x-Fast and *y*-Fast Tries

Giulio Ermanno Pibiri
Ca' Foscari University of Venice

Venice, Italy, 10 February 2026

Problem definition

- Maintain a **sorted integer set** $S \subseteq \{0,\ldots,U-1\}$, for some **universe size** U, under the following operations and queries for an integer $0 \leq x < U$.

  - Min(): return the smallest element in S.
  - Max(): return the largest element in S.
  - Member(x): return "Yes" if $x \in S$, "No" otherwise.
  - Insert(x): add x to S (assuming $x \notin S$).
  - Delete(x): remove x from S (assuming $x \in S$).
  - Successor(x): return the smallest $y \in S$ such that $y > x$, or $\perp$ if no such element exists.
  - Predecessor(x): return the largest $y \in S$ such that $y < x$, or $\perp$ if no such element exists.

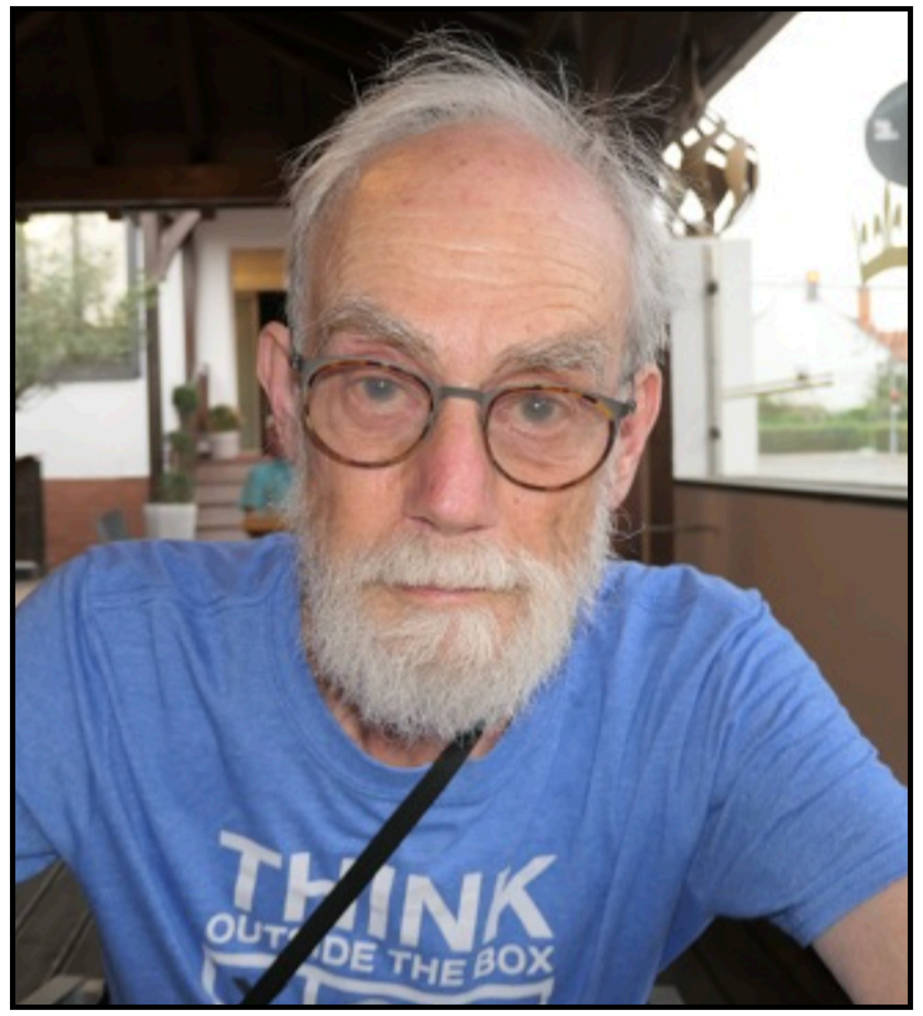
- Let n be $|S|$ in the following.

Summary from last week

- **van Emde Boas** trees (1975-77) solve the problem in $O(\log \log U)$ worst-case time and space $\Theta(U)$.

- When $U = n^c$ for some $c \geq 1$ then $\log_2 \log_2 U = \Theta(\log \log n)$ and vEB trees are **exponentially** faster than AVL and RB trees.
  This is **optimal** for Successor/Predecessor.

- **Key insight.** The $\log_2 U$-bit representation of an integer can be split recursively to speed up operations and queries.



Peter van Emde Boas

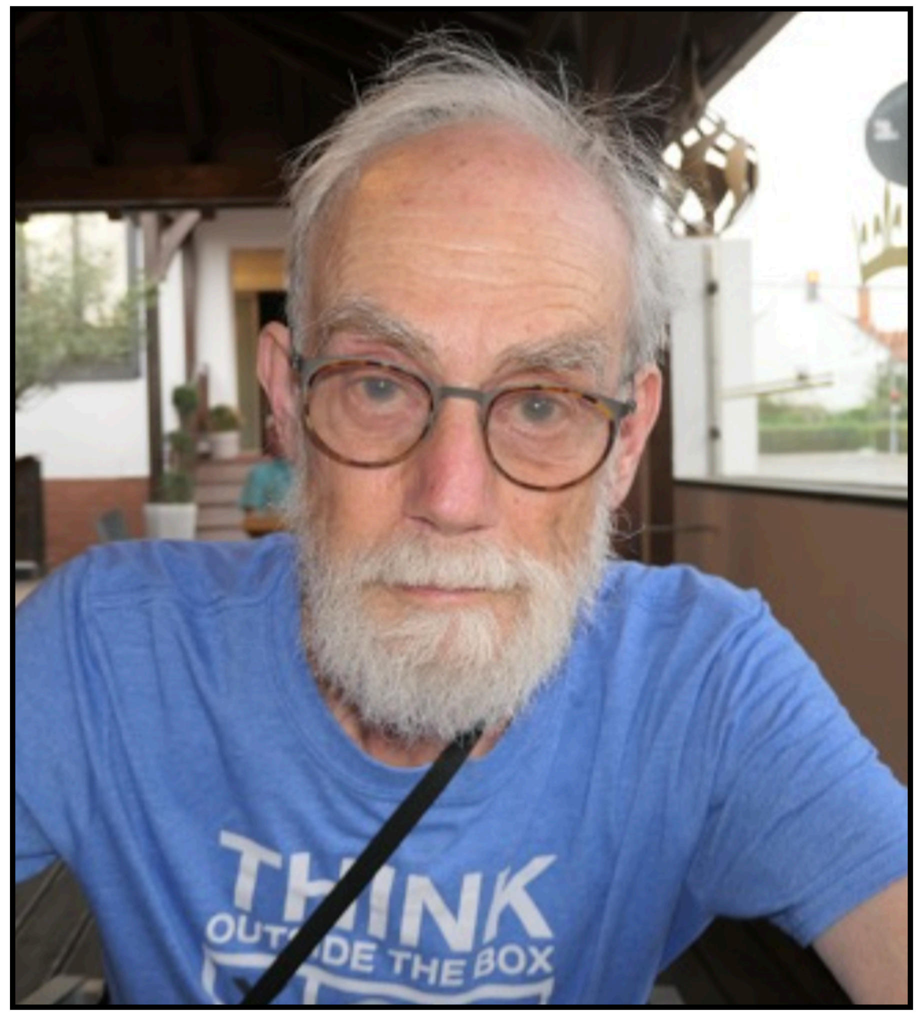
Summary from last week

- **van Emde Boas** trees (1975-77) solve the problem in $O(\log \log U)$ worst-case time and space $\Theta(U)$.

- When $U = n^c$ for some $c \geq 1$ then $\log_2 \log_2 U = \Theta(\log \log n)$ and vEB trees are **exponentially** faster than AVL and RB trees.
  This is **optimal** for Successor/Predecessor.

- **Key insight.** The $\log_2 U$-bit representation of an integer can be split recursively to speed up operations and queries.

- **Q.** Is it possible combine $O(\log \log U)$ time with space $\Theta(n)$?

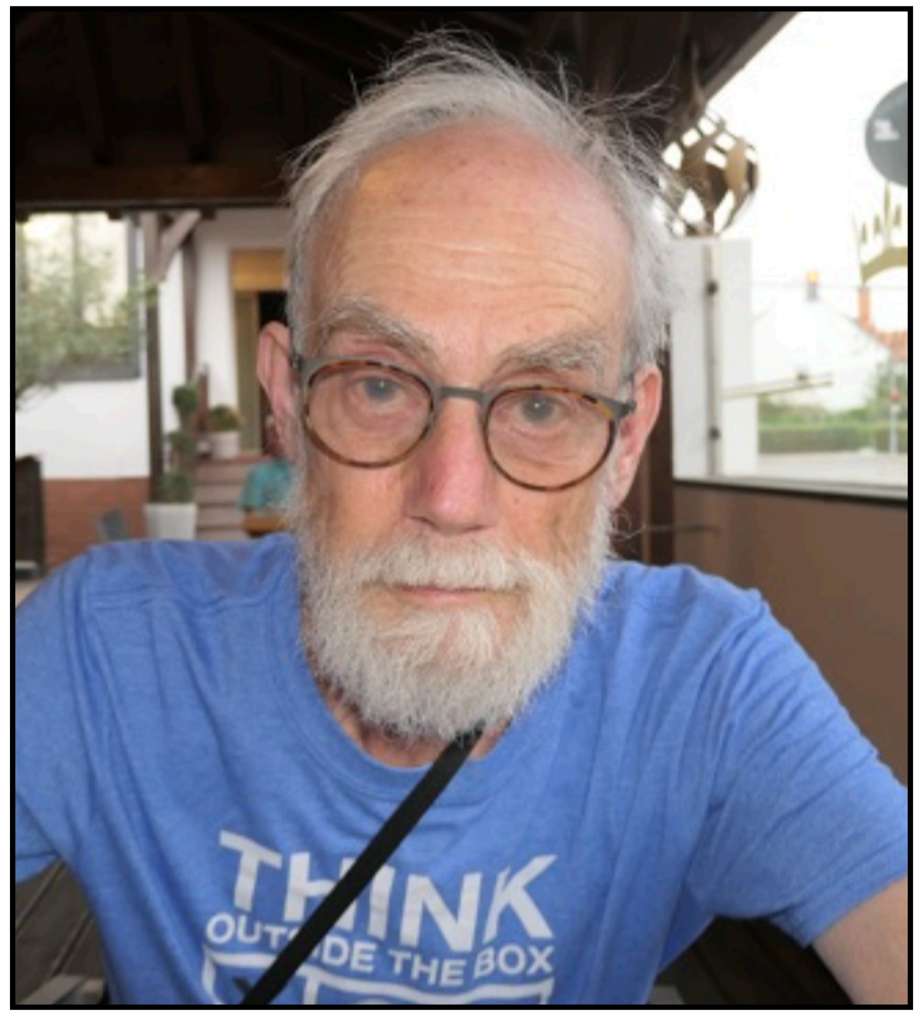


Peter van Emde Boas

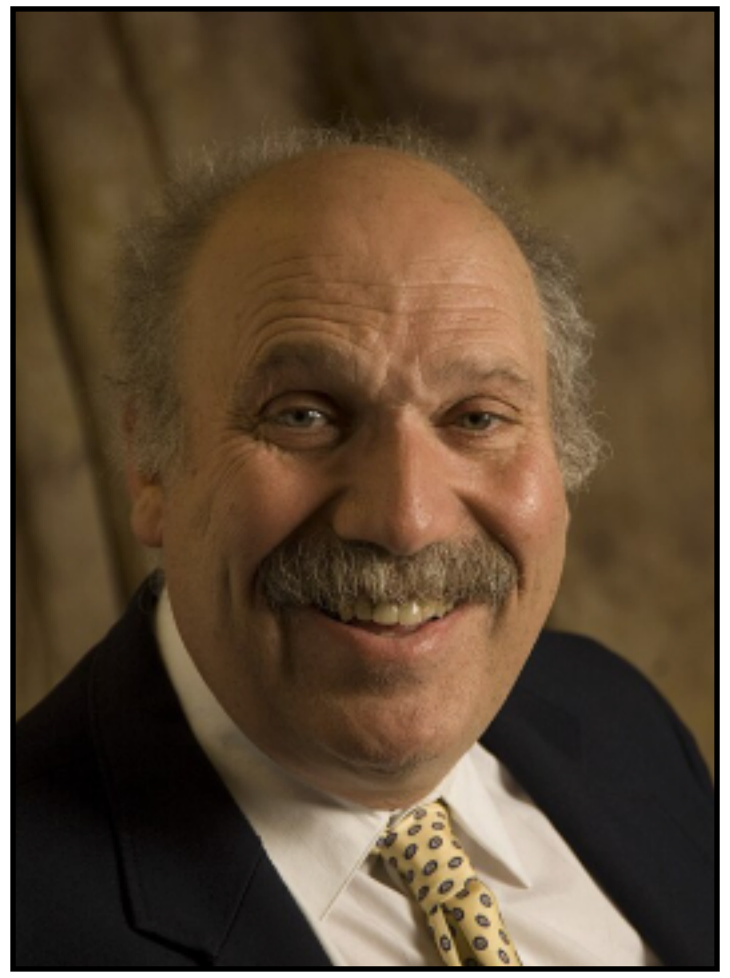
Summary from last week

- **van Emde Boas** trees (1975-77) solve the problem in $O(\log \log U)$ worst-case time and space $\Theta(U)$.
- When $U = n^c$ for some $c \geq 1$ then $\log_2 \log_2 U = \Theta(\log \log n)$ and vEB trees are **exponentially** faster than AVL and RB trees.
  This is **optimal** for Successor/Predecessor.
- **Key insight.** The $\log_2 U$-bit representation of an integer can be split recursively to speed up operations and queries.

- **Q.** Is it possible combine $O(\log \log U)$ time with space $\Theta(n)$?
- **A.** **Dan Willard** showed it is in 1983, with the *y*-fast trie.



Peter van Emde Boas

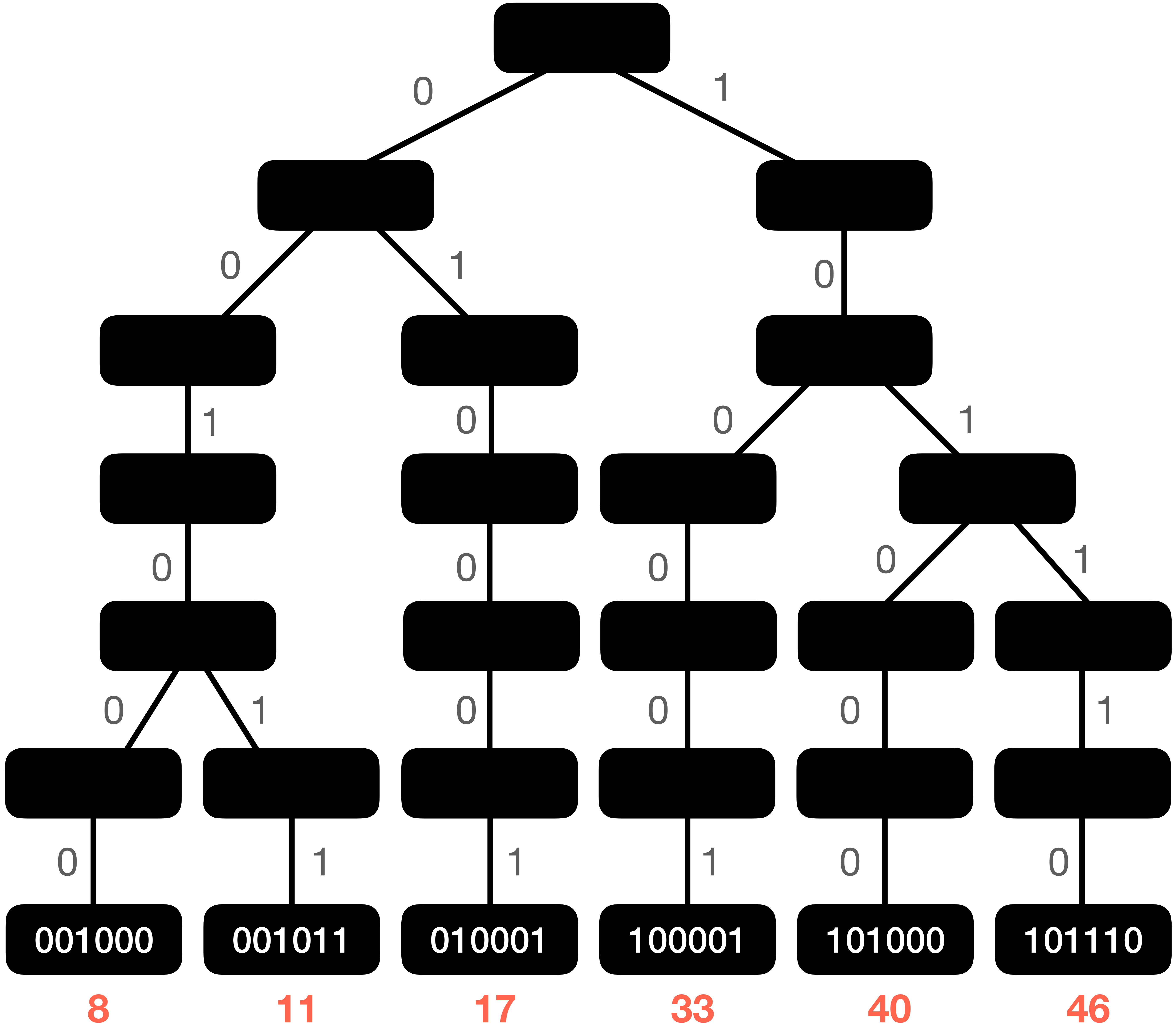


Dan Willard

Bitwise tries for integers

- **Tries** are data structures to represent a collection of strings, where common prefixes are represented once.
- Integers are binary strings of length $\log_2 U$, hence the trie has $\log_2 U$ levels.
- All operations in $O(\log U)$ time.
- For example: Successor(x) and Predecessor(x) by following the bits of x and “walking backwards” if necessary.

$S = \{8, 11, 17, 33, 40, 46\}$, $U = 64$

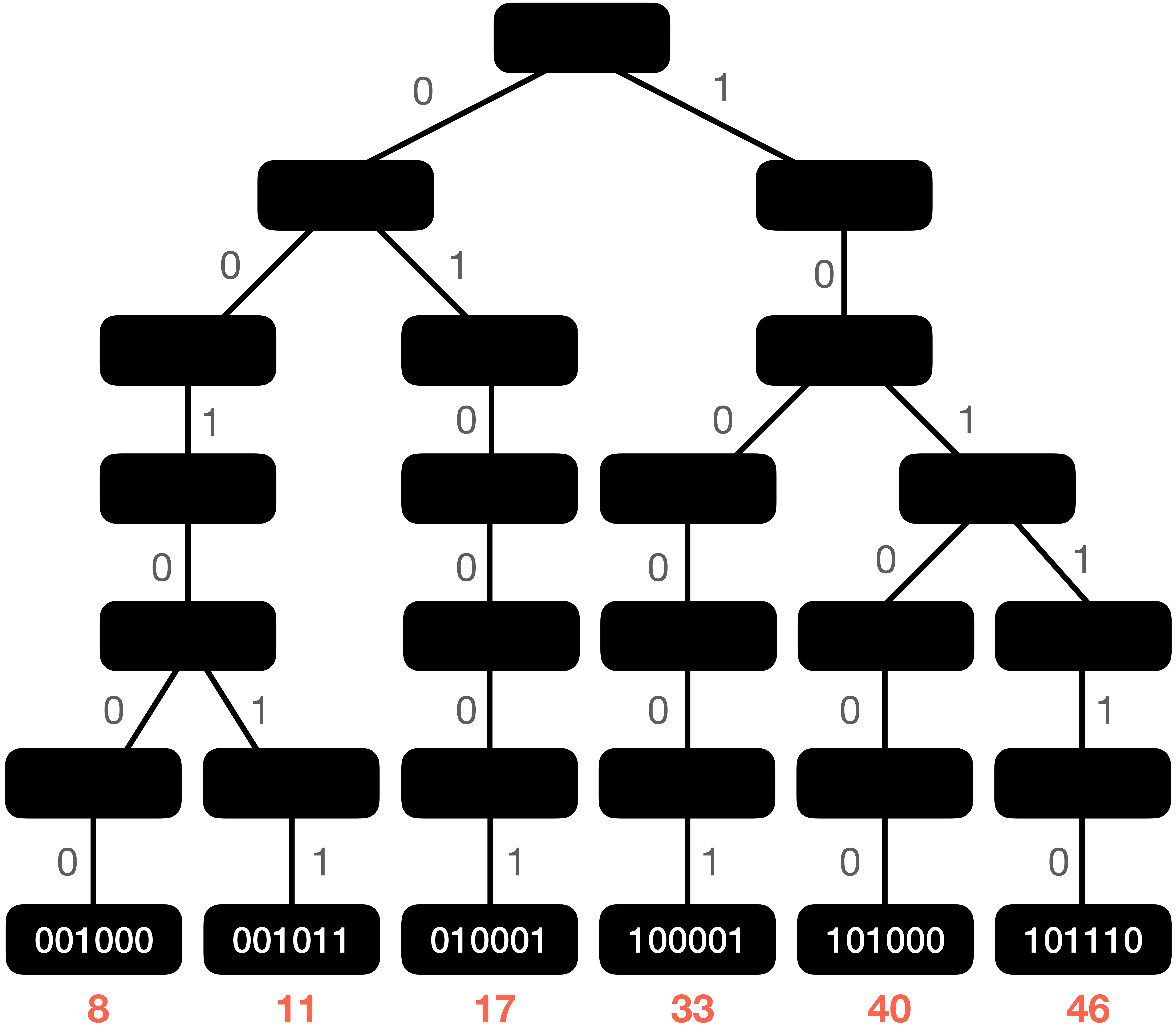


Bitwise tries for integers

$S = \{8,11,17,33,40,46\}$, $U = 64$

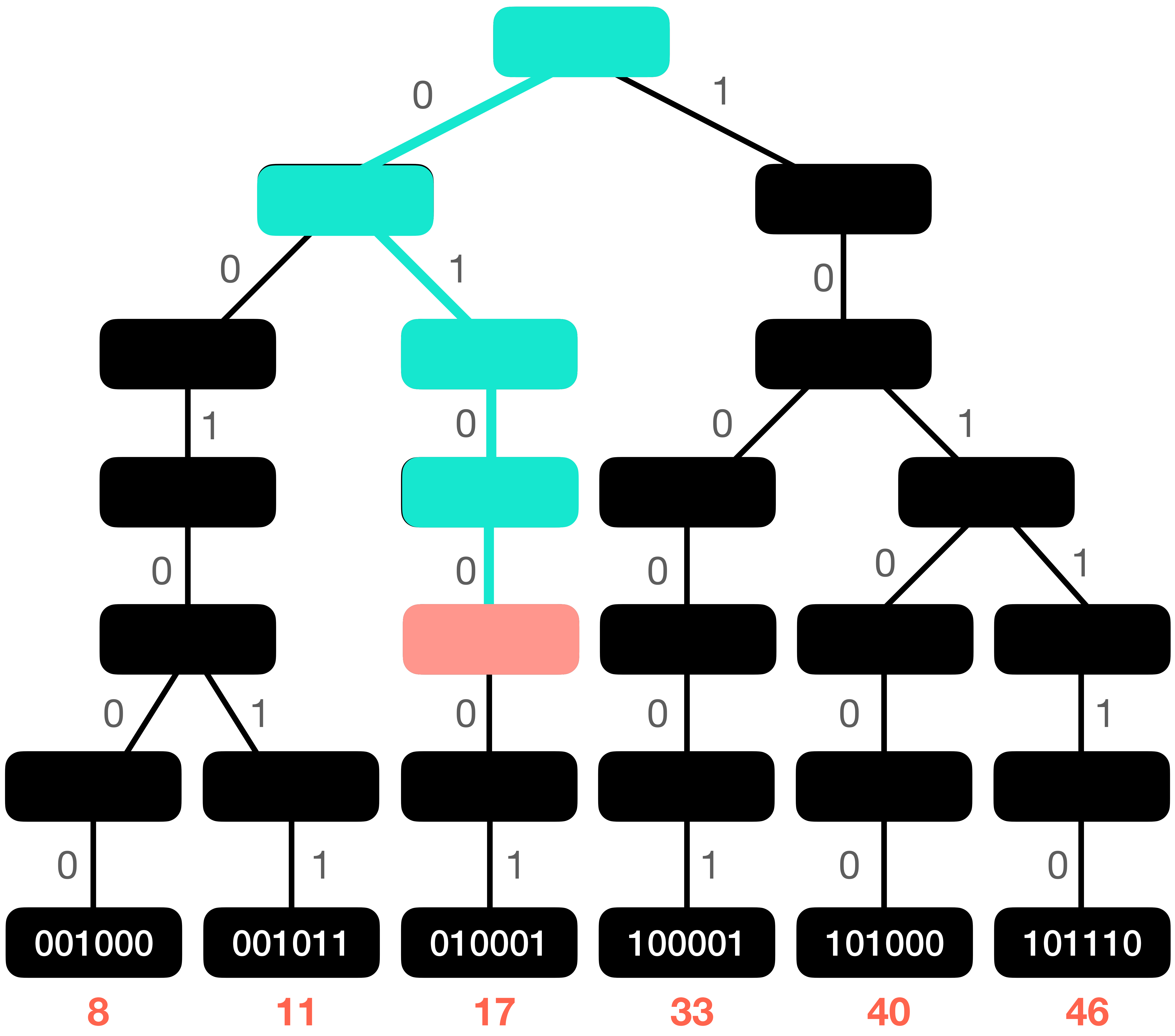
- **Tries** are data structures to represent a collection of strings, where common prefixes are represented once.
- Integers are binary strings of length $\log_2 U$, hence the trie has $\log_2 U$ levels.
- All operations in $O(\log U)$ time.
- For example: Successor(x) and Predecessor(x) by following the bits of x and "walking backwards" if necessary.

Member(19), $[19]_2 = 010011$



Bitwise tries for integers

$S = \{8,11,17,33,40,46\}$, $U = 64$



- **Tries** are data structures to represent a collection of strings, where common prefixes are represented once.
- Integers are binary strings of length $\log_2 U$, hence the trie has $\log_2 U$ levels.
- All operations in $O(\log U)$ time.
- For example: Successor(x) and Predecessor(x) by following the bits of x and "walking backwards" if necessary.

