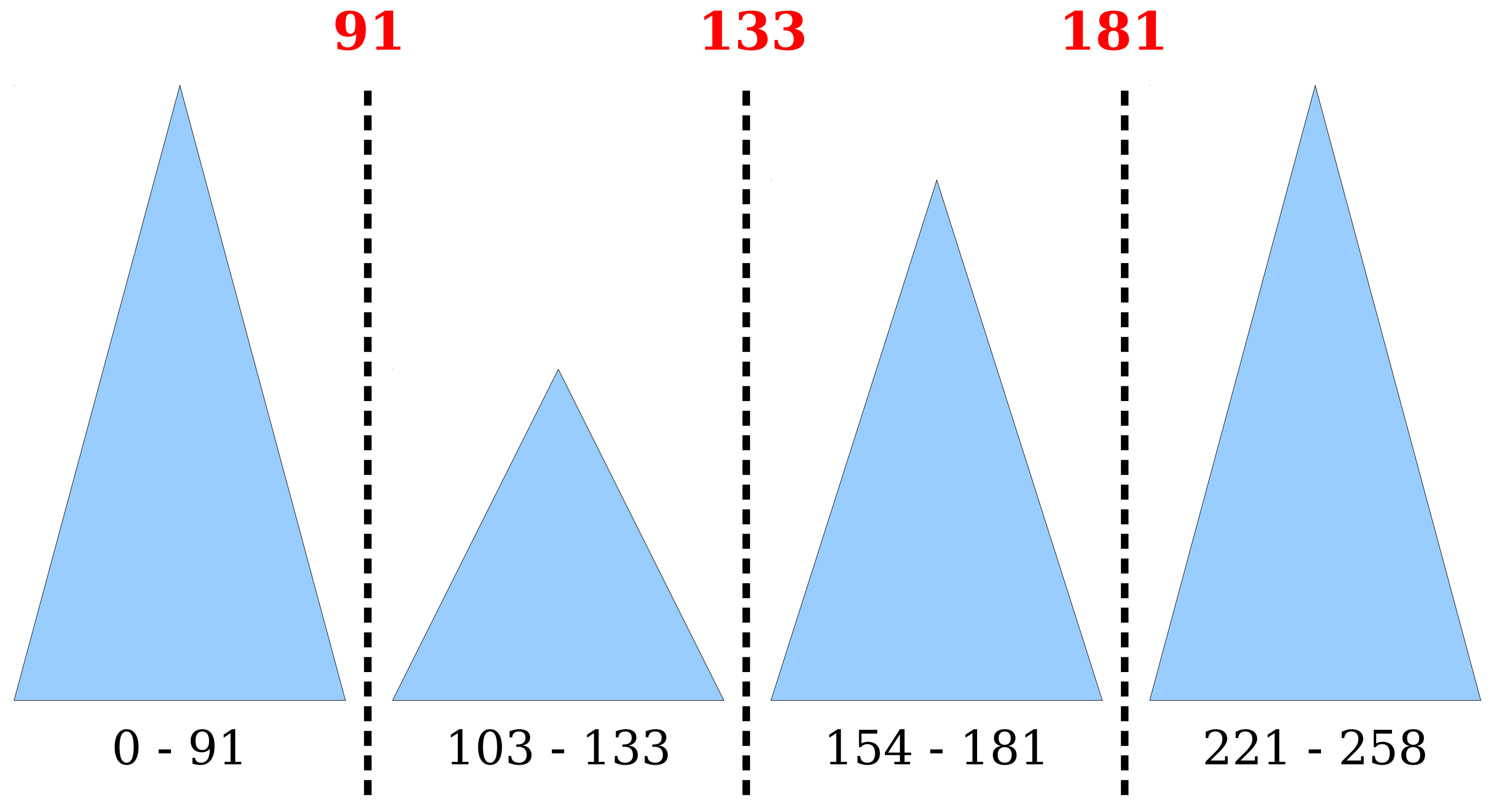


The Idea



The Idea

How do we store the set of maxes so that we get efficient ***successor*** queries?



y-Fast Tries

- A ***y-Fast Trie*** is constructed as follows:
  - Keys are stored in a collection of red/black trees, each of which has between ½ log U and 2 log U keys.
  - From each tree (except the first), choose a *representative* element.
    - Representatives demarcate the boundaries between trees.
  - Store each representative in the x-fast trie.
- Intuitively:
  - The x-fast trie helps locate which red/black trees need to be consulted for an operation.
  - Most operations are then done on red/black trees, which then take time O(log log U) each.

Analyzing *y*-Fast Tries

- The operations ***lookup***, ***successor***, ***min***, and ***max*** can all be implemented by doing O(1) BST operations and one call to ***successor*** in the *x*-fast trie.
  - Total runtime: **O(log log *U*)**.
- ***insert*** and ***delete*** do O(1) BST operations, but also have to do O(1) insertions or deletions into the *x*-fast trie.
  - Total runtime: **O(log *U*)**.
  - … or is it?

Analyzing *y*-Fast Tries

- Each insertion does O(log log U) work inserting and (potentially) splitting a red/black tree.
- The insertion in the *x*-fast trie takes time O(log U).
- However, we only split a red/black tree if its size doubles from log U to 2 log U, so we must have done at least O(log U) insertions before we needed to split.
- The extra cost amortizes across those operations to O(1), so the *amortized* cost of an insertion is **O(log log U)**.

Analyzing *y*-Fast Tries

- Each deletion does O(log log U) work deleting from, (potentially) joining a red/black tree, and (potentially) splitting the resulting red/black tree.
- The insertions and deletions in the *x*-fast trie take time at most O(log U).
- However, we only join a tree with its neighbor if its size dropped from log U to ½ log U, which means there were O(log U) intervening deletions.
- The extra cost amortizes across those operations to O(1), so the *amortized* cost of an insertion is **O(log log U)**.

Space Usage

- So what about space usage?
- Total space used across all the red/black trees is $O(n)$.
- The x-fast trie stores $\Theta(n / \log U)$ total elements.
- Space usage:

$$\Theta((n / \log U) \cdot \log U) = \boldsymbol{\Theta(n)}.$$

- We're back down to linear space!

For Reference

- van Emde Boas tree
  - ***insert***: O(log log U)
  - ***delete***: O(log log U)
  - ***lookup***: O(log log U)
  - ***max***: O(1)
  - ***succ***: O(log log U)
  - ***is-empty***: O(1)
  - Space: O(U)
- y-Fast Trie
  - ***insert***: O(log log U)*
  - ***delete***: O(log log U)*
  - ***lookup***: O(log log U)
  - ***max***: O(log log U)
  - ***succ***: O(log log U)
  - ***is-empty***: O(1)
  - Space: O(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, augmented BSTs.
- *y*-fast tries also use the “blocking” technique from RMQ we used to shave off log factors.

Next Time

- ***Disjoint-Set Forests***
  - A data structure for incremental connectivity in general graphs.
- ***The Ackermann Inverse Function***
  - One of the slowest-growing functions you'll ever encounter in practice.