Member(19), $[19]_2 = \underline{010}011$

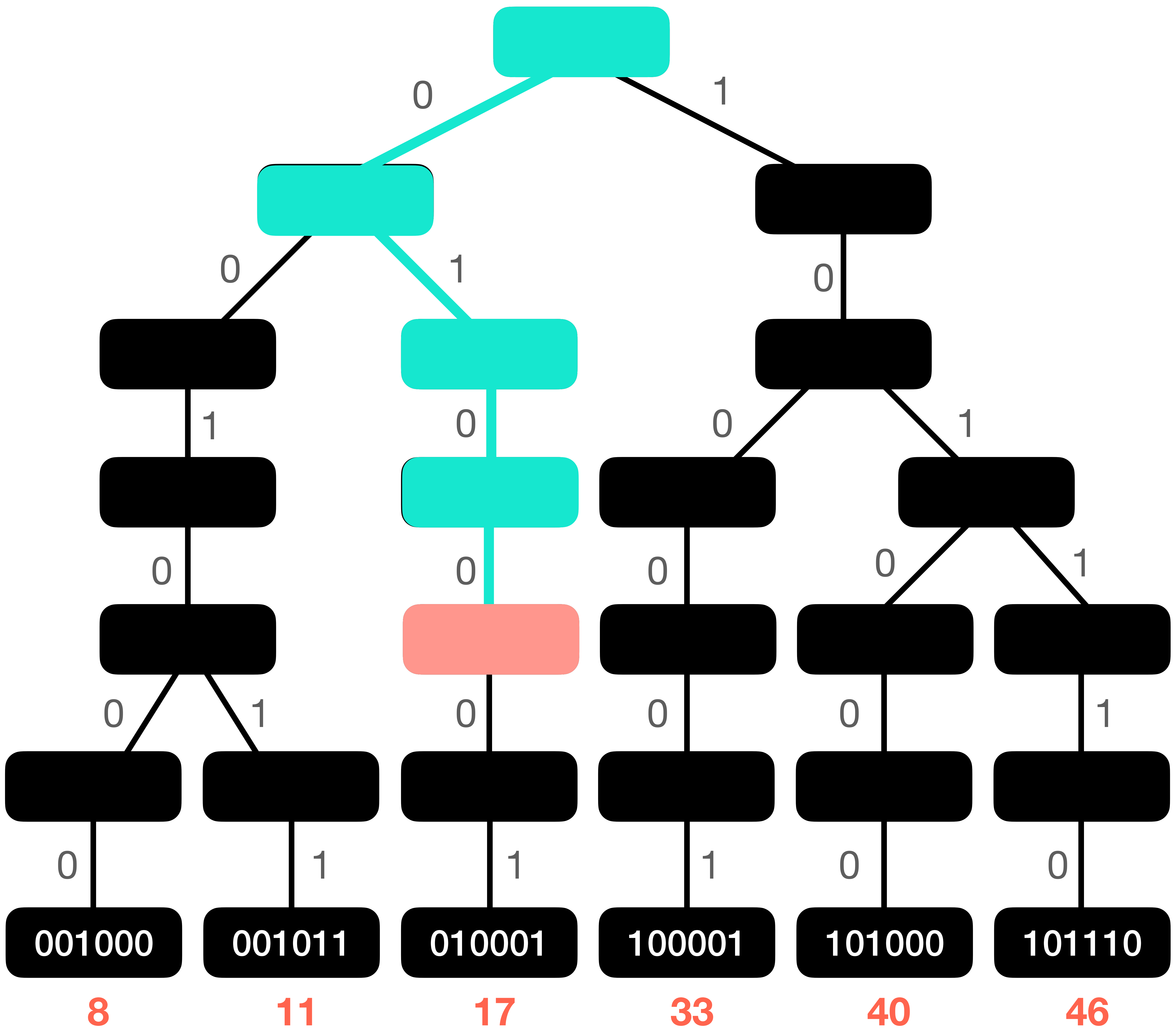
Bitwise tries for integers

- **Tries** are data structures to represent a collection of strings, where common prefixes are represented once.
- Integers are binary strings of length $\log_2 U$, hence the trie has $\log_2 U$ levels.
- All operations in $O(\log U)$ time.
- For example: Successor(x) and Predecessor(x) by following the bits of x and "walking backwards" if necessary.

Member(19), $[19]_2 = 010011$

Successor(38), $[38]_2 = 100110$

$S = \{8,11,17,33,40,46\}$, $U = 64$



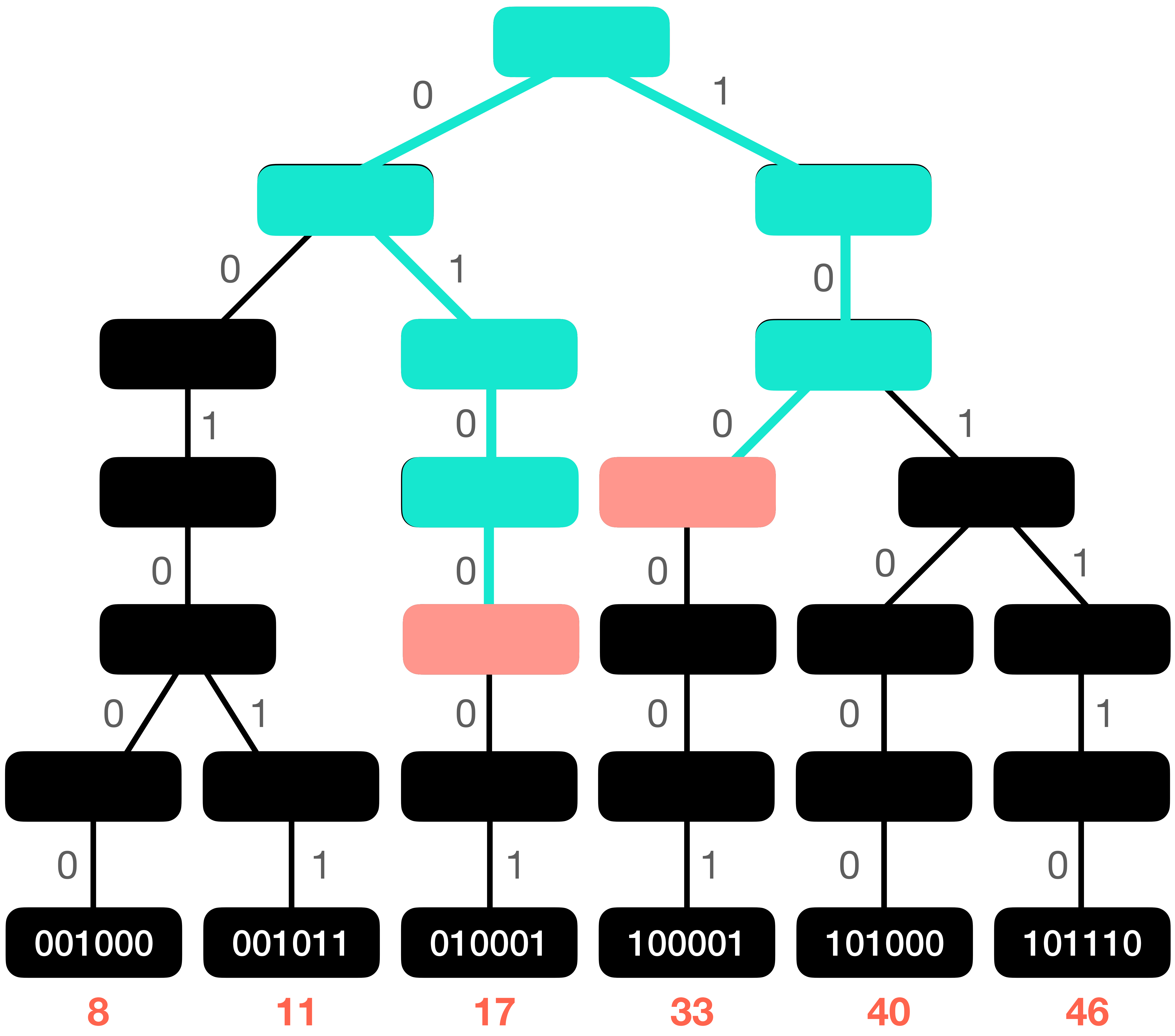
Bitwise tries for integers

- **Tries** are data structures to represent a collection of strings, where common prefixes are represented once.
- Integers are binary strings of length $\log_2 U$, hence the trie has $\log_2 U$ levels.
- All operations in $O(\log U)$ time.
- For example: Successor(x) and Predecessor(x) by following the bits of x and "walking backwards" if necessary.

Member(19), $[19]_2 = \underline{010}011$

Successor(38), $[38]_2 = \underline{100}110$

$S = \{8,11,17,33,40,46\}$, $U = 64$



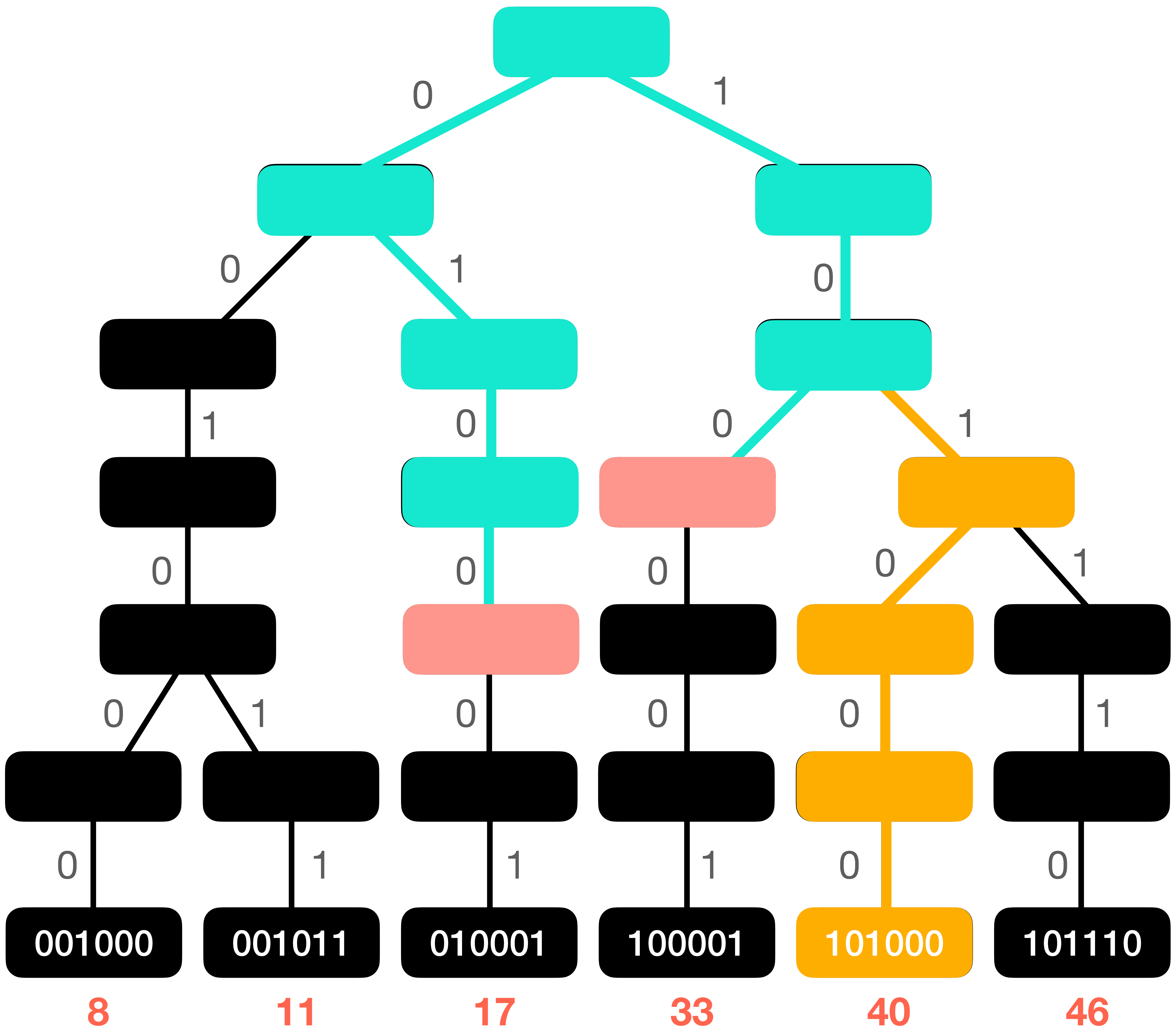
Bitwise tries for integers

- **Tries** are data structures to represent a collection of strings, where common prefixes are represented once.
- Integers are binary strings of length $\log_2 U$, hence the trie has $\log_2 U$ levels.
- All operations in $O(\log U)$ time.
- For example: Successor(x) and Predecessor(x) by following the bits of x and "walking backwards" if necessary.

Member(19), $[19]_2 = \underline{010}011$

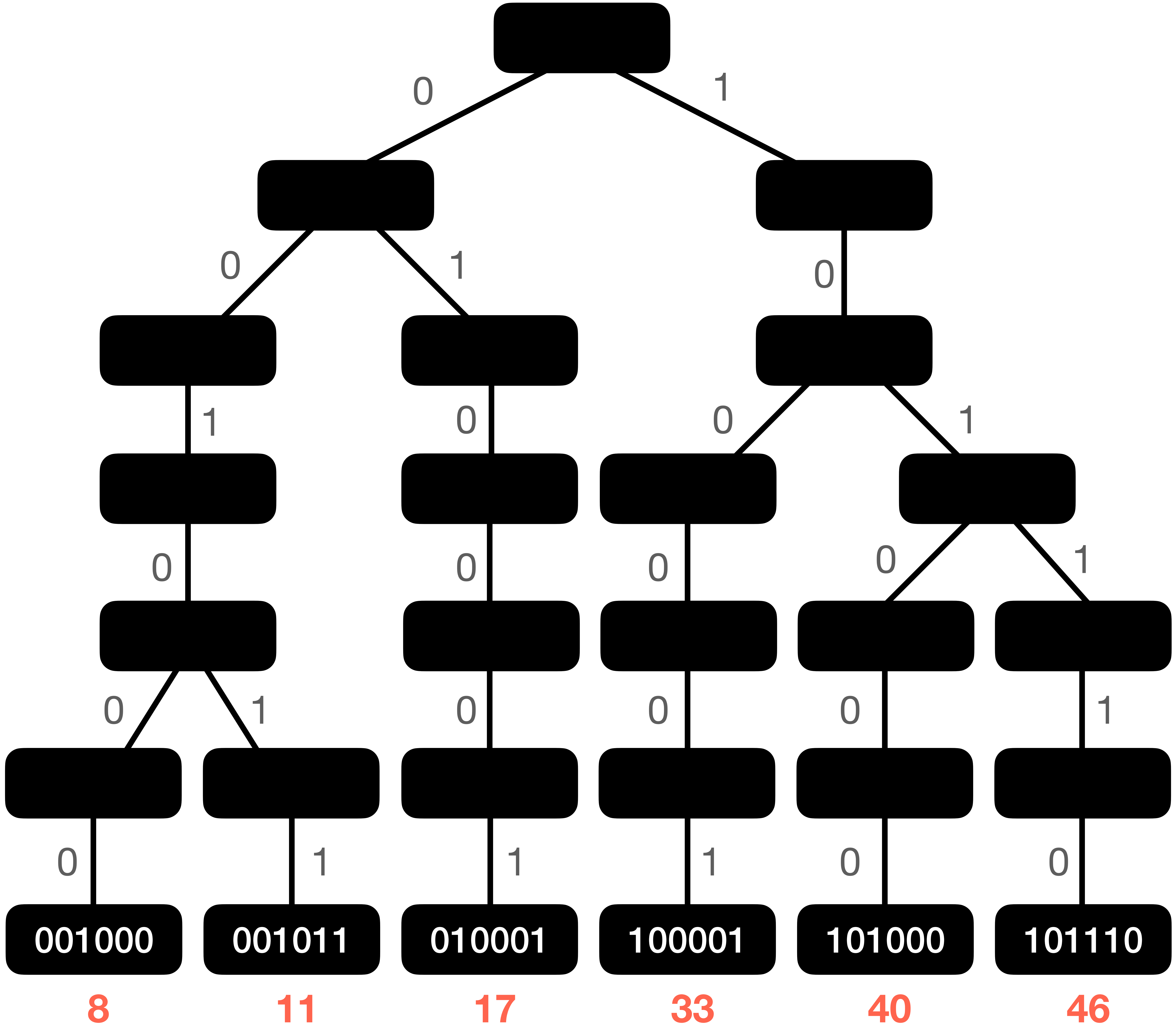
Successor(38), $[38]_2 = \underline{100}110$

$S = \{8,11,17,33,40,46\}$, $U = 64$



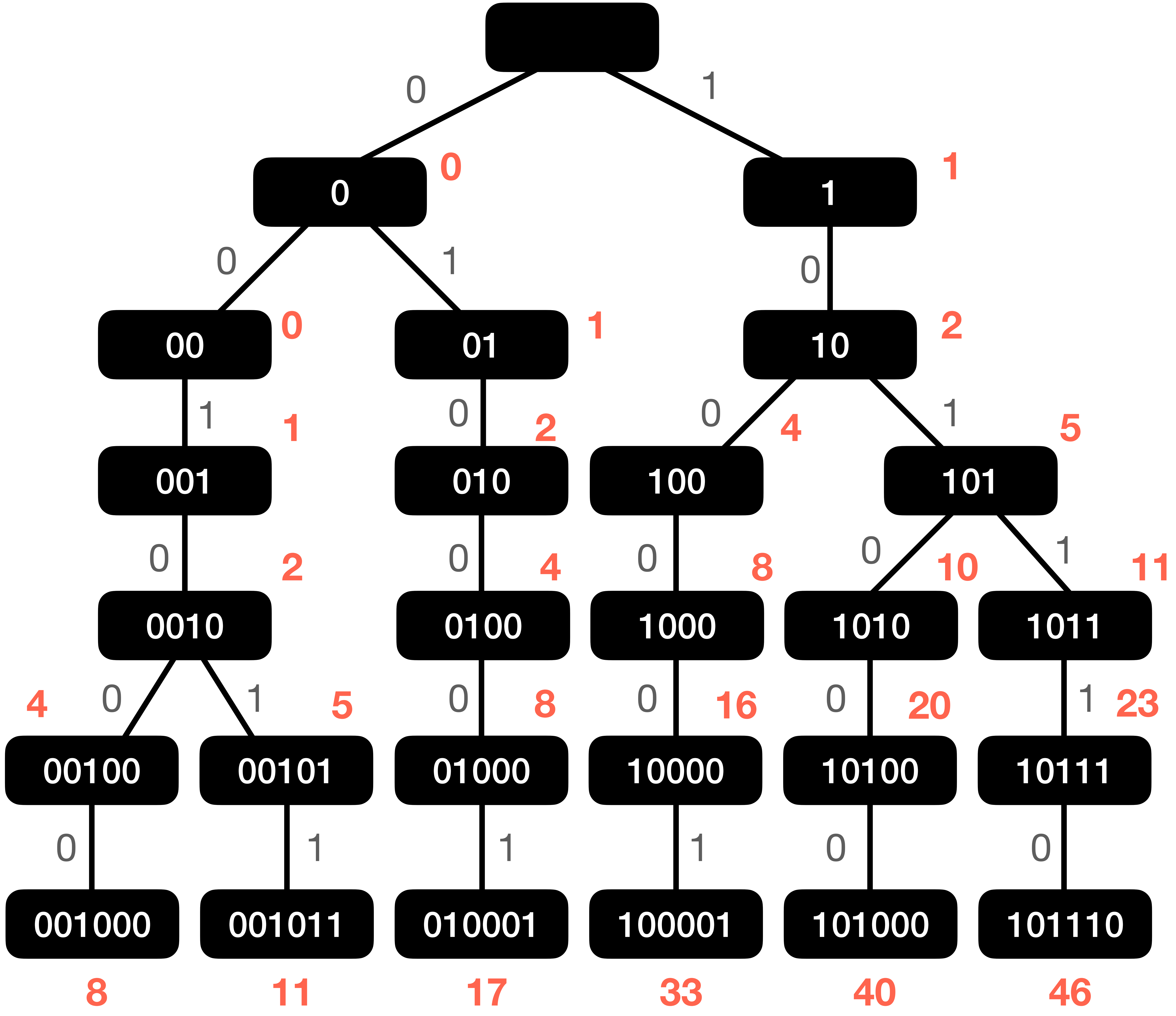
Bitwise tries for integers

- Let’s consider all the prefixes in the internal nodes.



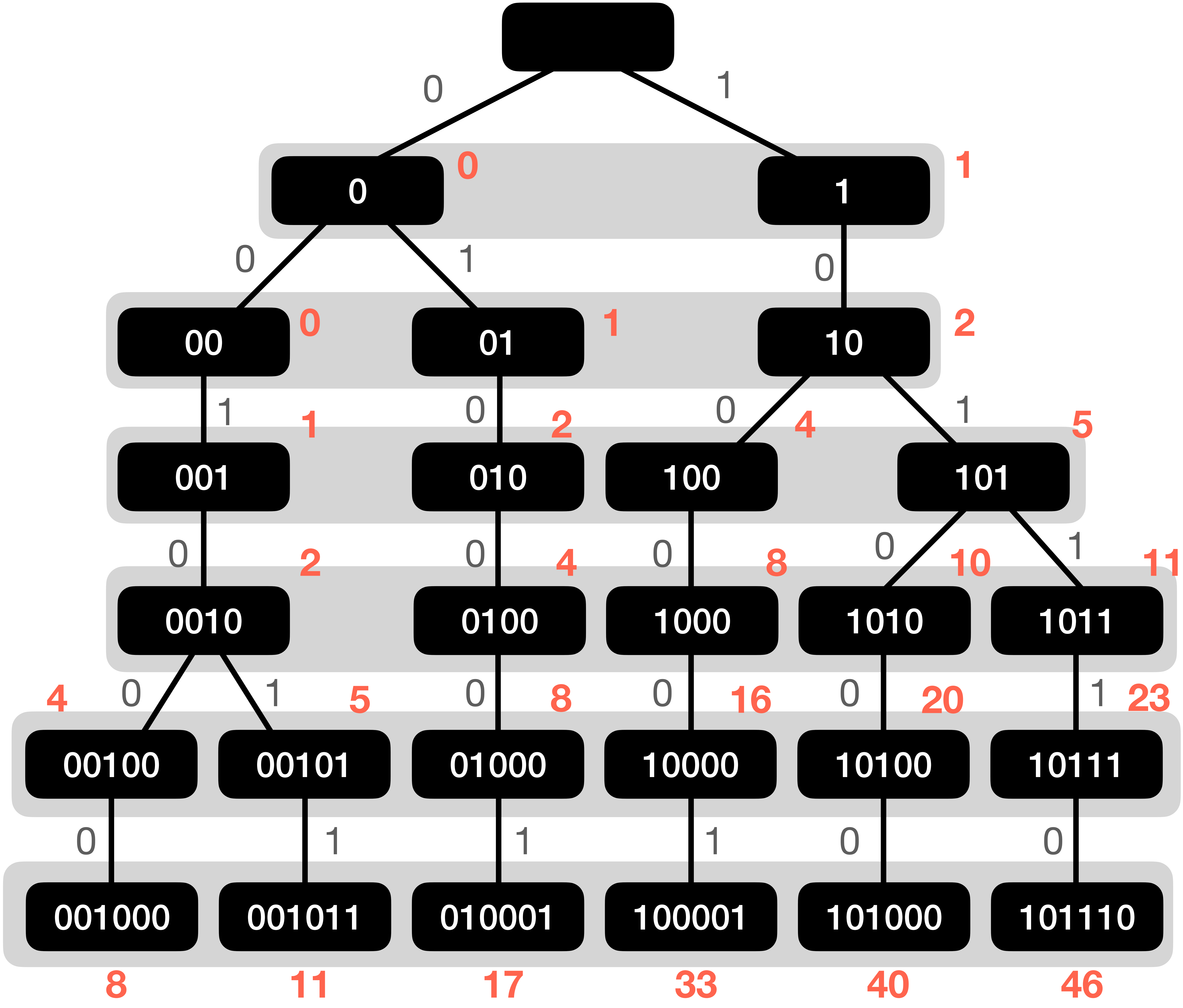
Bitwise tries for integers

- Let's consider all the prefixes in the internal nodes.



Binary searching the prefixes

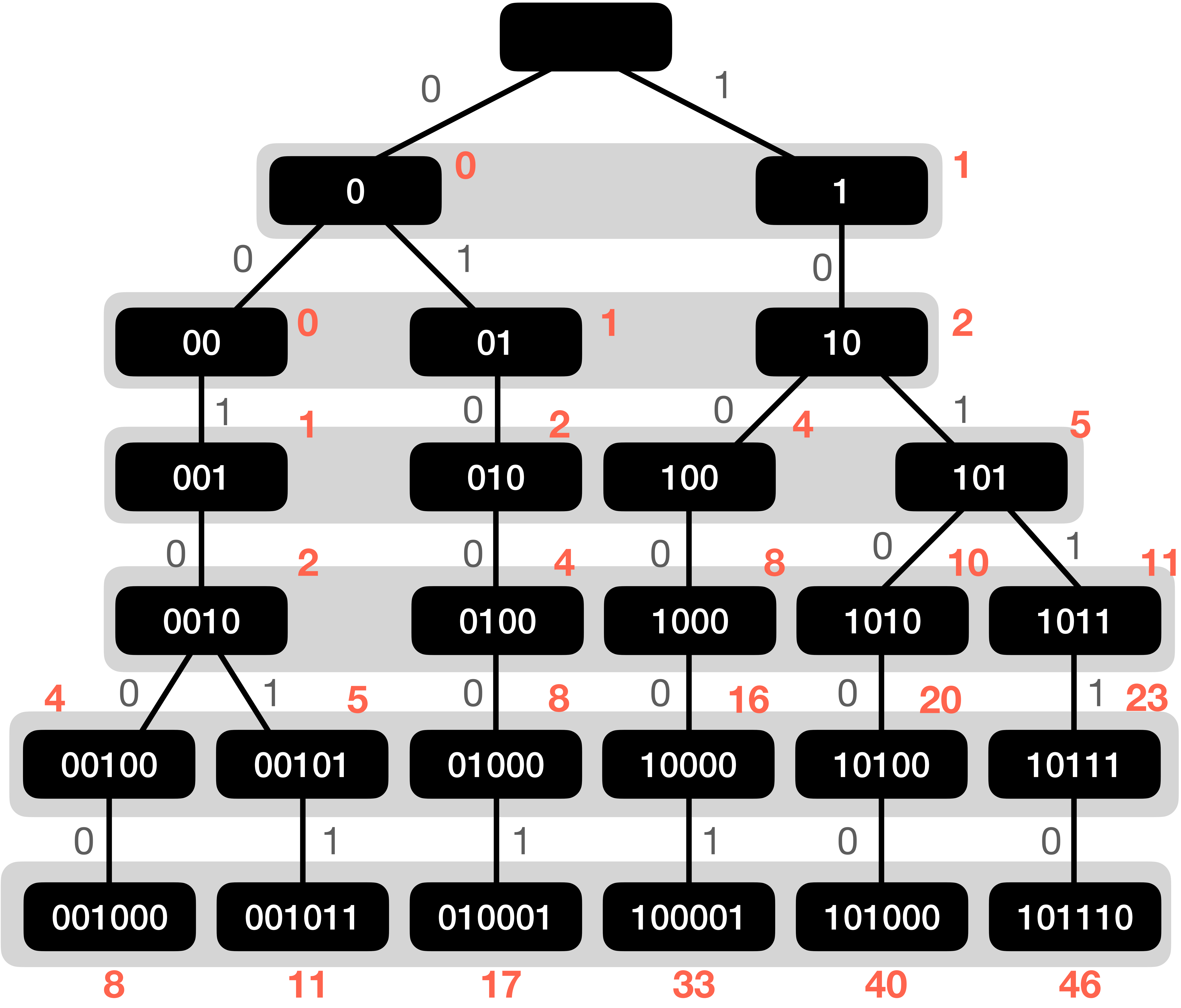
- Let's consider all the prefixes in the internal nodes.
- **Claim.** Longer prefixes of a number correspond to larger numbers.
- Hence, we can do a **binary search** on the binary representation of a query x to determine the **longest common prefix (LCP)** between x and a node in the trie.



Binary searching the prefixes

- Let's consider all the prefixes in the internal nodes.
- **Claim.** Longer prefixes of a number correspond to larger numbers.
- Hence, we can do a **binary search** on the binary representation of a query x to determine the **longest common prefix (LCP)** between x and a node in the trie.

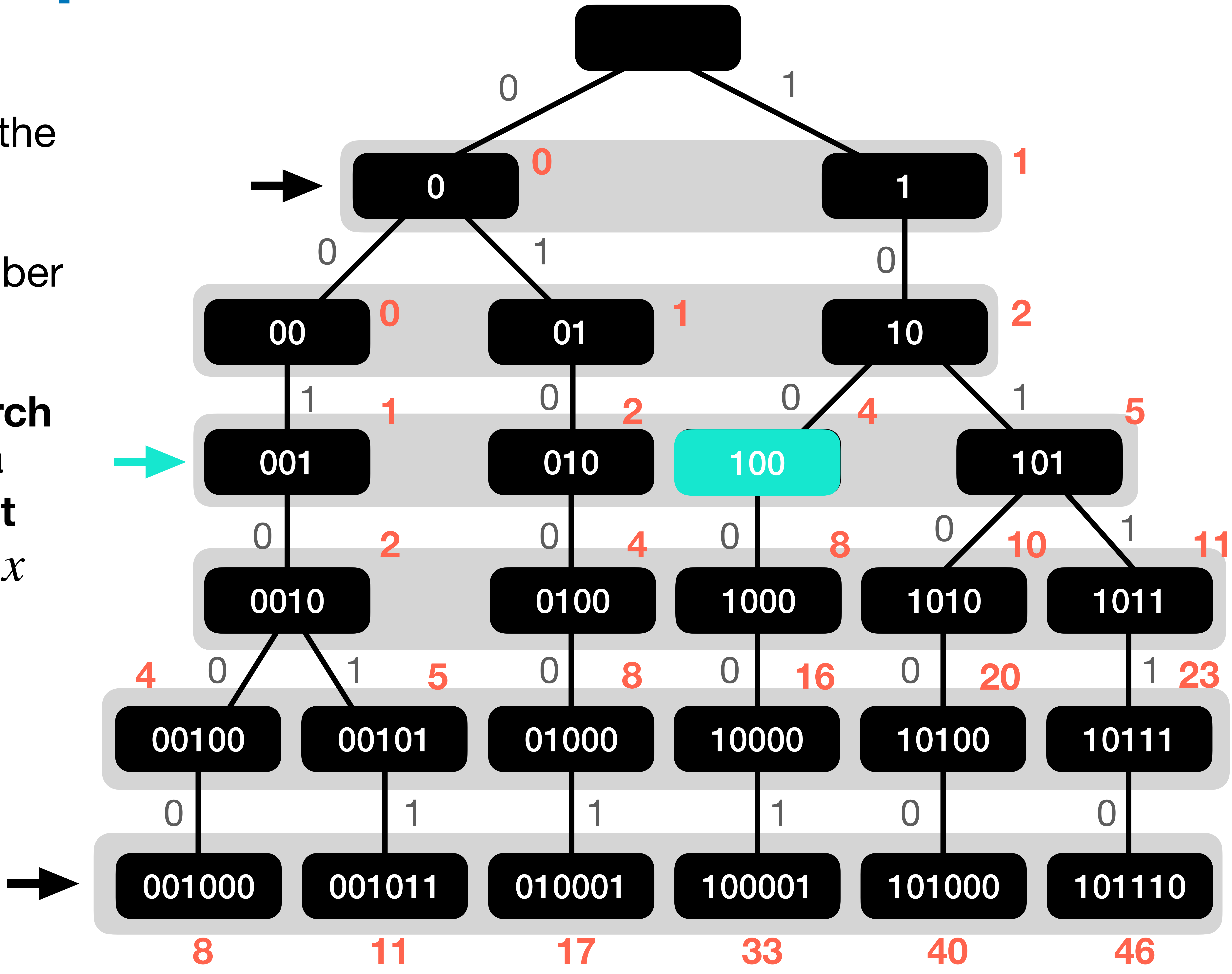
Successor(38), $[38]_2 = 100110$



Binary searching the prefixes

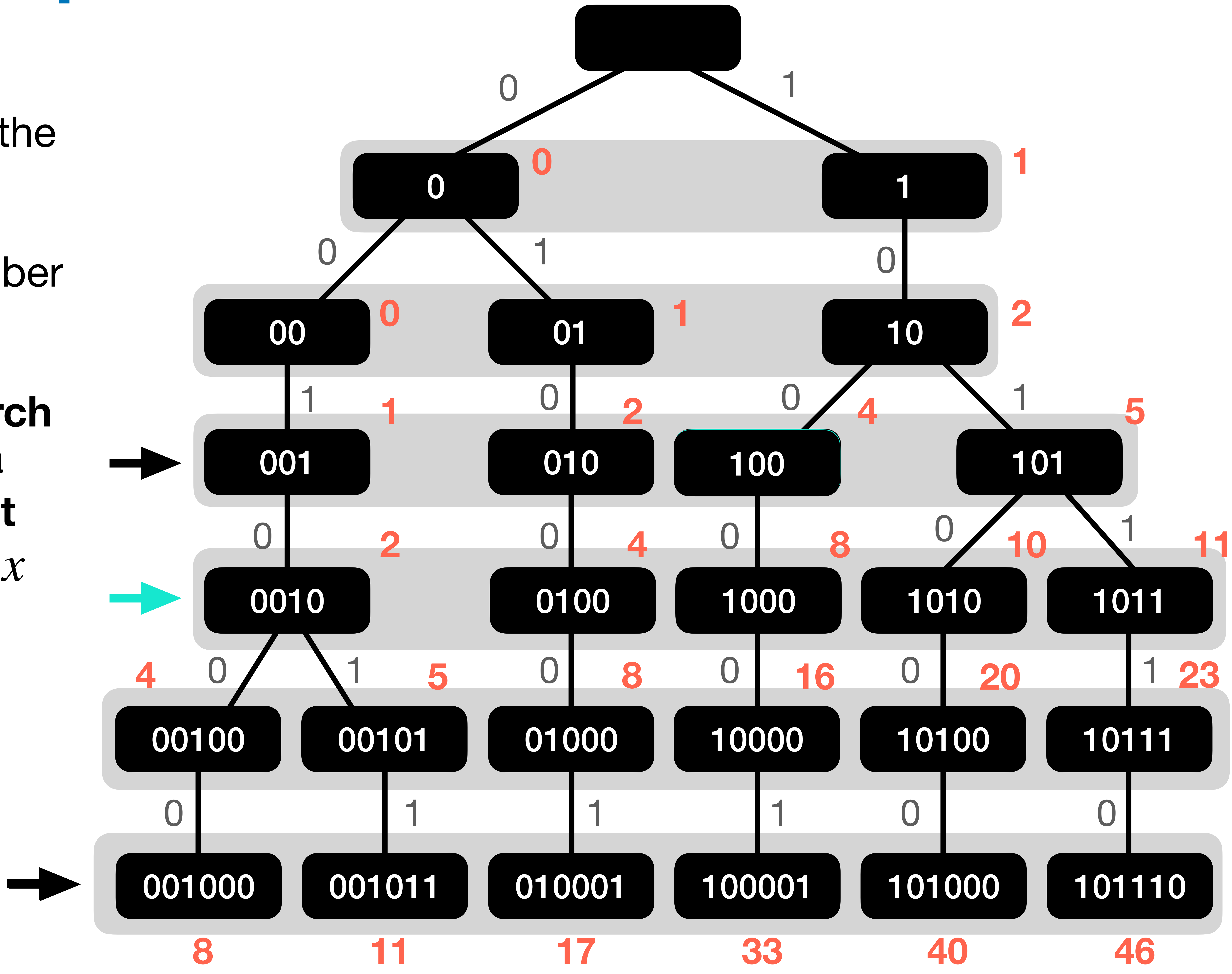
- Let's consider all the prefixes in the internal nodes.
- **Claim.** Longer prefixes of a number correspond to larger numbers.
- Hence, we can do a **binary search** on the binary representation of a query x to determine the **longest common prefix (LCP)** between x and a node in the trie.

Successor(38), $[38]_2 = \underline{100}110$



Binary searching the prefixes

- Let's consider all the prefixes in the internal nodes.
- **Claim.** Longer prefixes of a number correspond to larger numbers.
- Hence, we can do a **binary search** on the binary representation of a query x to determine the **longest common prefix (LCP)** between x and a node in the trie.

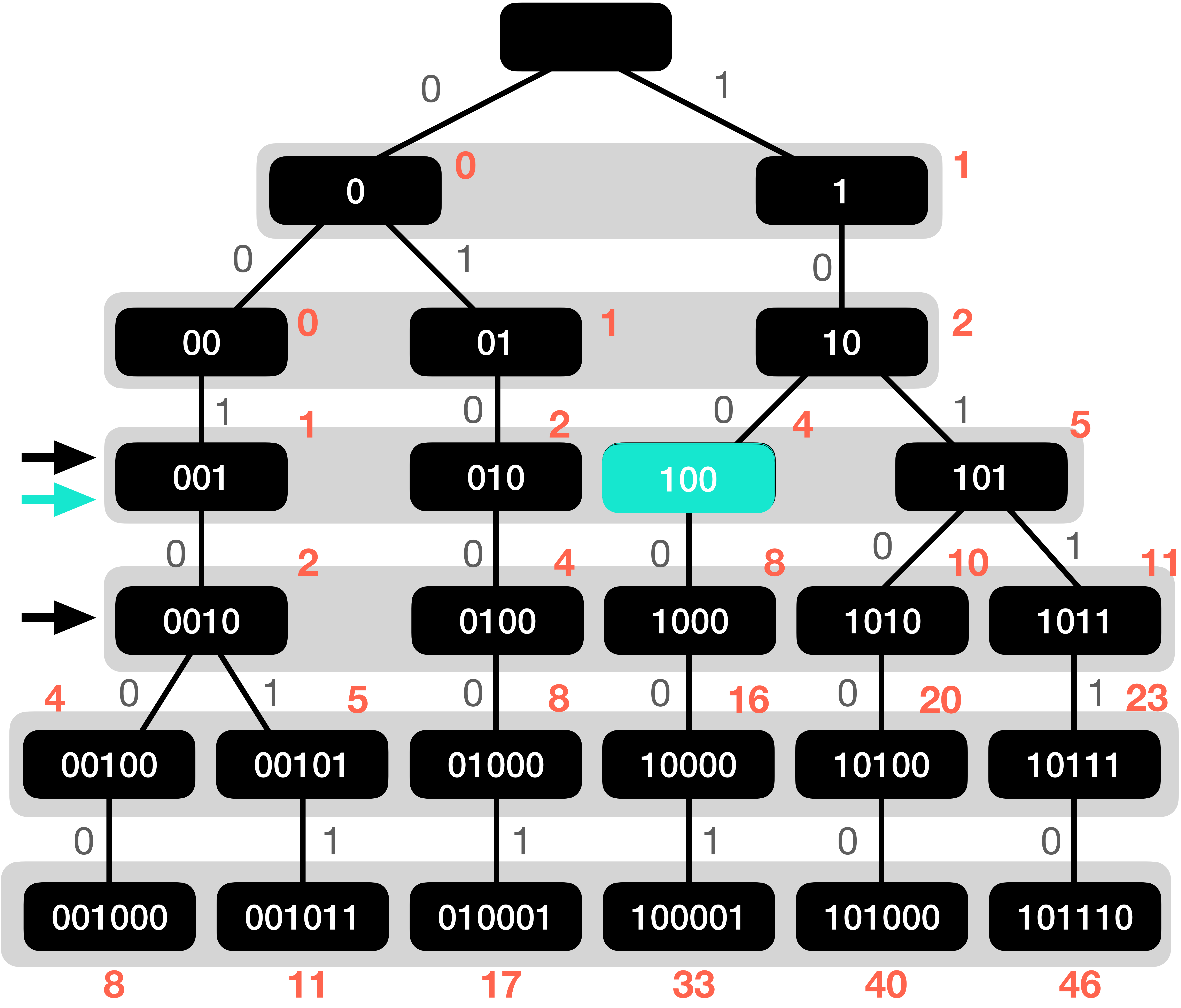


Successor(38), $[38]_2 = \underline{100}110$

Binary searching the prefixes

- Let's consider all the prefixes in the internal nodes.
- **Claim.** Longer prefixes of a number correspond to larger numbers.
- Hence, we can do a **binary search** on the binary representation of a query x to determine the **longest common prefix (LCP)** between x and a node in the trie.

Successor(38), $[38]_2 = 100110$



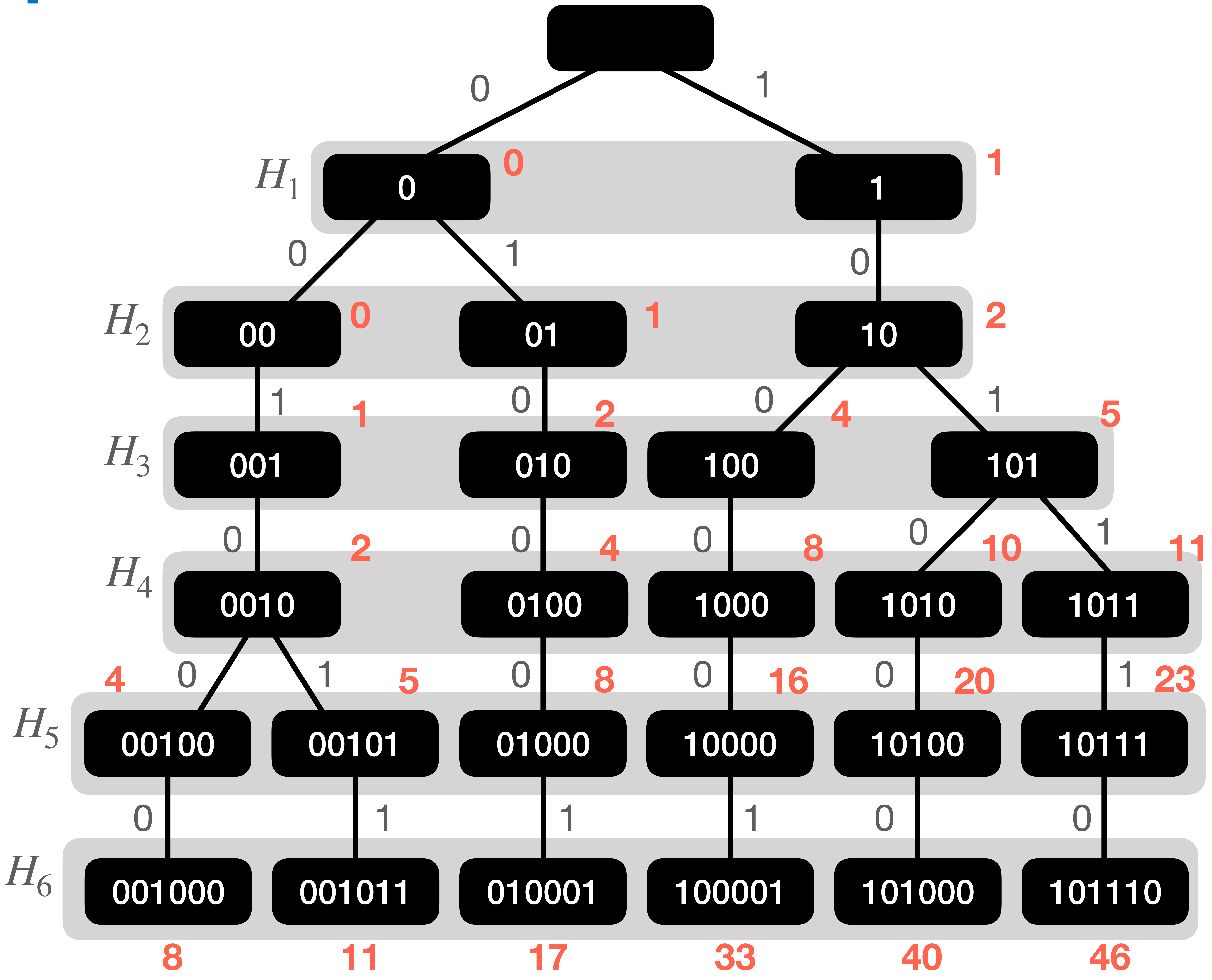
Binary searching the prefixes

- Binary search needs to determine whether a given node (a prefix) is present at a level.

- So, all the nodes at level i are inserted in a **hash table** H_i with $O(1)$ time per lookup.

- The LCP can now be determined in $O(\log \log U)$.

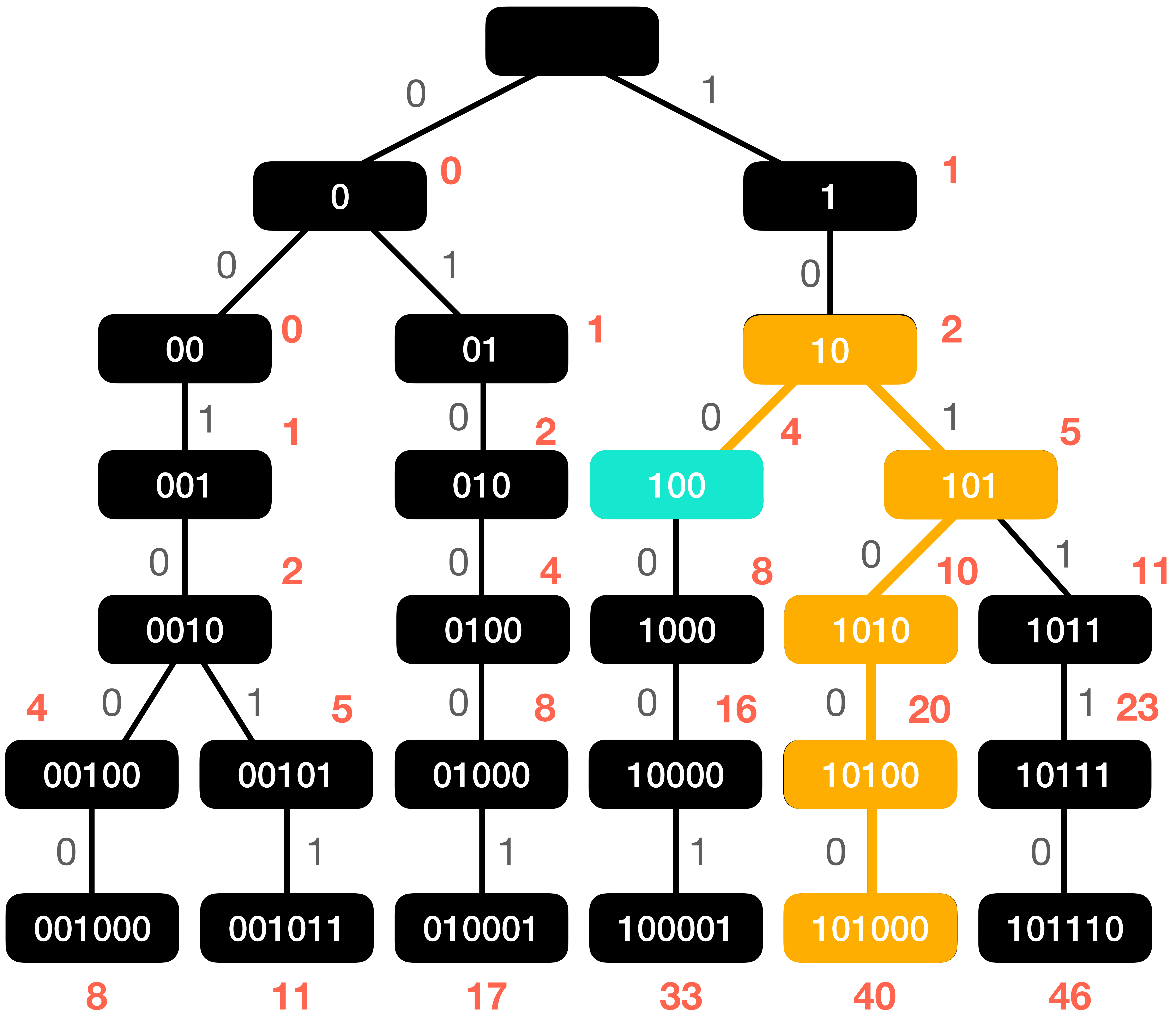
- **Bonus.** Member(x) now runs in $O(1)$ by probing the bottom-level hash table.



Successor ?

- The LCP can be determined in $O(\log \log U)$.
- From the node corresponding to the LCP we then walk backwards until we find a branching node with a 1 on the right and take the leftmost path from there.
- Still $O(\log U)$ time in the worst case.

Successor(38), $[38]_2 = 100110$

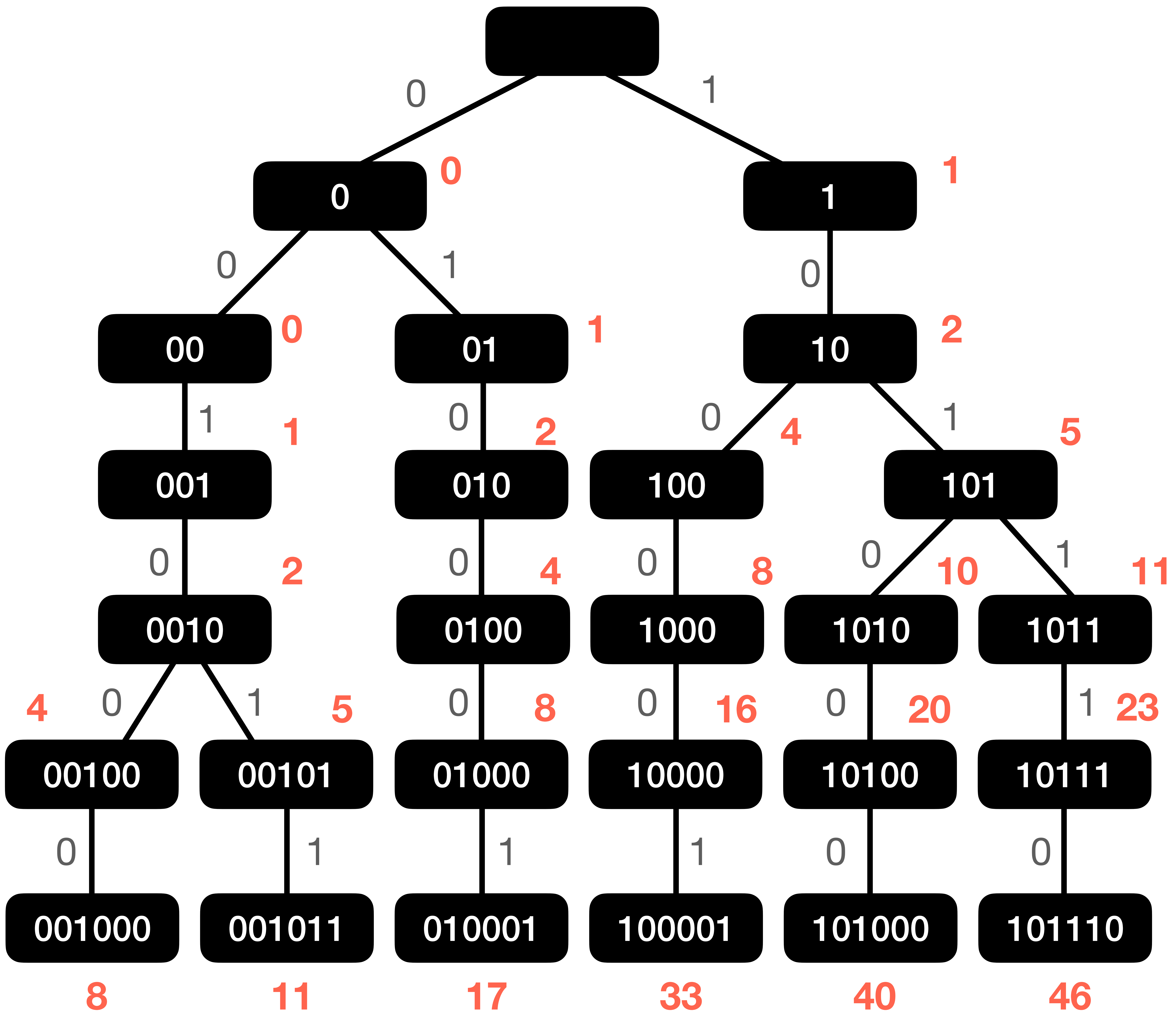


The *x*-fast trie

- **Claim.** If the binary search stops in a node v, then either v is a leaf or it has **one child** only.

  (If that were not the case, a longer match could have been found.)

- **Idea.** From v, **skip** to a leaf.



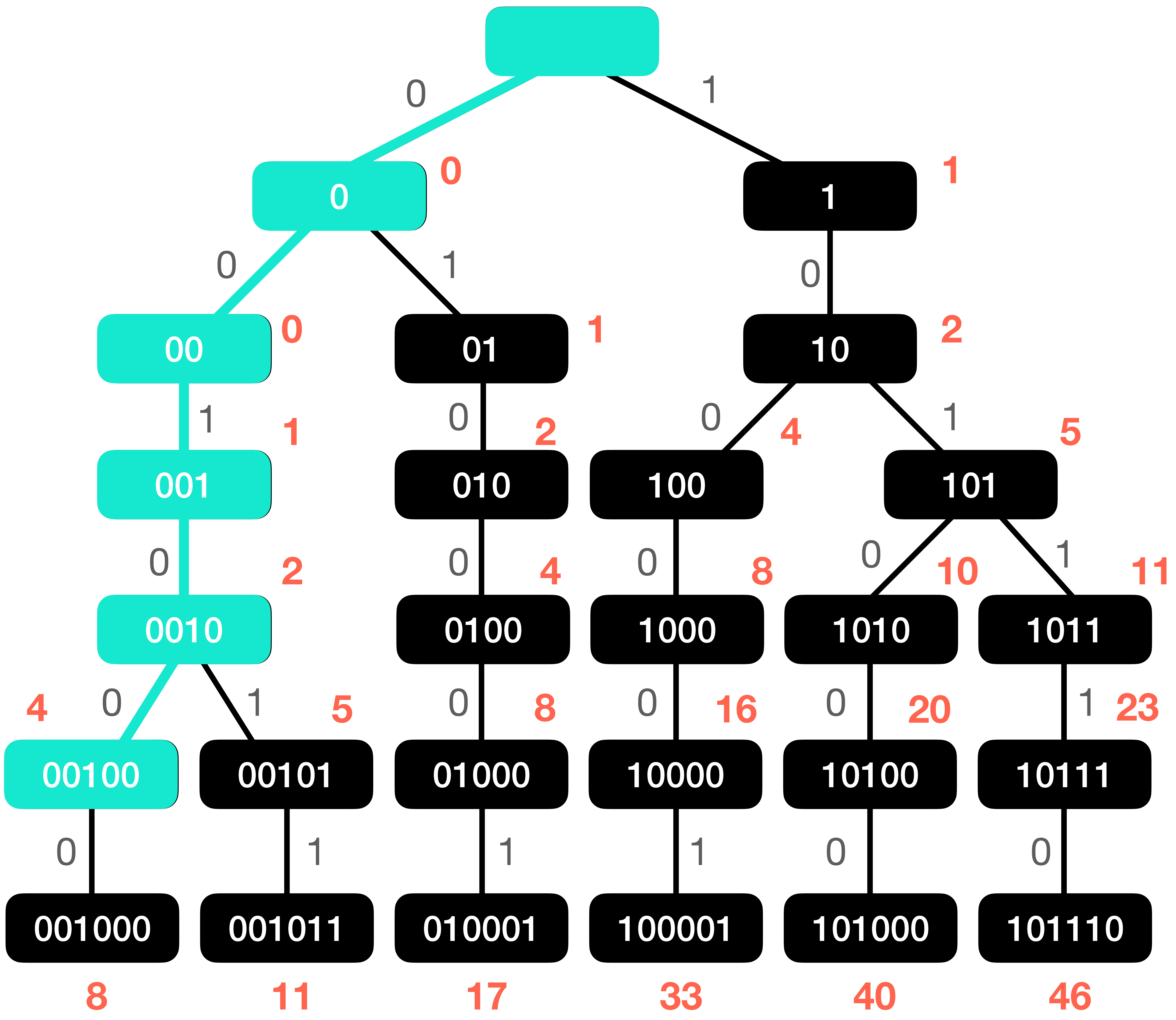
The *x*-fast trie

- **Claim.** If the binary search stops in a node v, then either v is a leaf or it has **one child** only.

  (If that were not the case, a longer match could have been found.)

- **Idea.** From v, **skip** to a leaf.

Successor(9), $[9]_2 = \underline{0010}01$



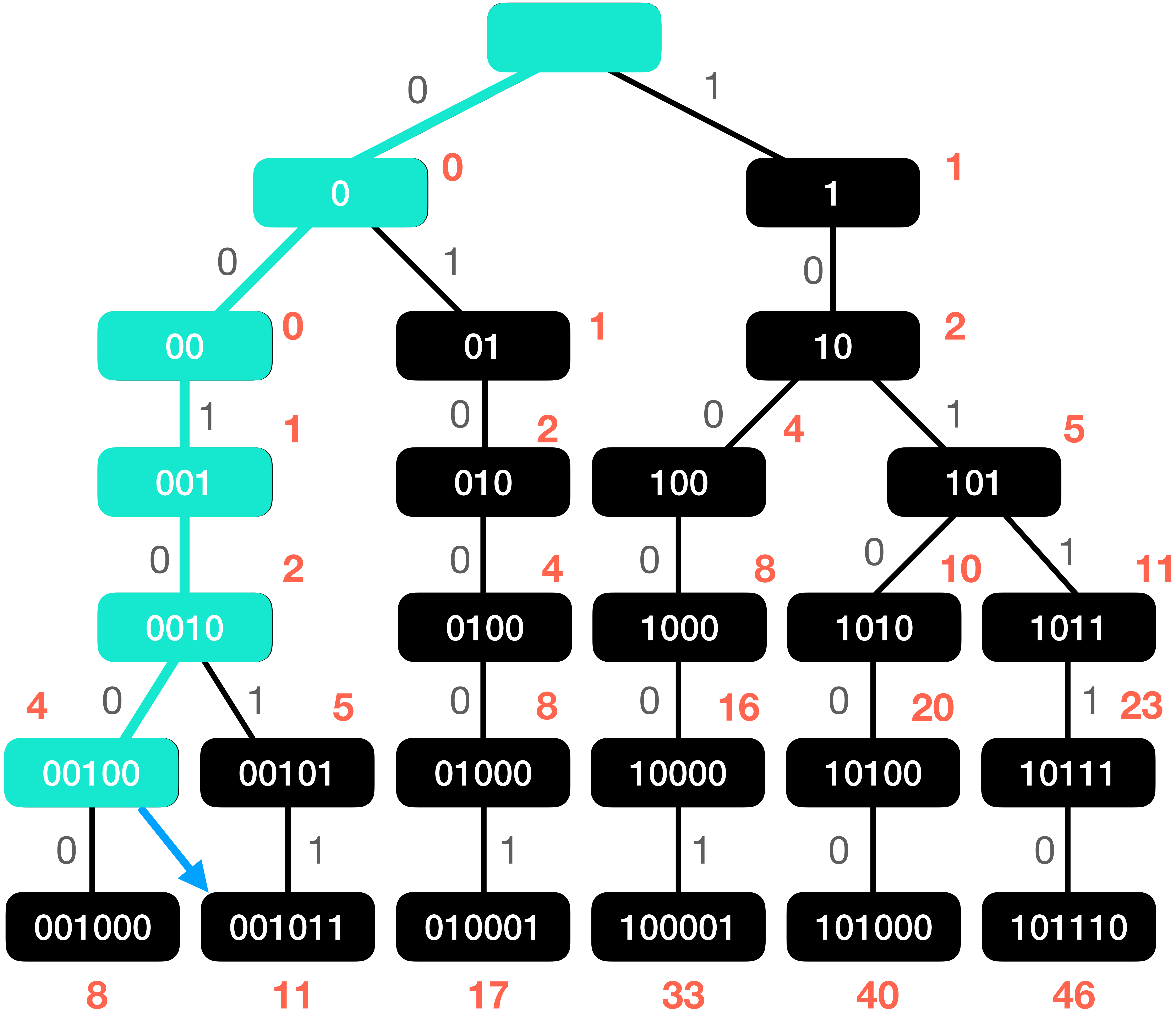
The *x*-fast trie

- **Claim.** If the binary search stops in a node v, then either v is a leaf or it has **one child** only.

  (If that were not the case, a longer match could have been found.)

- **Idea.** From v, **skip** to a leaf.

  Successor(9), $[9]_2 = \underline{00100}1$



The *x*-fast trie

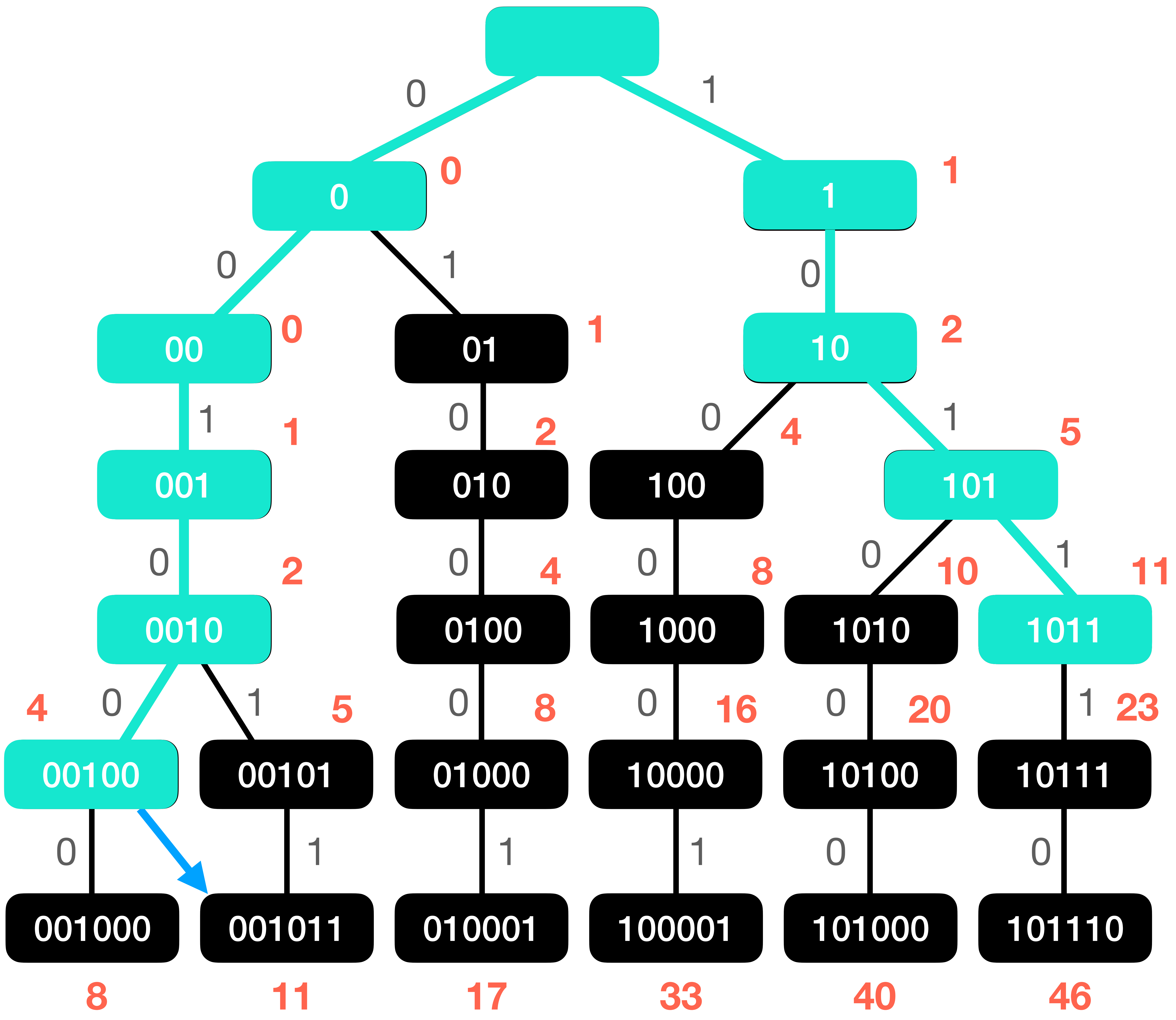
- **Claim.** If the binary search stops in a node v, then either v is a leaf or it has **one child** only.

  (If that were not the case, a longer match could have been found.)

- **Idea.** From v, **skip** to a leaf.

Successor(9), $[9]_2 = \underline{0010}01$

Predecessor(44), $[44]_2 = \underline{1011}00$



The *x*-fast trie

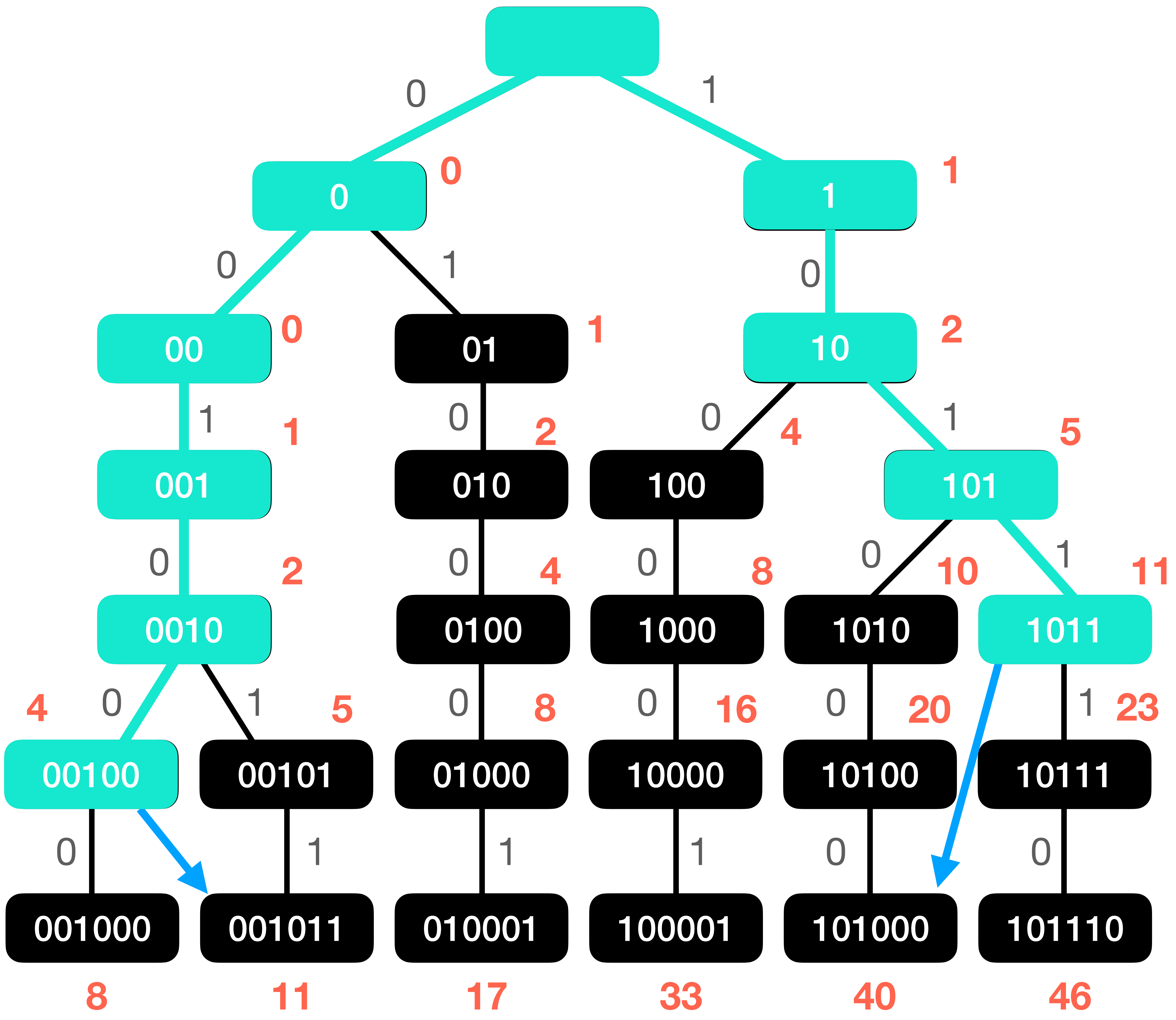
- **Claim.** If the binary search stops in a node v, then either v is a leaf or it has **one child** only.

  (If that were not the case, a longer match could have been found.)

- **Idea.** From v, **skip** to a leaf.

Successor(9), $[9]_2 = \underline{0010}01$

Predecessor(44), $[44]_2 = \underline{1011}00$

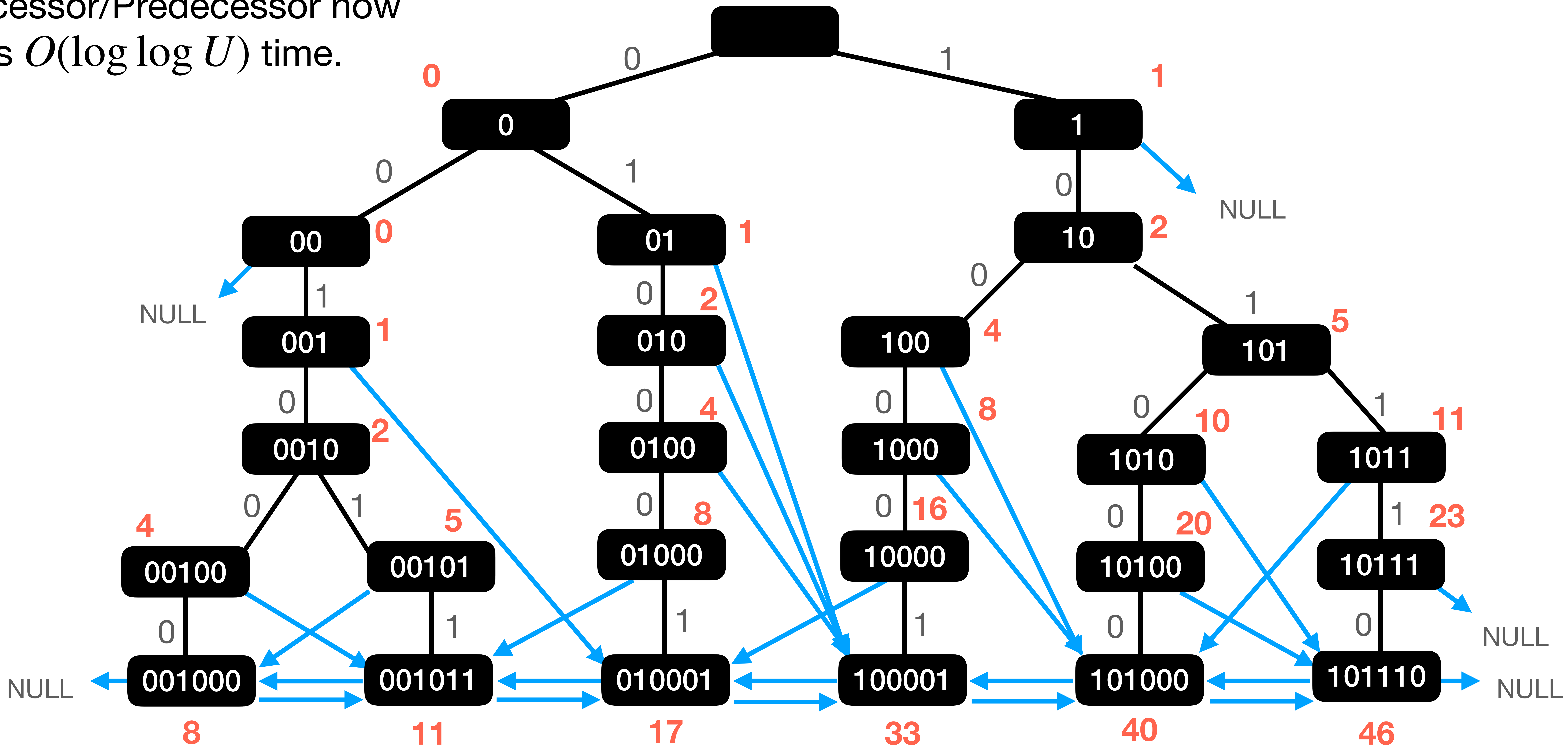


Skip pointers

- Let v be a node at level i, with **one child**.
  - If the child is labelled with **0**: v skips to the leaf Successor$\left(\left[[v]_2.1.0^{\log_2 U - i - 1}\right]_{10} \right)$.
  - If the child is labelled with **1**: v skips to the leaf Predecessor$\left(\left[[v]_2.0.0^{\log_2 U - i - 1}\right]_{10} \right)$.
- Also each leaf has a pointer to the prev/next leaf.

The *x*-fast trie

- Successor/Predecessor now takes $O(\log \log U)$ time.



The *x*-fast trie

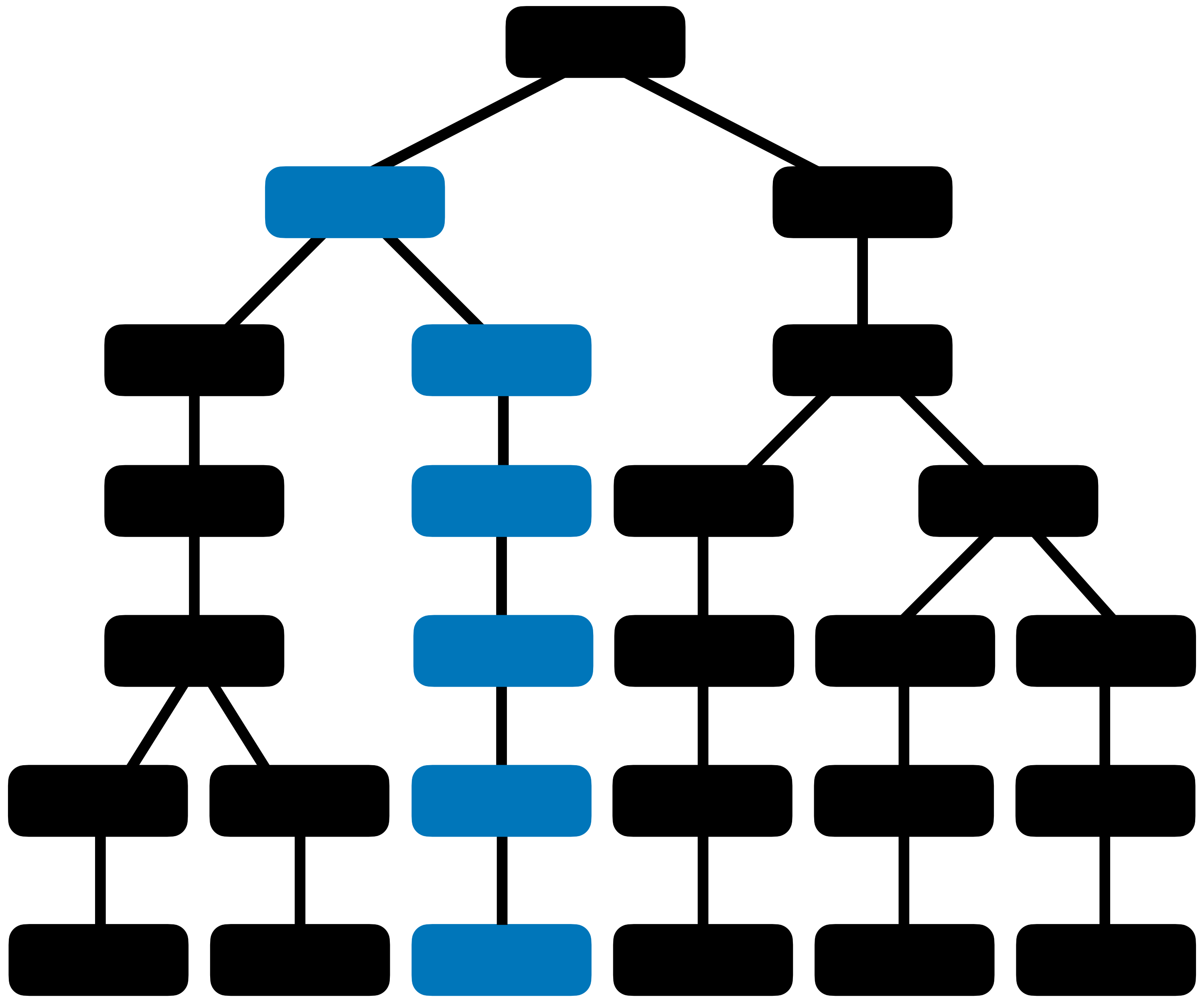
- Min/Max in $O(1)$ by caching these values.
- Member in $O(1)$.
- Predecessor/Successor in $O(\log \log U)$.
- Also, Subset$(\ell, r) := \{x \in S \mid \ell < x < r\}$ can be returned in time $O(\log \log U + |\text{Subset}(\ell, r)|)$: scan the linked-list from Successor(ℓ) to Predecessor(r). (Range query.)

- **Q.** Space?

The *x*-fast trie

- Min/Max in $O(1)$ by caching these values.
- Member in $O(1)$.
- Predecessor/Successor in $O(\log\log U)$.
- Also, Subset$(\ell, r) := \{x \in S \mid \ell < x < r\}$ can be returned in time $O(\log\log U + |\text{Subset}(\ell, r)|)$: scan the linked-list from Successor(ℓ) to Predecessor(r). (Range query.)

- **Q.** Space?
- **A.** A key contributes to at most $\log_2 U$ nodes, hence space is $O(n \log U)$.

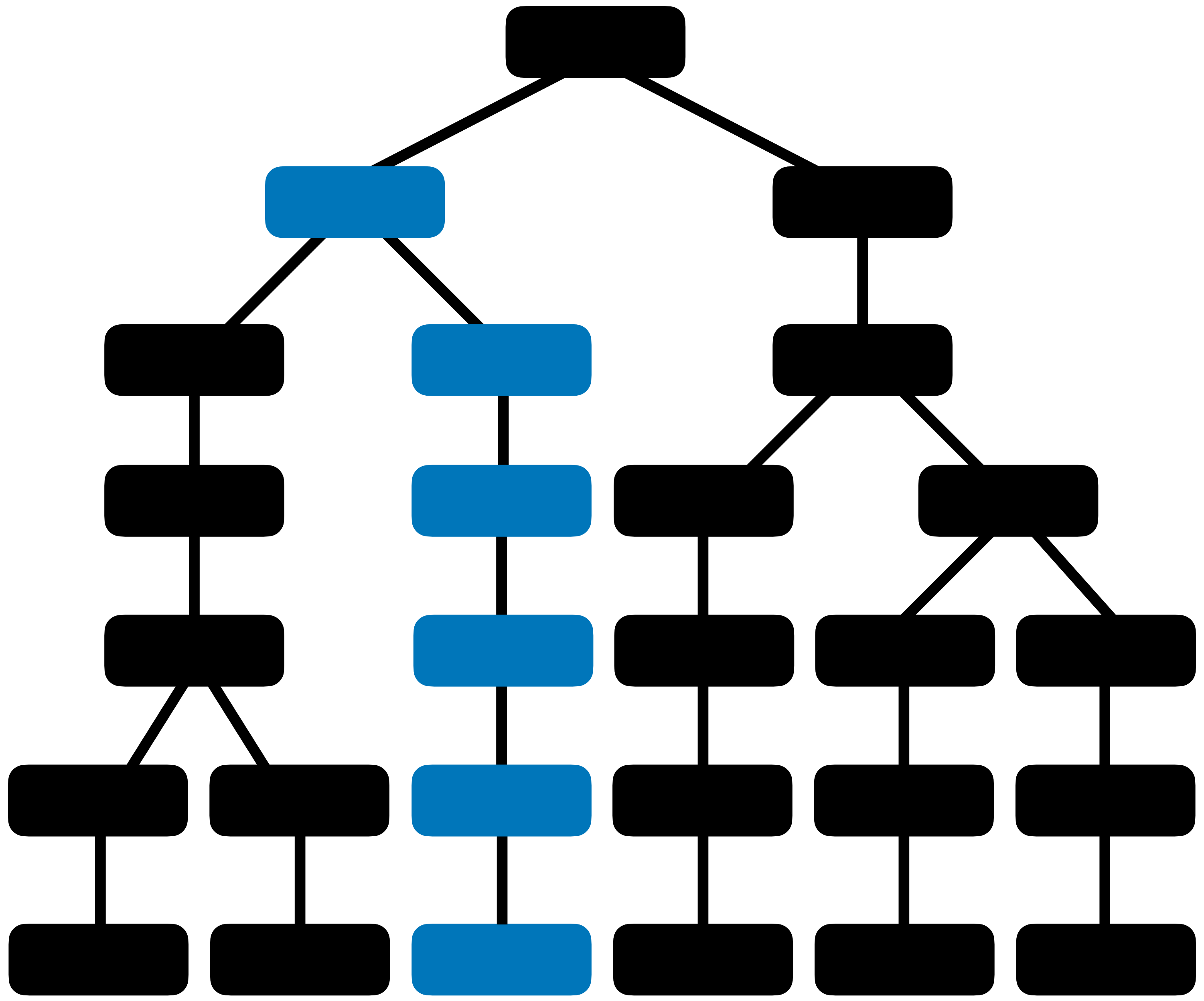


The *x*-fast trie

- Min/Max in $O(1)$ by caching these values.
- Member in $O(1)$.
- Predecessor/Successor in $O(\log \log U)$.
- Also, Subset$(\ell, r) := \{x \in S \mid \ell < x < r\}$ can be returned in time $O(\log \log U + |\text{Subset}(\ell, r)|)$: scan the linked-list from Successor(ℓ) to Predecessor(r). (Range query.)

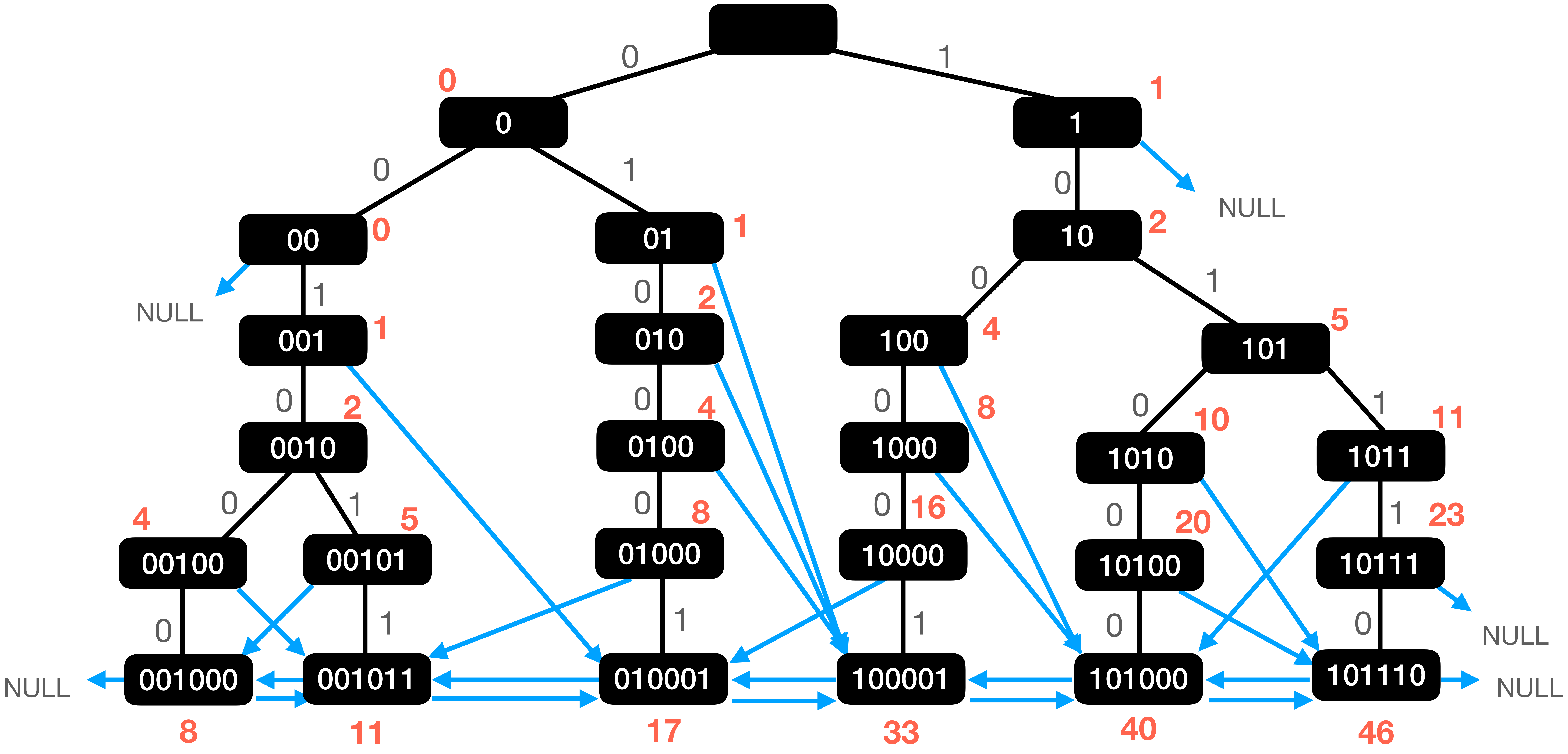
- **Q.** Space?
- **A.** A key contributes to at most $\log_2 U$ nodes, hence space is $O(n \log U)$.

- **Q.** Insert/Delete?



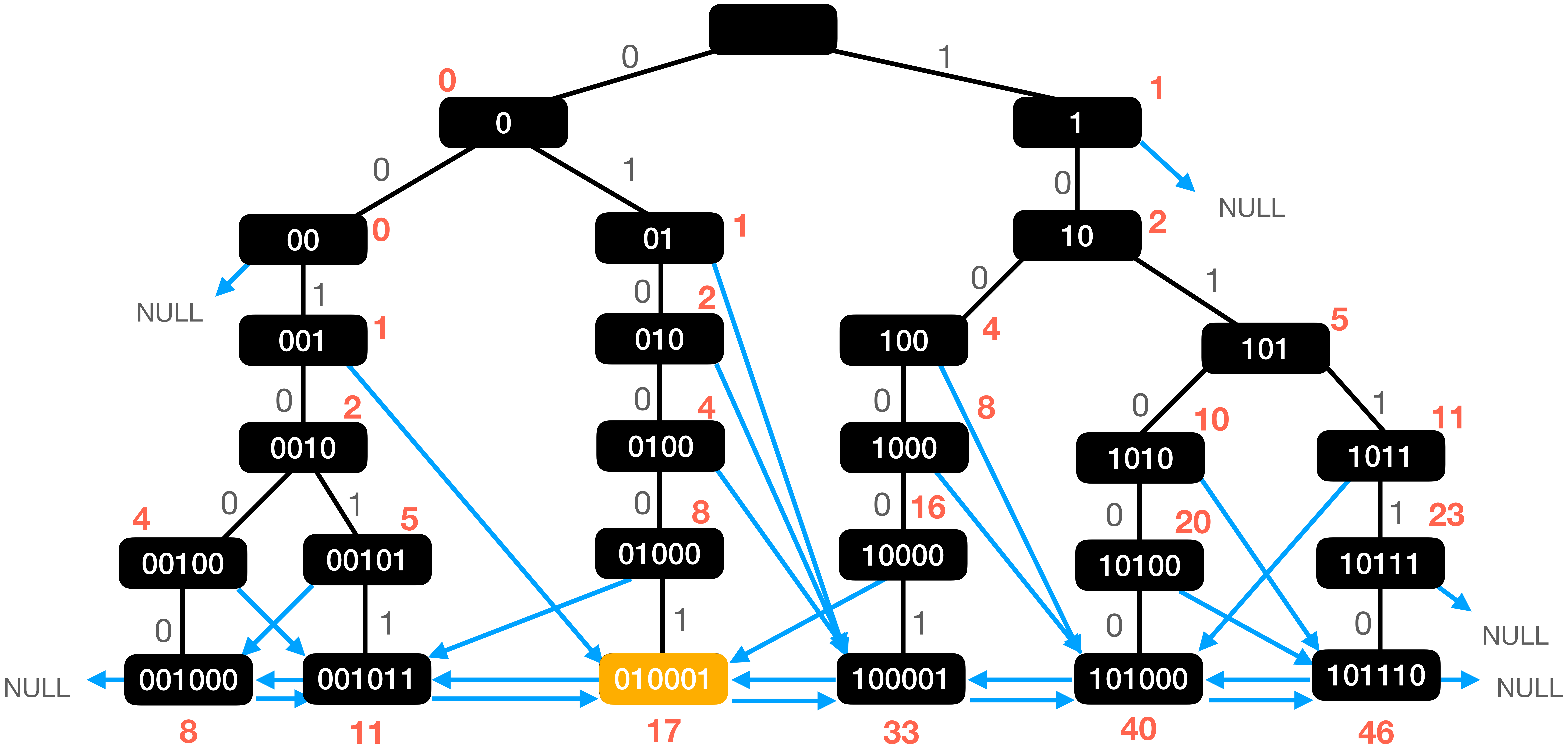
Insert

Insert(13), $[13]_2 = 001101$:



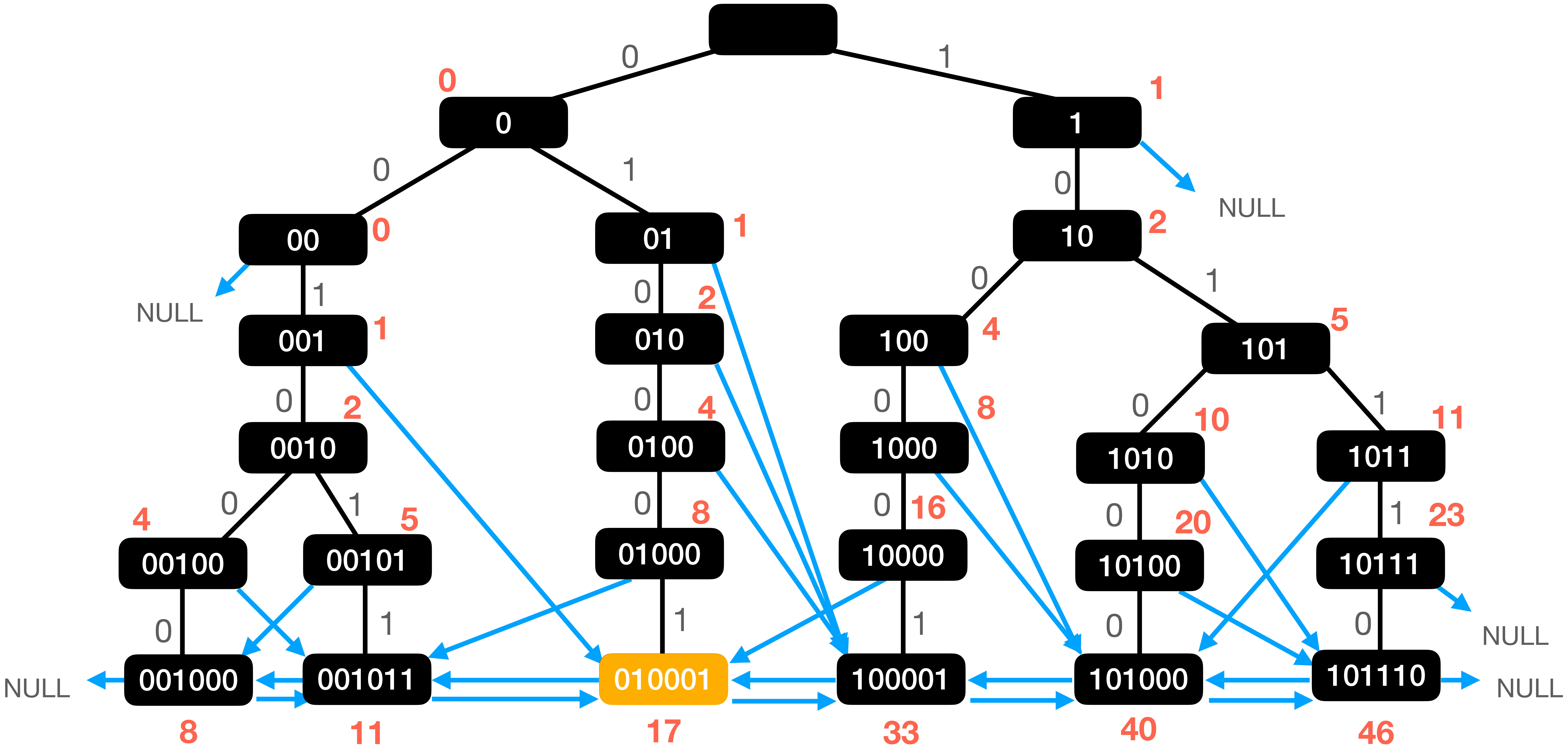
Insert

Insert(13), $[13]_2 = 001101$: 1. find Successor(13);



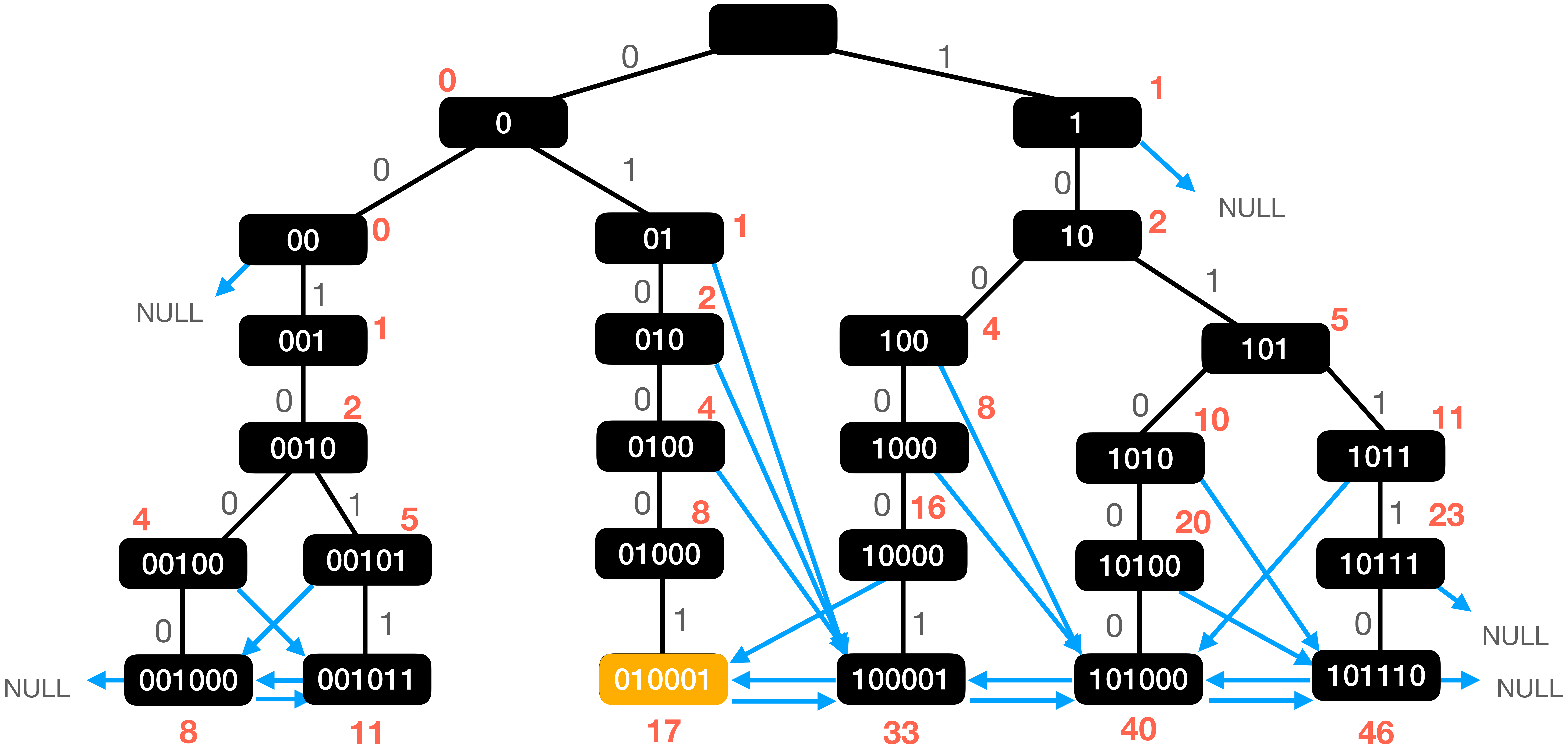
Insert

Insert(13), $[13]_2 = 001101$: 1. find Successor(13); 2. add x to the trie;



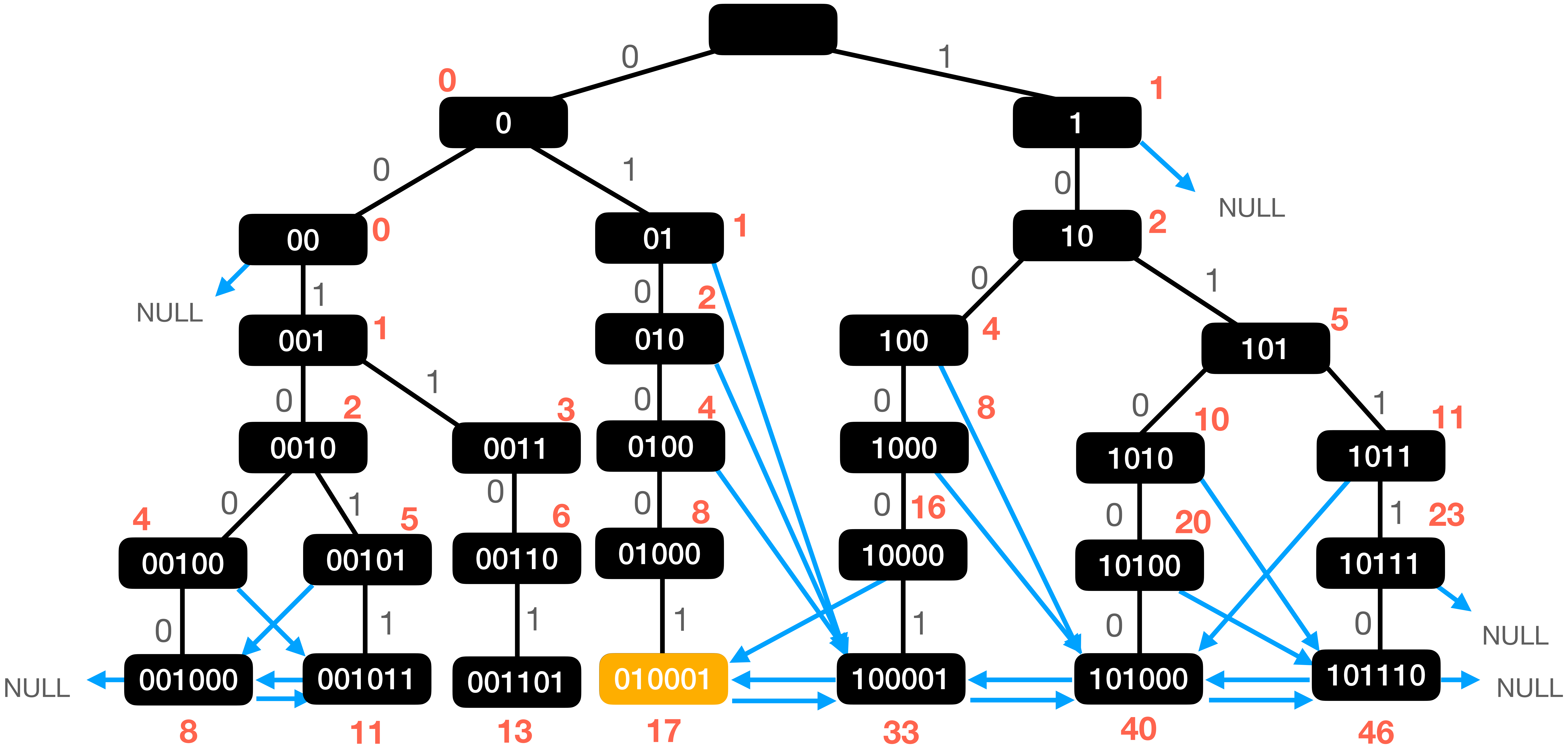
Insert

Insert(13), $[13]_2 = 001101$: 1. find Successor(13); 2. add x to the trie;



Insert

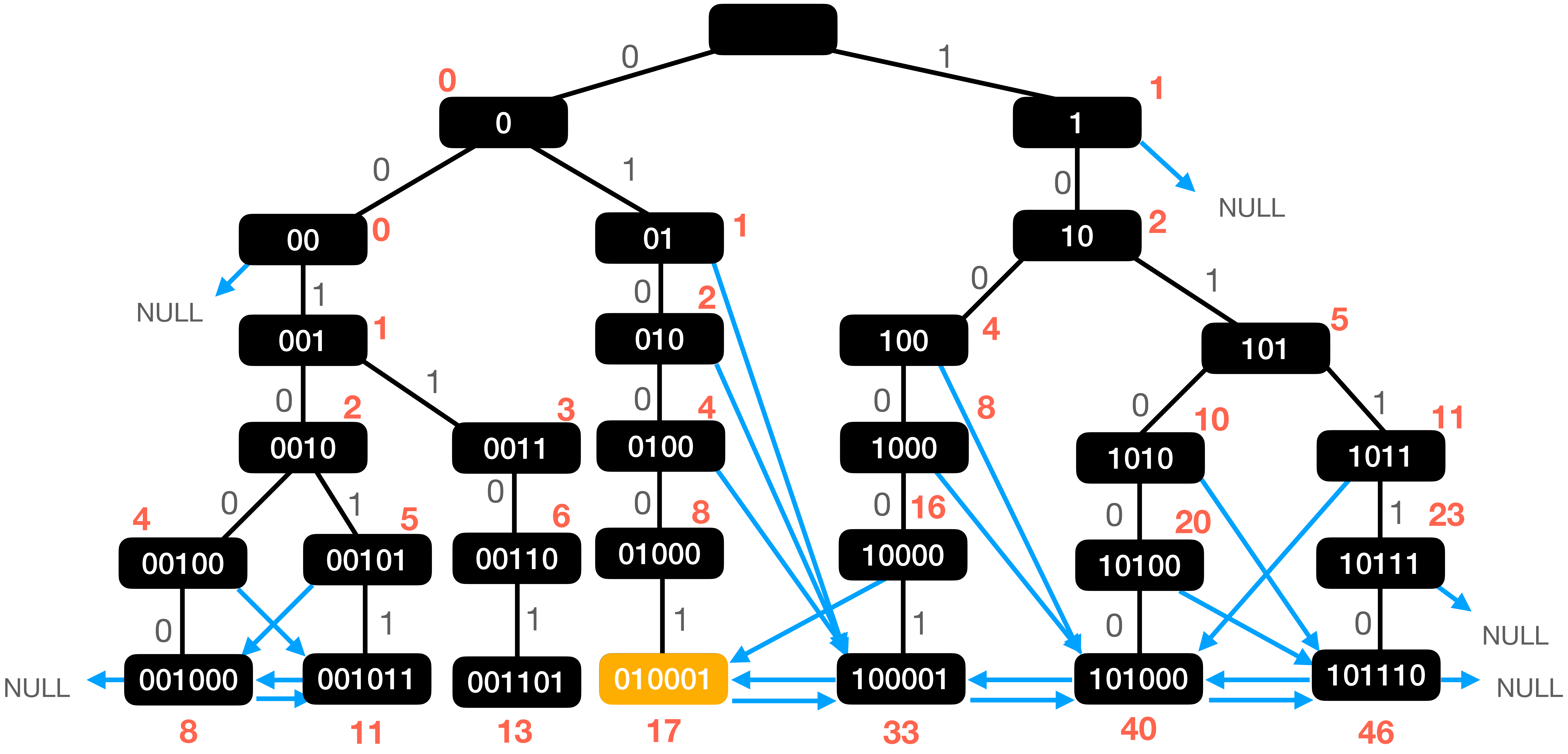
Insert(13), $[13]_2 = 001101$: **1**. find Successor(13); **2**. add x to the trie;



Insert

Insert(13), $[13]_2 = 001101$: 1. find Successor(13); 2. add x to the trie;

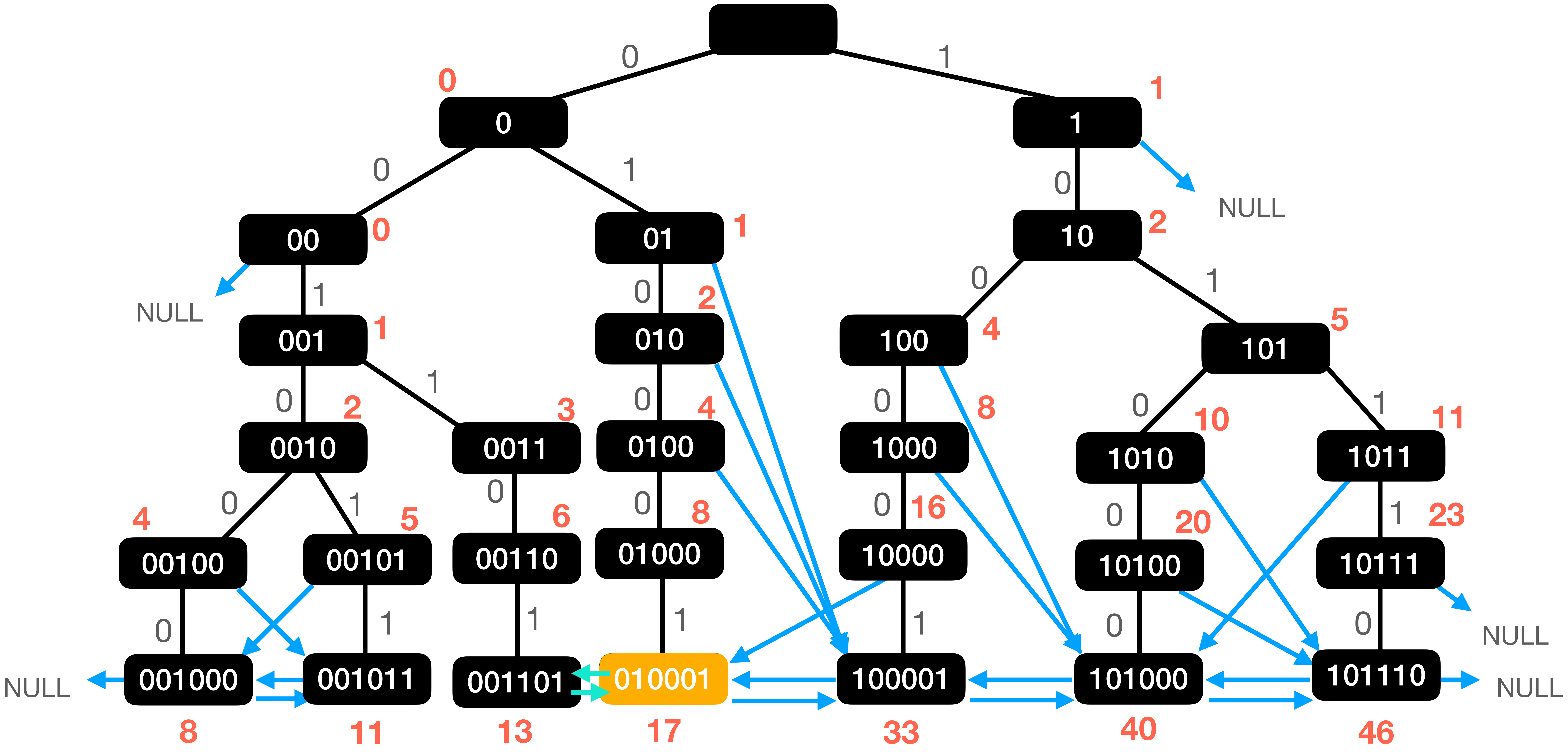
3. Walk backwards from predecessor, successor, and x to update skip pointers.



Insert

Insert(13), $[13]_2 = 001101$: 1. find Successor(13); 2. add x to the trie;

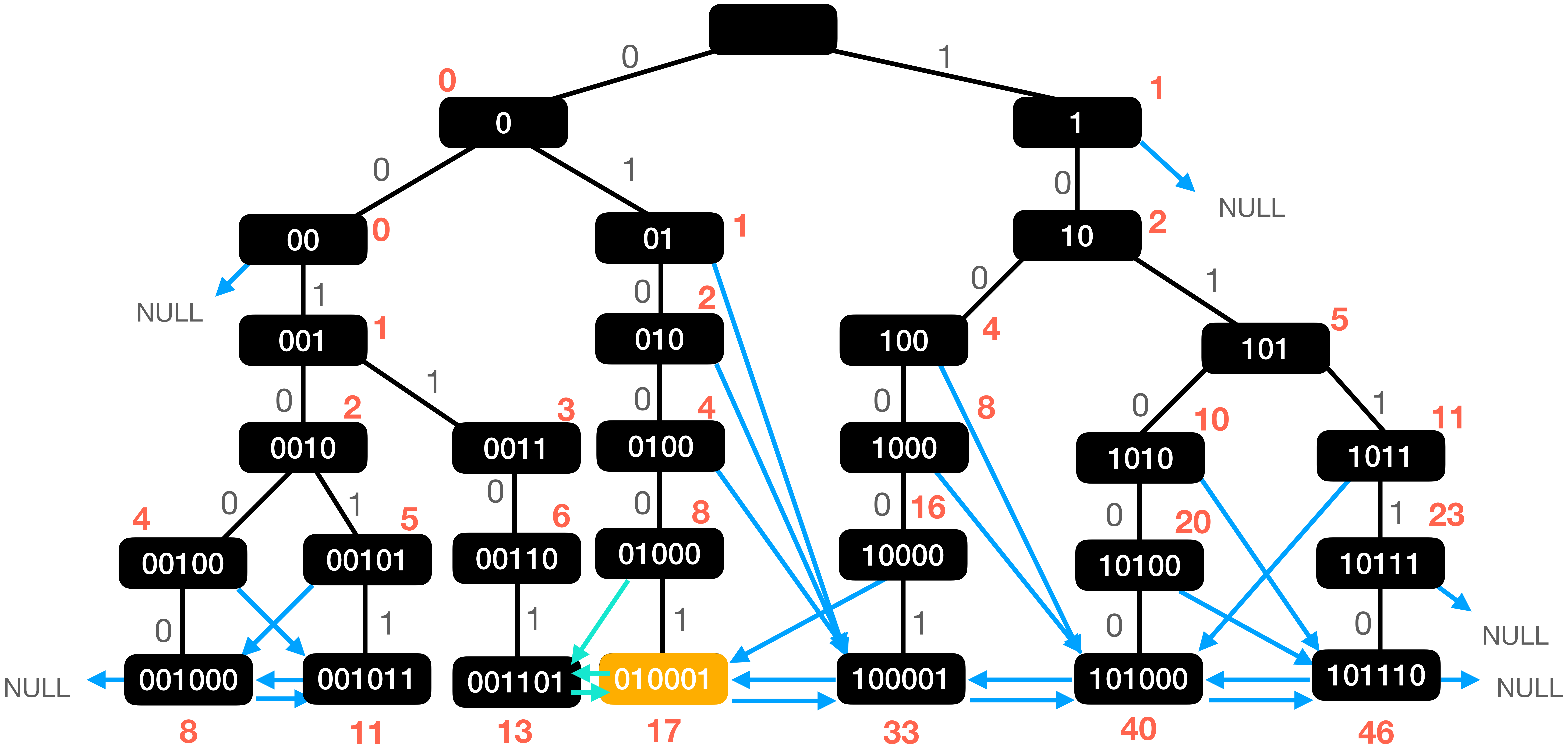
3. Walk backwards from predecessor, successor, and x to update skip pointers.



Insert

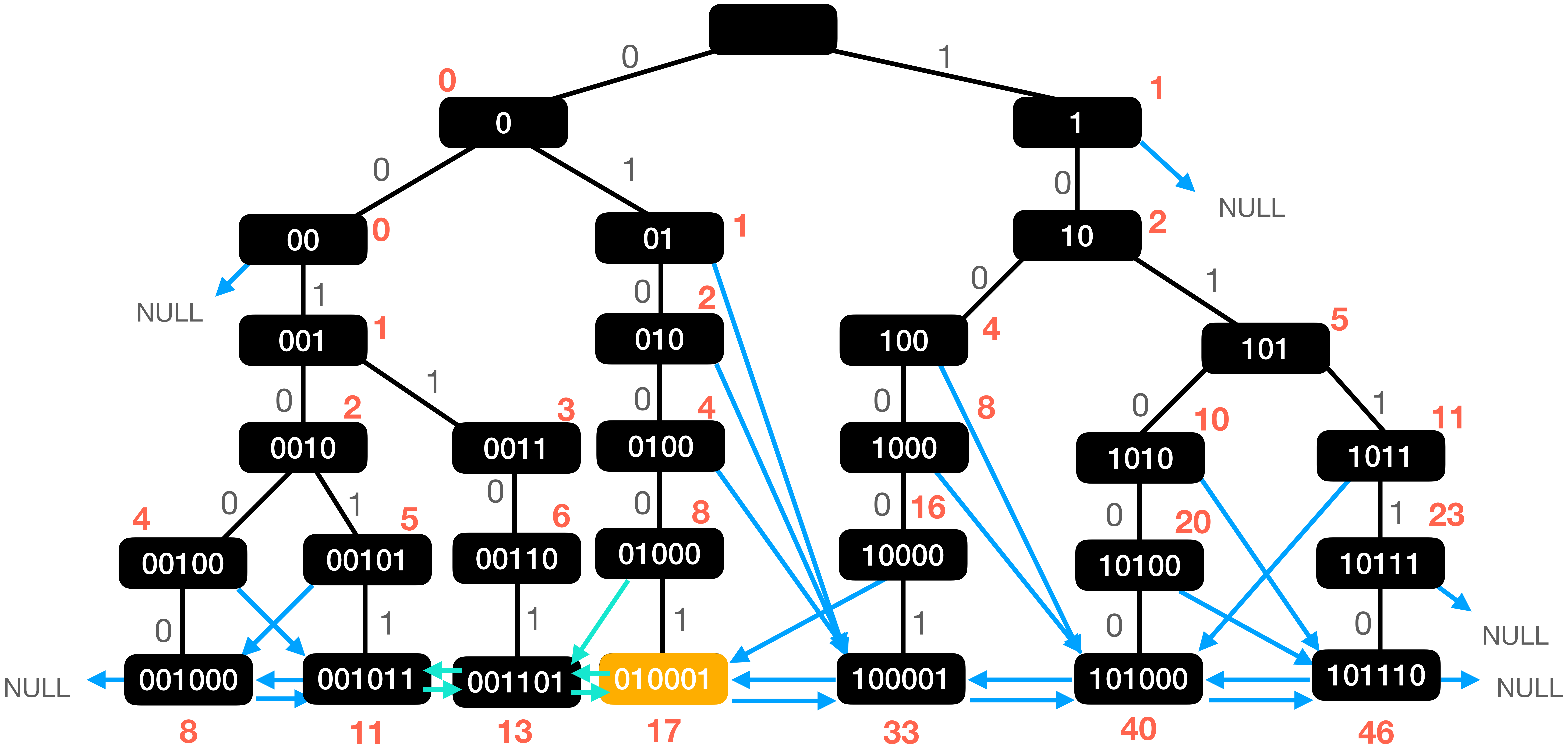
Insert(13), $[13]_2 = 001101$: 1. find Successor(13); 2. add x to the trie;

3. Walk backwards from predecessor, successor, and x to update skip pointers.



Insert

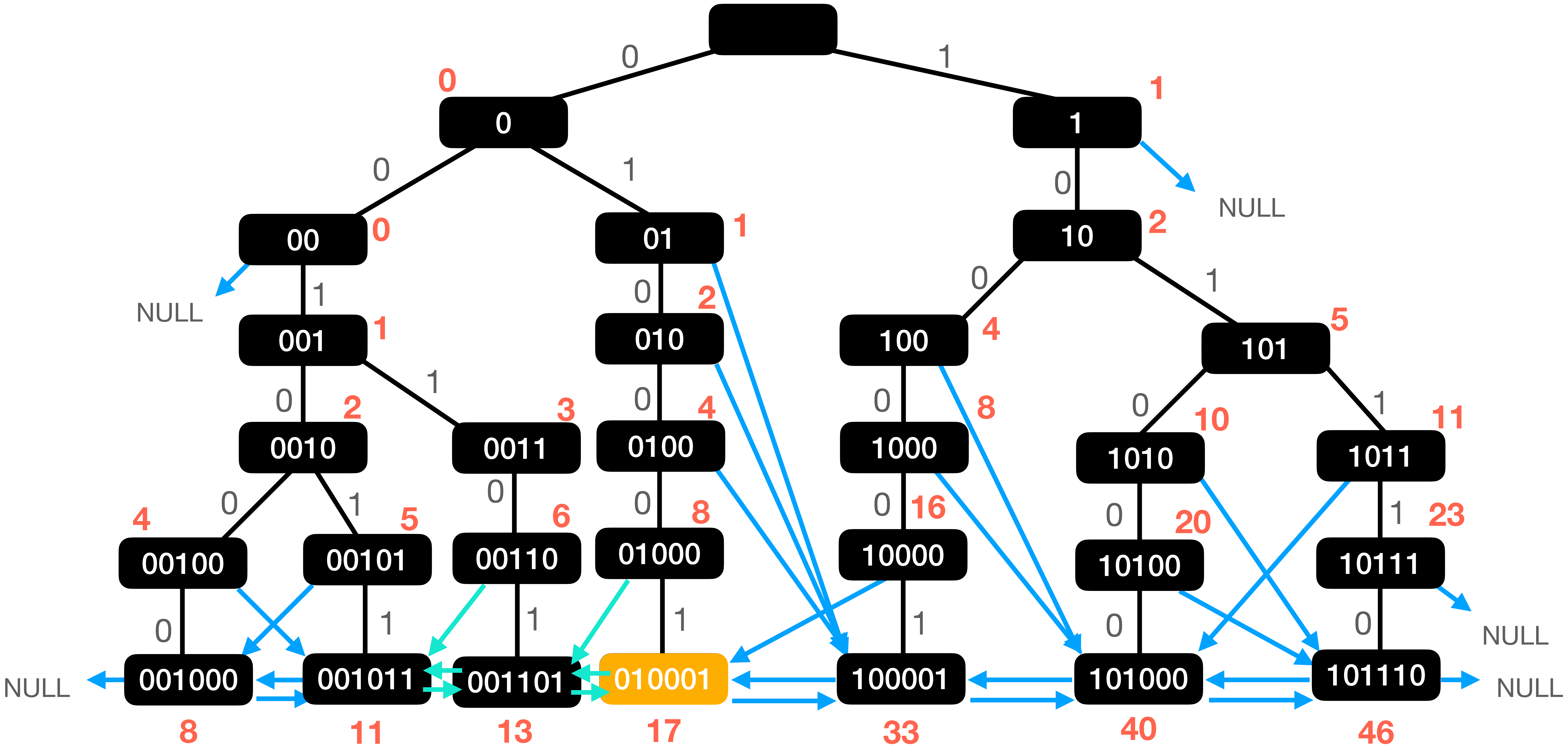
Insert(13), $[13]_2 = 001101$: **1**. find Successor(13); **2**. add x to the trie;
3. Walk backwards from predecessor, successor, and x to update skip pointers.



Insert

Insert(13), $[13]_2 = 001101$: 1. find Successor(13); 2. add x to the trie;

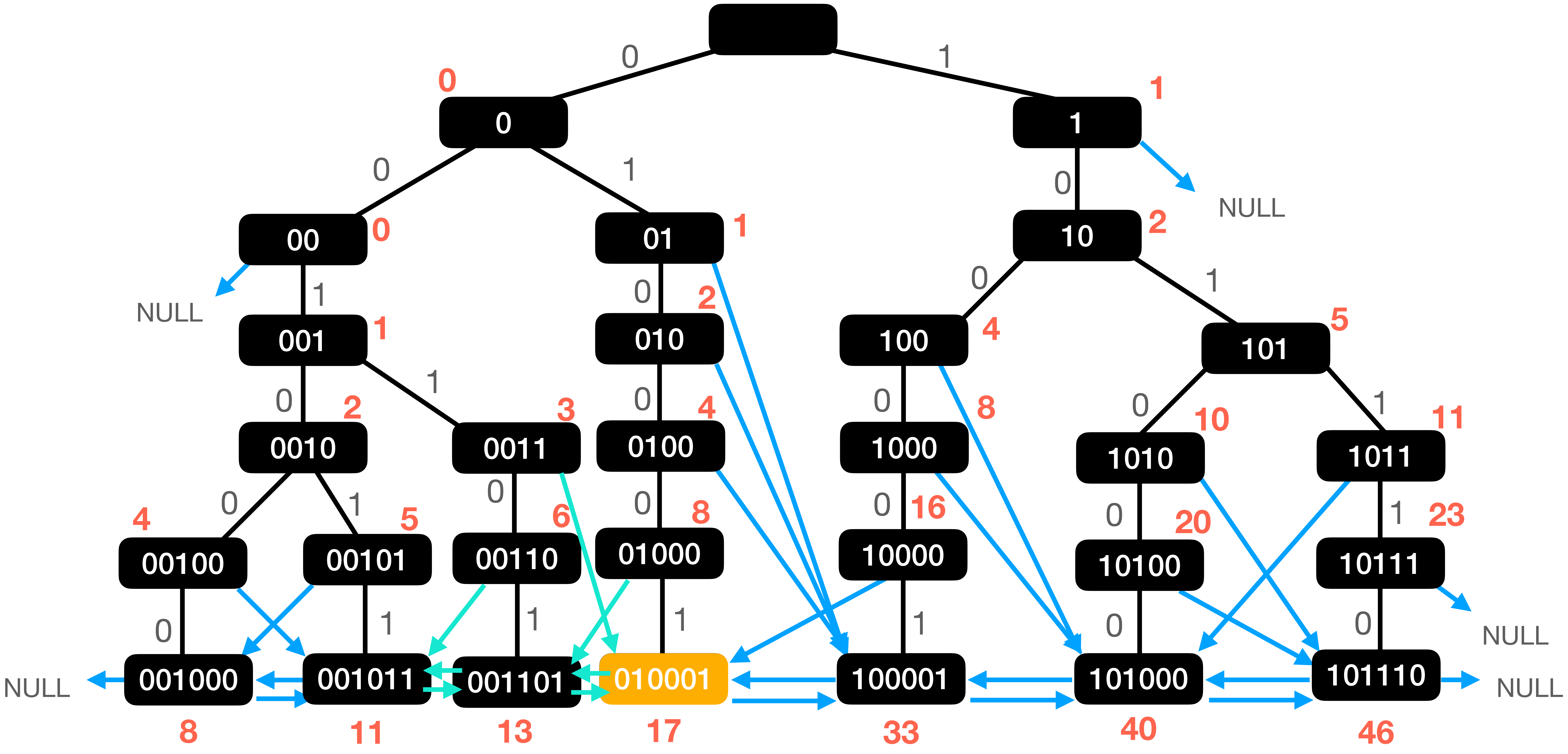
3. Walk backwards from predecessor, successor, and x to update skip pointers.



Insert

Insert(13), $[13]_2 = 001101$: 1. find Successor(13); 2. add x to the trie;

3. Walk backwards from predecessor, successor, and x to update skip pointers.

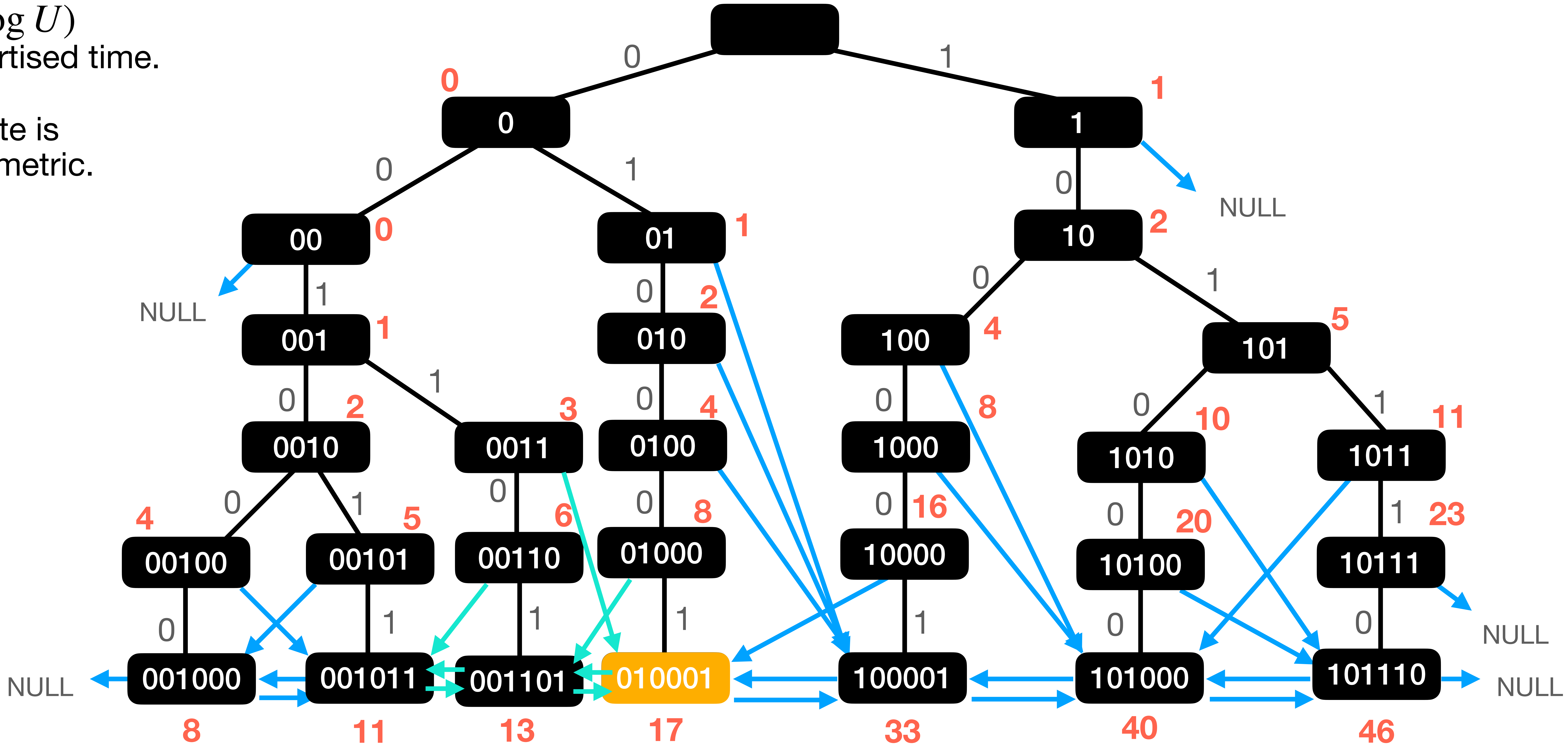


Insert

Insert(13), $[13]_2 = 001101$: **1**. find Successor(13); **2**. add x to the trie;

3. Walk backwards from predecessor, successor, and x to update skip pointers.

- $O(\log U)$ amortised time.
- Delete is symmetric.



The *x*-fast trie

- Min/Max: $O(1)$.
- Member: $O(1)$.
- Predecessor/Successor in $O(\log \log U)$.
- Subset(ℓ, r) in $O(\log \log U + |\text{Subset}(\ell, r)|)$.
- Insert/Delete in $O(\log U)$.
- Space $O(n \log U)$.

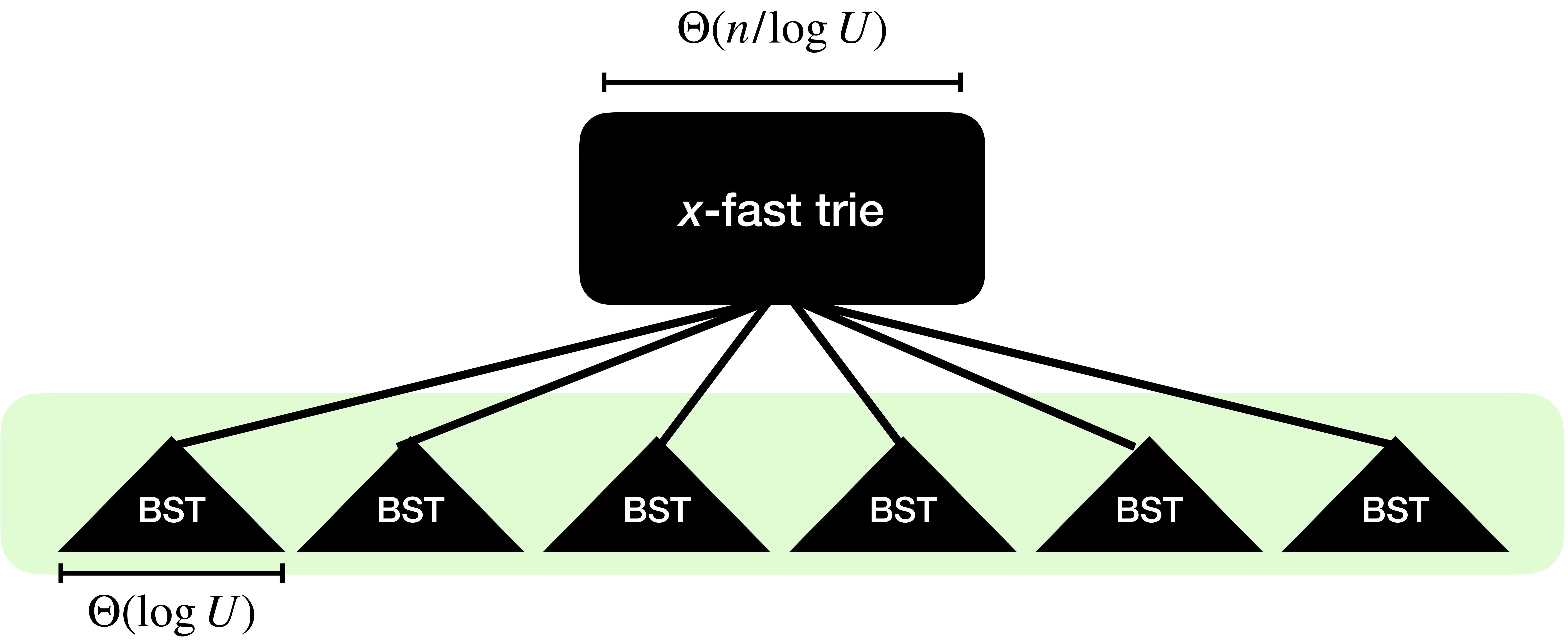
The *x*-fast trie

- Min/Max: $O(1)$.
- Member: $O(1)$.
- Predecessor/Successor in $O(\log \log U)$.
- Subset(ℓ, r) in $O(\log \log U + |\text{Subset}(\ell, r)|)$.
- Insert/Delete in $O(\log U)$.
- Space $O(n \log U)$.

This is what currently looks unsatisfactory…

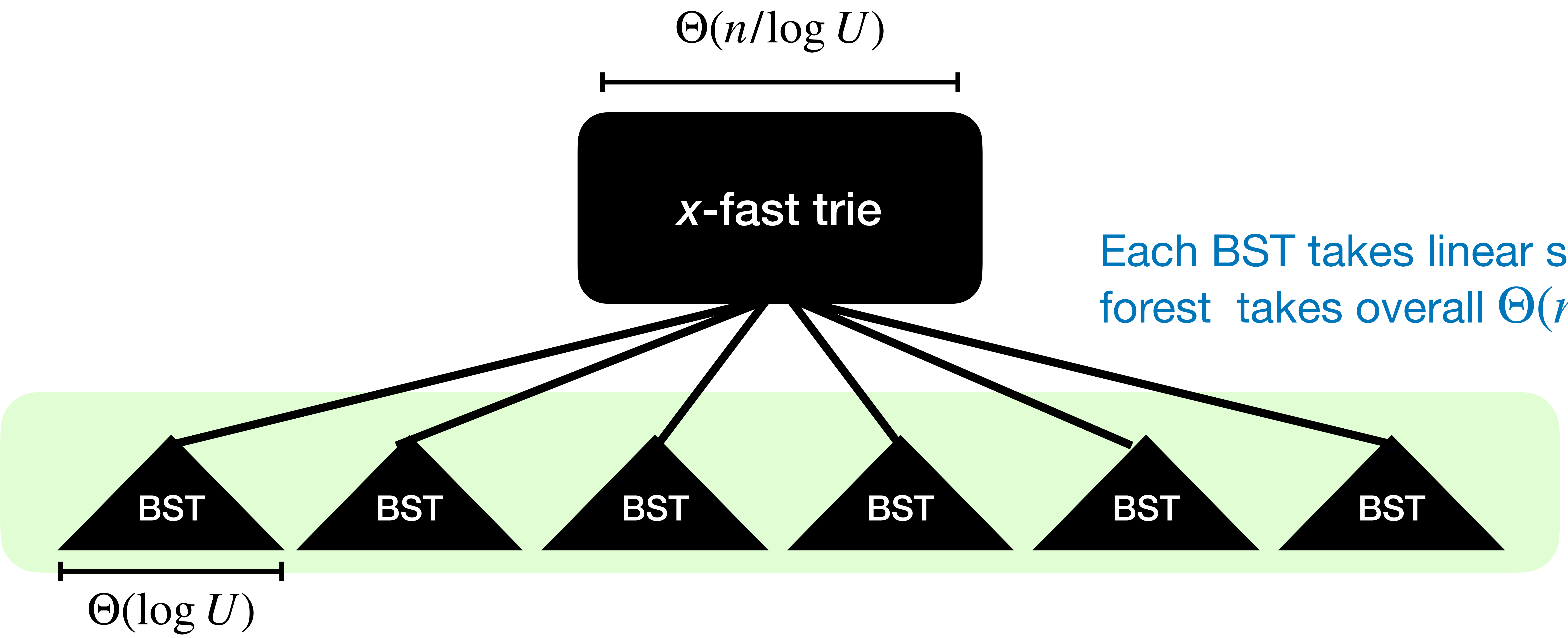
The *y*-fast trie

- **Idea.** Chunking again! Split the set of keys into (mini) chunks of size $\Theta(\log U)$ and store each chunk in a balanced BST. Choose one representative per chunk. The set of $\Theta(n/\log U)$ representatives is stored with a *x*-fast trie.
- Use the *x*-fast trie as a *router* to identify the proper BST.



The *y*-fast trie

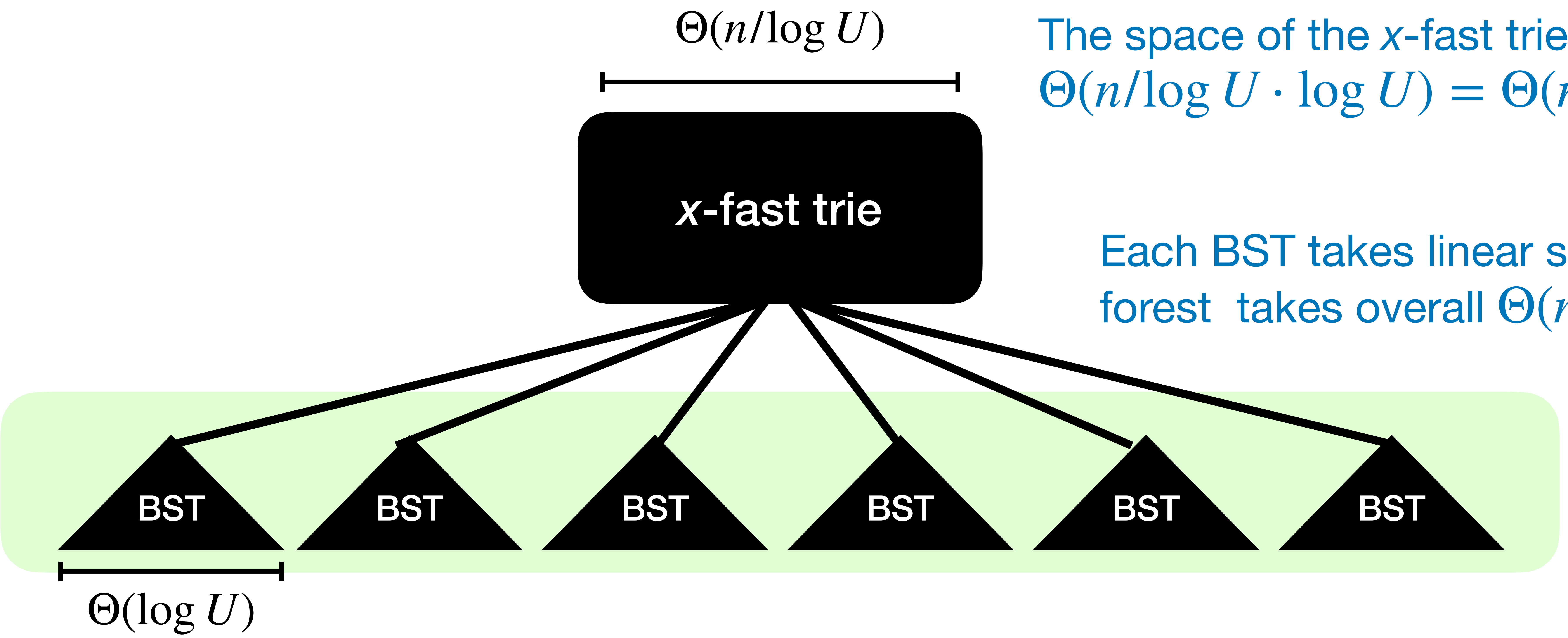
- **Idea.** Chunking again! Split the set of keys into (mini) chunks of size $\Theta(\log U)$ and store each chunk in a balanced BST. Choose one representative per chunk. The set of $\Theta(n/\log U)$ representatives is stored with a *x*-fast trie.
- Use the *x*-fast trie as a *router* to identify the proper BST.



Each BST takes linear space, so the forest takes overall $\Theta(n)$ space.

The *y*-fast trie

- **Idea.** Chunking again! Split the set of keys into (mini) chunks of size $\Theta(\log U)$ and store each chunk in a balanced BST. Choose one representative per chunk. The set of $\Theta(n/\log U)$ representatives is stored with a *x*-fast trie.
- Use the *x*-fast trie as a *router* to identify the proper BST.



The space of the *x*-fast trie is $\Theta(n/\log U \cdot \log U) = \Theta(n)$.

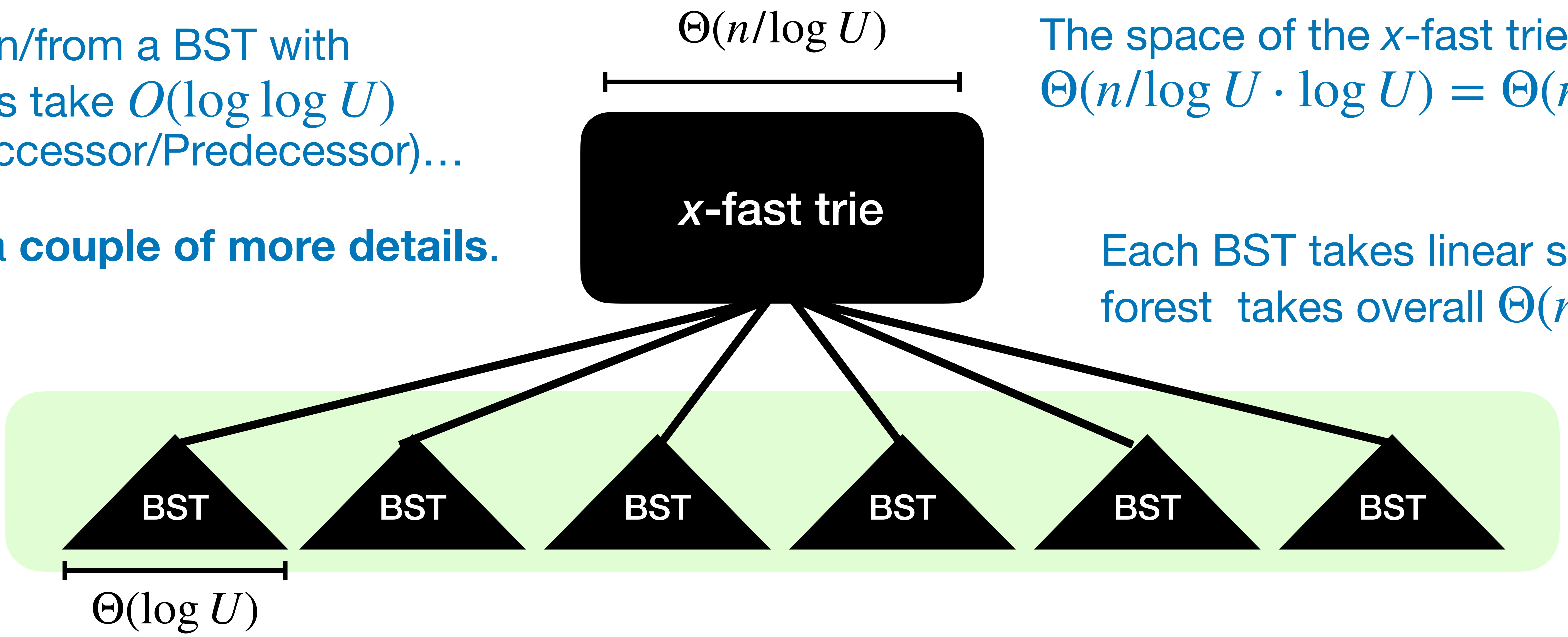
Each BST takes linear space, so the forest takes overall $\Theta(n)$ space.

The *y*-fast trie

- **Idea.** Chunking again! Split the set of keys into (mini) chunks of size $\Theta(\log U)$ and store each chunk in a balanced BST. Choose one representative per chunk. The set of $\Theta(n/\log U)$ representatives is stored with a *x*-fast trie.
- Use the *x*-fast trie as a *router* to identify the proper BST.

Insert/Delete in/from a BST with $\Theta(\log U)$ keys take $O(\log\log U)$ (as well as Successor/Predecessor)…

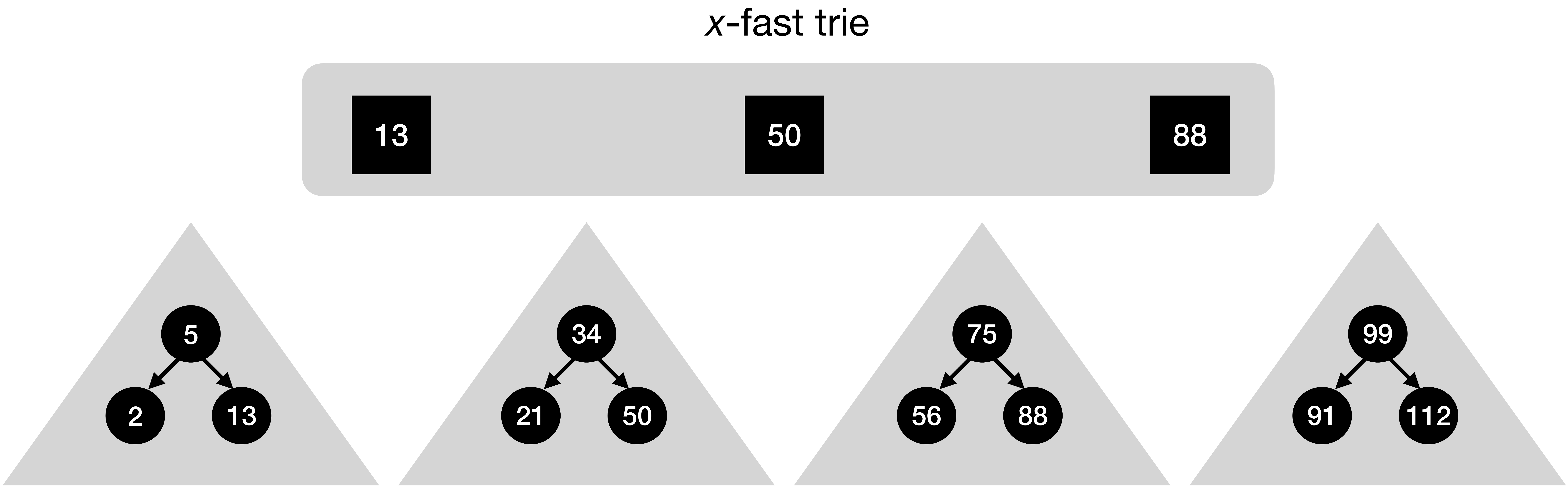
But we need a **couple of more details.**



The space of the *x*-fast trie is $\Theta(n/\log U \cdot \log U) = \Theta(n)$.

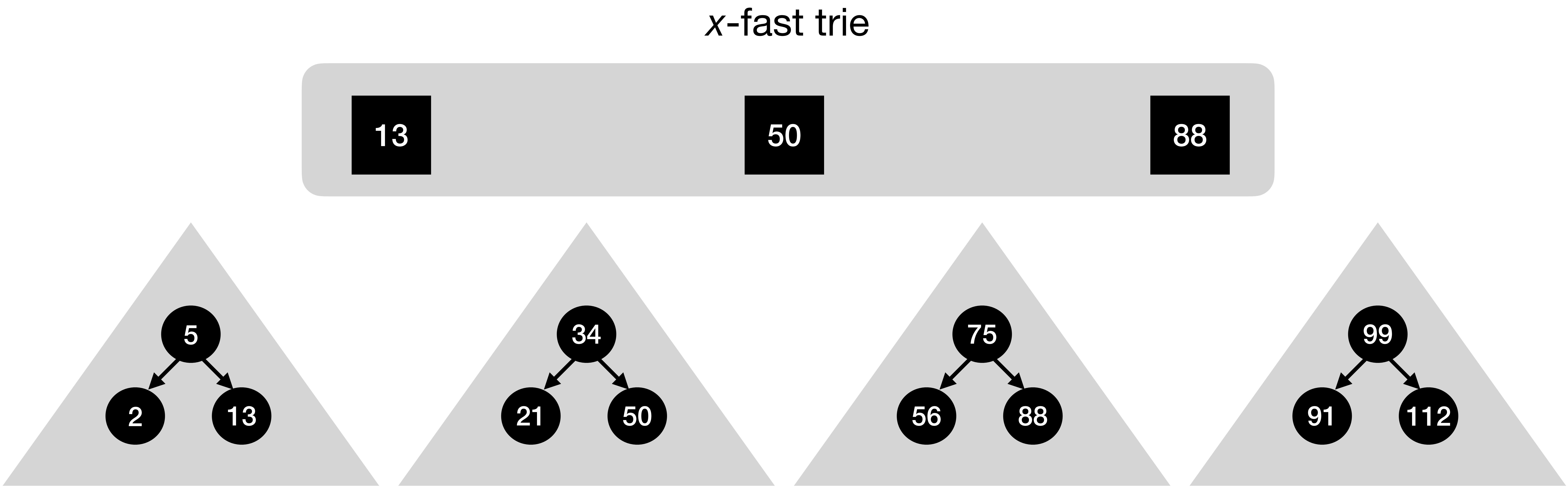
Each BST takes linear space, so the forest takes overall $\Theta(n)$ space.

The *y*-fast trie — Successor and Predecessor



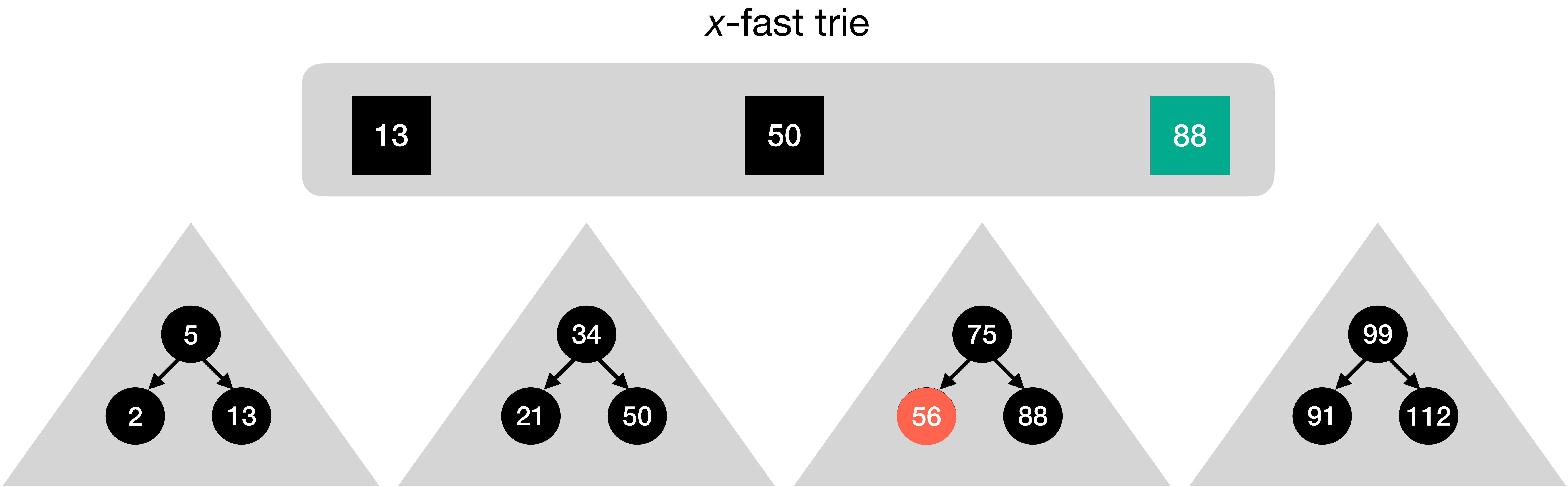
The *y*-fast trie — Successor and Predecessor

Successor(50)



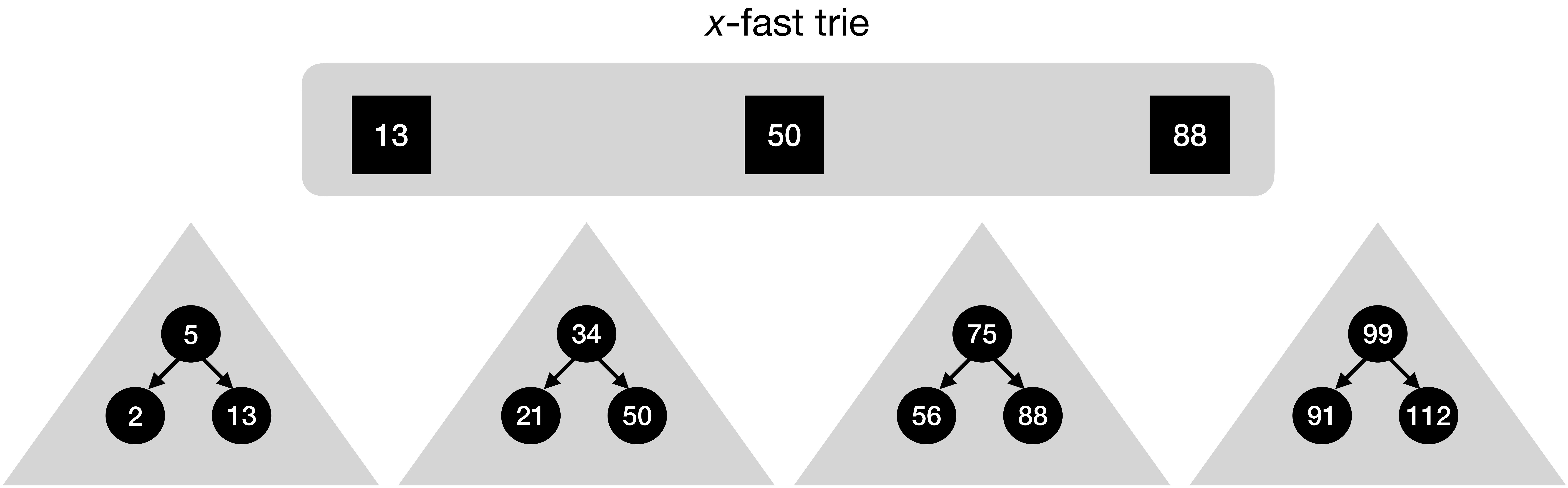
The *y*-fast trie — Successor and Predecessor

Successor(50)



The *y*-fast trie — Successor and Predecessor

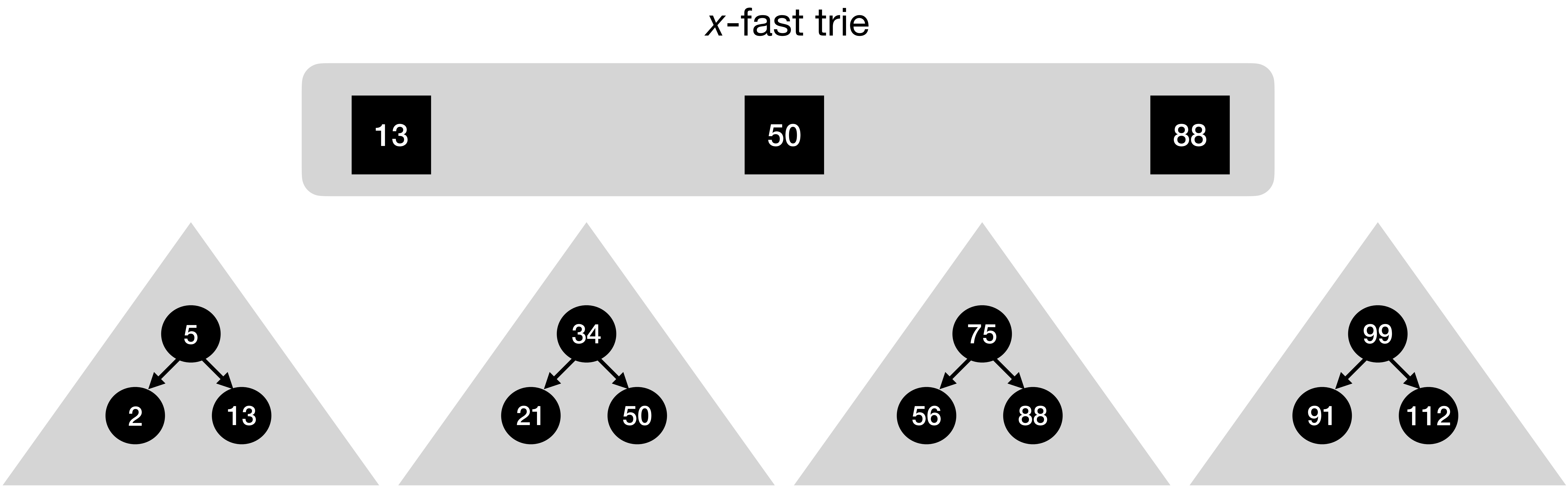
Successor(50)



The *y*-fast trie — Successor and Predecessor

Successor(50)

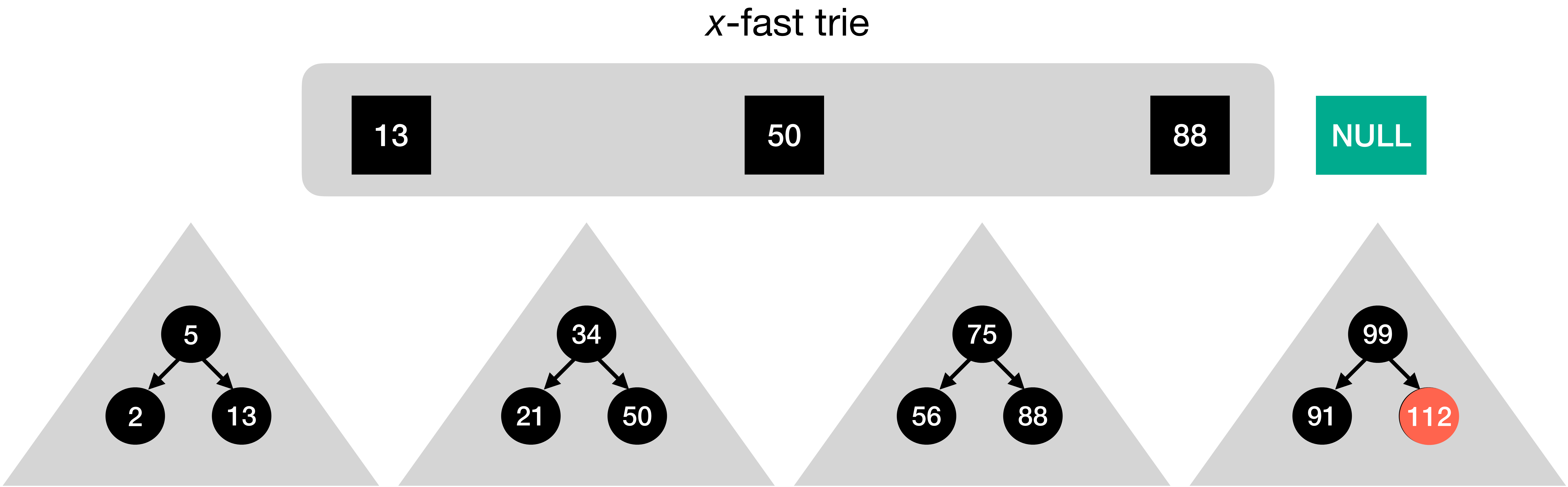
Successor(101)



The *y*-fast trie — Successor and Predecessor

Successor(50)

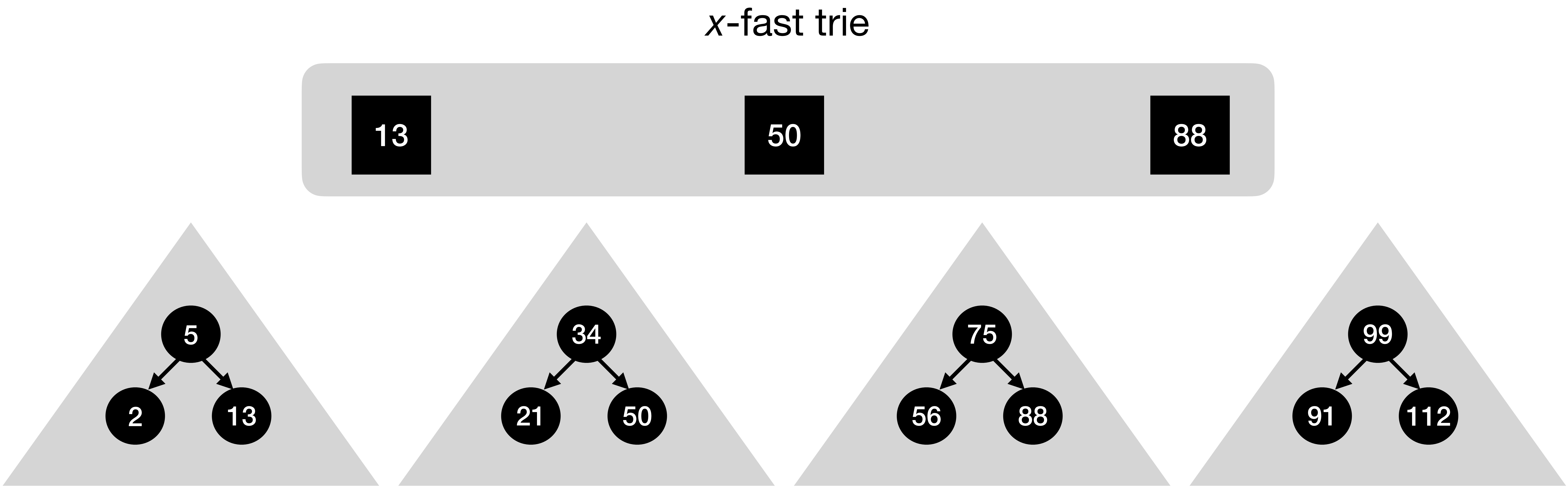
Successor(101)



The *y*-fast trie — Successor and Predecessor

Successor(50)

Successor(101)

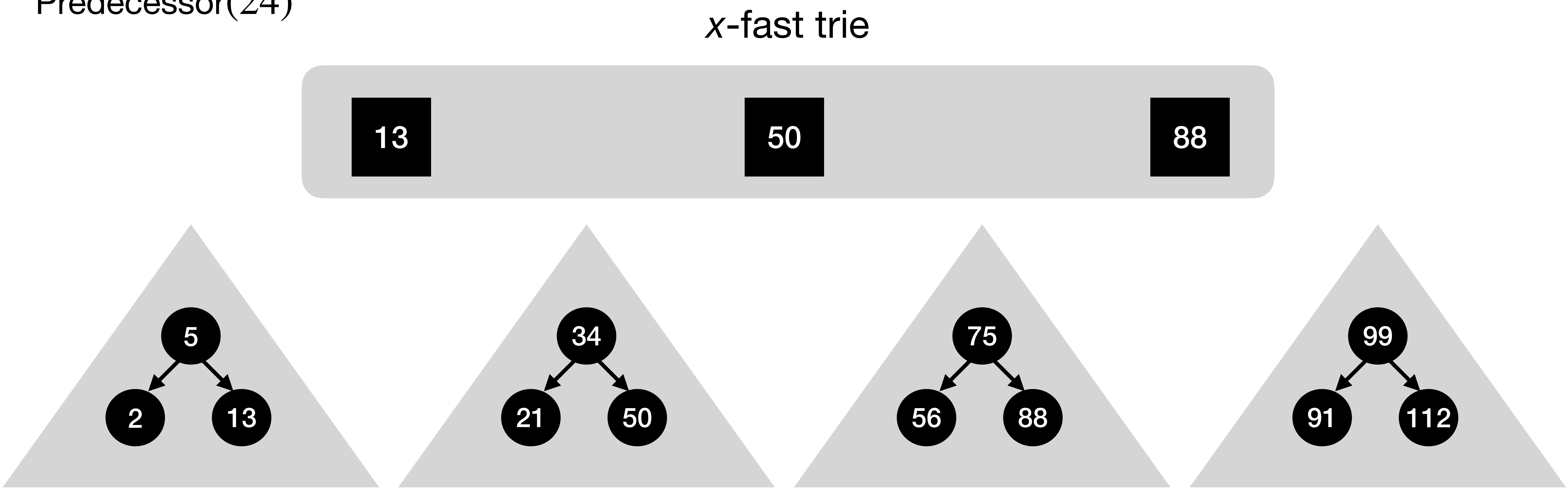


The *y*-fast trie — Successor and Predecessor

Successor(50)

Successor(101)

Predecessor(24)

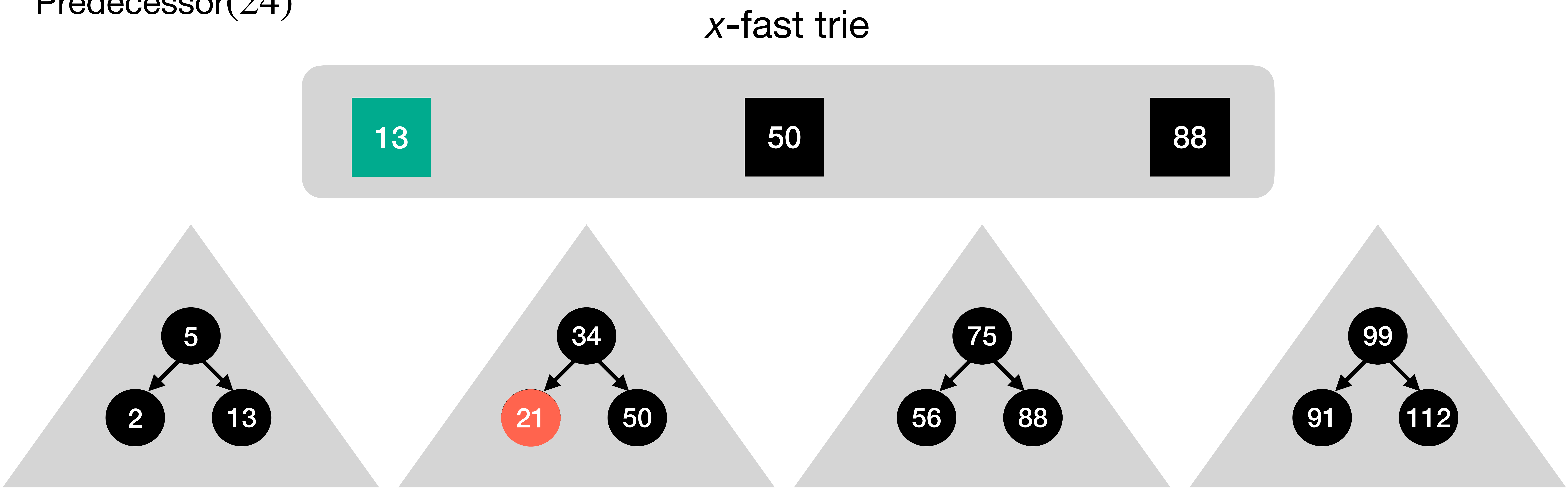


The *y*-fast trie — Successor and Predecessor

Successor(50)

Successor(101)

Predecessor(24)

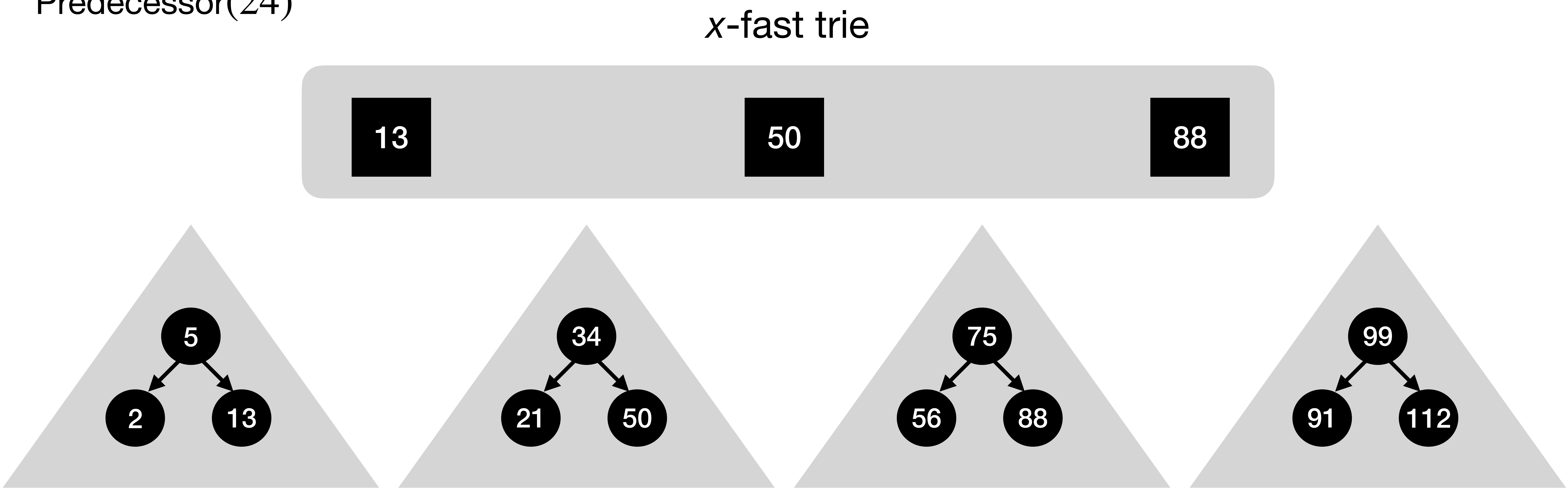


The *y*-fast trie — Successor and Predecessor

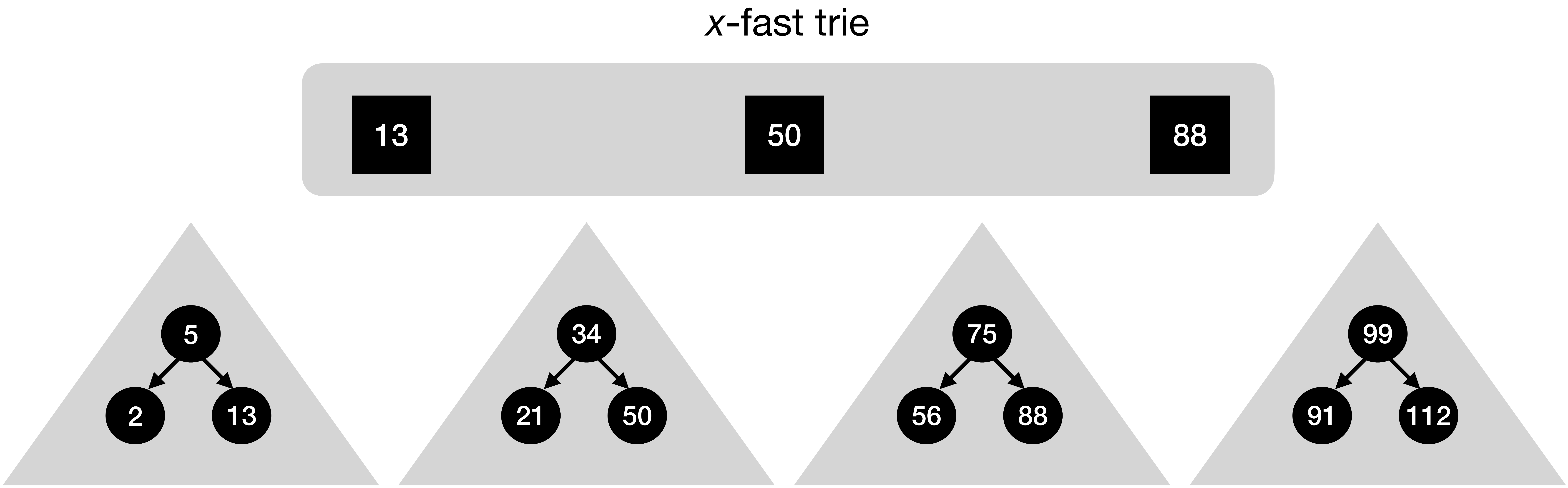
Successor(50)

Successor(101)

Predecessor(24)



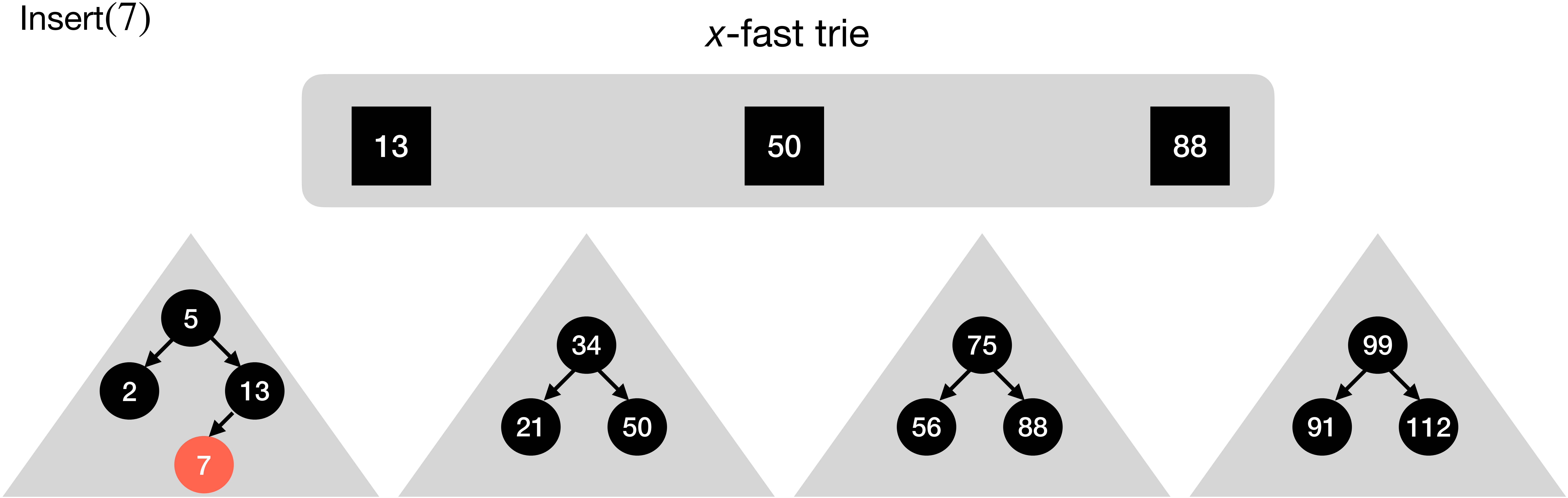
The *y*-fast trie — Insert and Delete



- Maintain the invariant that each BST contains at least $(\log_2 U)/c$ keys and at most $c \log_2 U$ keys, for some constant $c > 1$ (c controls a trade-off between queries and updates).
- **Split (on the median element) and merge trees** to maintain the invariant in $O(\log \log U)$ time. (This needs to augment the internal nodes of the trees with the size of their subtrees.)
- Update the *x*-fast trie accordingly in $O(\log U)$ time.

The *y*-fast trie — Insert and Delete

Insert(7)

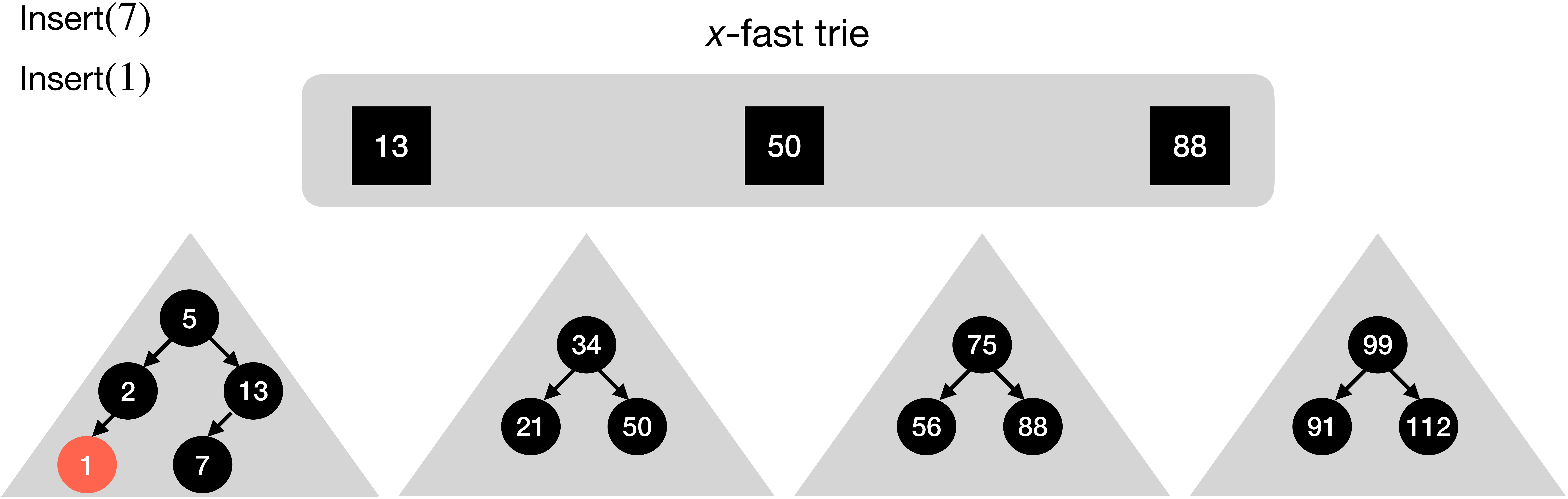


- Maintain the invariant that each BST contains at least $(\log_2 U)/c$ keys and at most $c \log_2 U$ keys, for some constant $c > 1$ (c controls a trade-off between queries and updates).
- **Split (on the median element) and merge trees** to maintain the invariant in $O(\log \log U)$ time. (This needs to augment the internal nodes of the trees with the size of their subtrees.)
- Update the *x*-fast trie accordingly in $O(\log U)$ time.

The *y*-fast trie — Insert and Delete

Insert(7)

Insert(1)



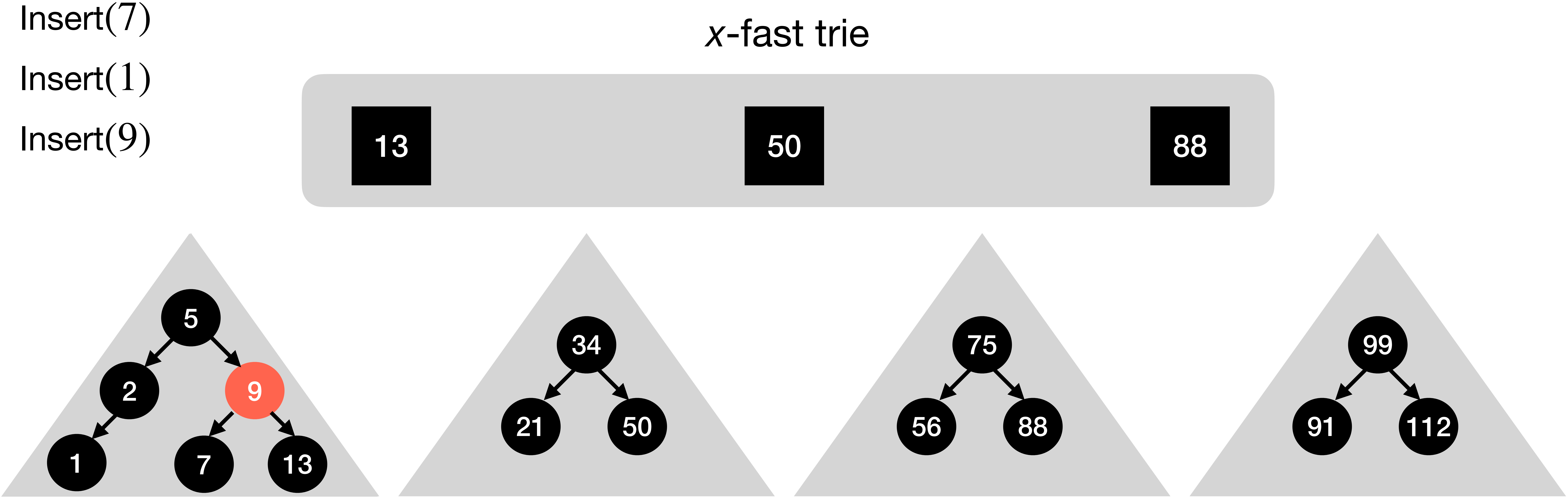
- Maintain the invariant that each BST contains at least $(\log_2 U)/c$ keys and at most $c \log_2 U$ keys, for some constant $c > 1$ (c controls a trade-off between queries and updates).
- **Split (on the median element) and merge trees** to maintain the invariant in $O(\log \log U)$ time. (This needs to augment the internal nodes of the trees with the size of their subtrees.)
- Update the *x*-fast trie accordingly in $O(\log U)$ time.

The *y*-fast trie — Insert and Delete

Insert(7)

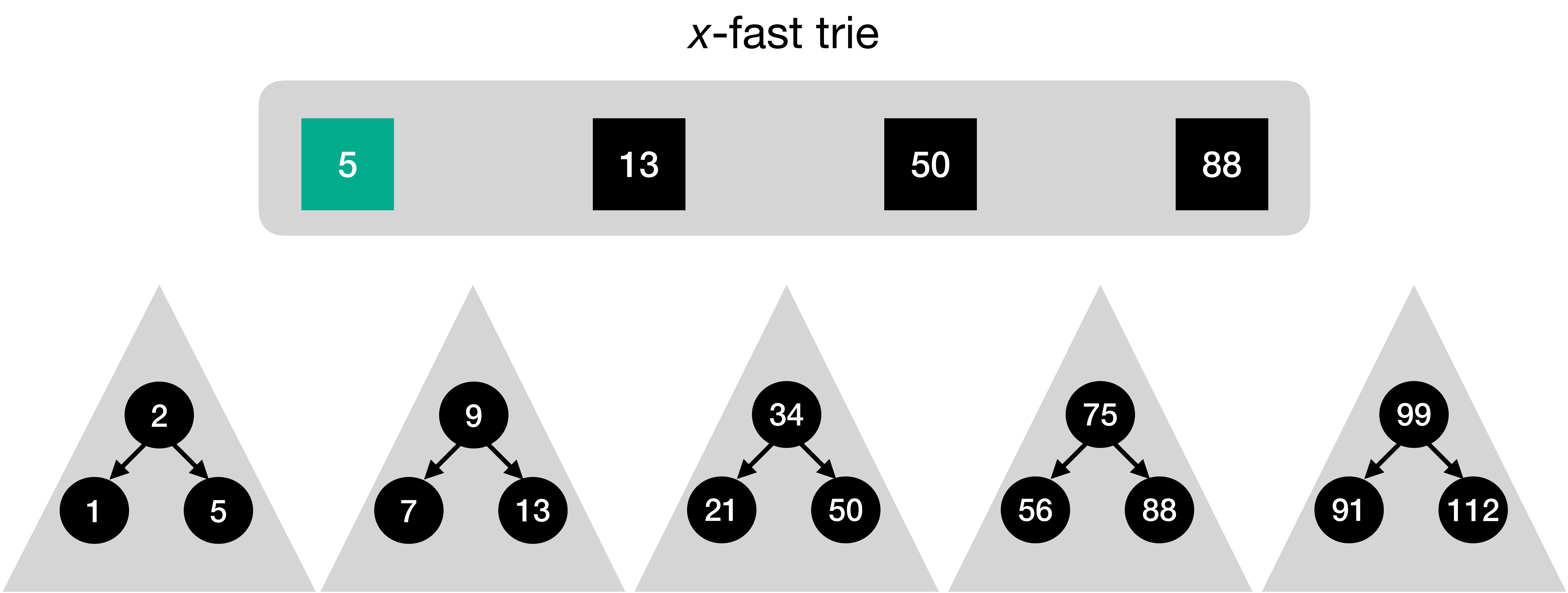
Insert(1)

Insert(9)



- Maintain the invariant that each BST contains at least $(\log_2 U)/c$ keys and at most $c \log_2 U$ keys, for some constant $c > 1$ (c controls a trade-off between queries and updates).
- **Split (on the median element) and merge trees** to maintain the invariant in $O(\log \log U)$ time. (This needs to augment the internal nodes of the trees with the size of their subtrees.)
- Update the *x*-fast trie accordingly in $O(\log U)$ time.

The *y*-fast trie — Insert and Delete



- Maintain the invariant that each BST contains at least $(\log_2 U)/c$ keys and at most $c \log_2 U$ keys, for some constant $c > 1$ (c controls a trade-off between queries and updates).
- **Split (on the median element) and merge trees** to maintain the invariant in $O(\log \log U)$ time. (This needs to augment the internal nodes of the trees with the size of their subtrees.)
- Update the *x*-fast trie accordingly in $O(\log U)$ time.

The *y*-fast trie — Amortisation

- If we **do not split** upon Insert, we pay $O(\log \log U)$.
- If we do, we pay $O(\log U)$. But note that to split, we must have done $O(\log U)$ insertions to a tree, hence on average we pay:

$$\frac{O(\log U) \cdot O(\log \log U) + O(\log U)}{O(\log U)} = O(\log \log U).$$

- Logic for Delete: if a tree becomes too small, we merge it with the next tree. Split again the resulting tree if necessary and update the *x*-fast trie.

The *y*-fast trie — Amortisation

- If we **do not split** upon Insert, we pay $O(\log\log U)$.

- If we do, we pay $O(\log U)$. But note that to split, we must have done $O(\log U)$ insertions to a tree, hence on average we pay:

$$\frac{O(\log U)\cdot O(\log\log U)+O(\log U)}{O(\log U)}=O(\log\log U).$$

- Logic for Delete: if a tree becomes too small, we merge it with the next tree. Split again the resulting tree if necessary and update the *x*-fast trie.

- We have a **sorting algorithm** that runs in $O(n\log\log U)$ amortized-time by simply inserting each integer in a *y*-fast trie! Radix Sort takes $O(n\cdot(\log U)/r)$ time. Sorting with an *y*-fast trie is therefore better for $r=o(\log U/\log\log U)$.

Summary

	x-fast trie	*y*-fast trie
Min/Max	$O(1)$	$O(1)$
Member	$O(1)$	$O(\log \log U)$
Predecessor/Successor	$O(\log \log U)$	$O(\log \log U)$
Subset	$O(\log \log U + \lvert \text{Subset} \rvert)$	$O(\log \log U + \lvert \text{Subset} \rvert)$
Insert/Delete	$O(\log U)$	$O(\log \log U)$
Space	$O(n \log U)$	$\Theta(n)$
Main tools	Binary search on key's prefixes stored into a hash table.	Sparsification and amortisation. A suboptimal structure might be ok if used on few elements.

The last question is…

- Why the names “*x*” and “*y*” ?!
- Quote:

  “…the more you think about what the B in B-Tree means,
  the better you understand B-Trees!”

References

- Dan E. Willard. *Log-logarithmic worst-case range queries are possible in space $\Theta(N)$*, Information Processing Letters, 17 (2): 81-84, 1983.

- P. van Emde Boas, *Preserving order in a forest in less than logarithmic time*, FOCS, 75-84, 1975.

- P. van Emde Boas; R. Kaas; E. Zijlstra, *Design and implementation of an efficient priority queue*, Math. Syst. Theory, 99–127, 1977.

- P. van Emde Boas, *Preserving order in a forest in less than logarithmic time and linear space*, Inf. Process. Lett. 6, 80–82, 1977.