van Emde Boas Trees

Giulio Ermanno Pibiri
Ca' Foscari University of Venice

Venice, Italy, 3 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.

Problem definition

- **Problem.** Maintain a dynamic, sorted, integer set $S \subseteq \{0,\ldots,U-1\}$, for some U.
- Notes:
  - We assume U fits in $O(1)$ memory words.
  - S is a set, hence **no duplicates are allowed**.
  - Balanced binary search trees (like AVL trees and RB trees) solve the problem in $O(\log n)$ time per op/query, where $n = |S|$. This holds for general keys.

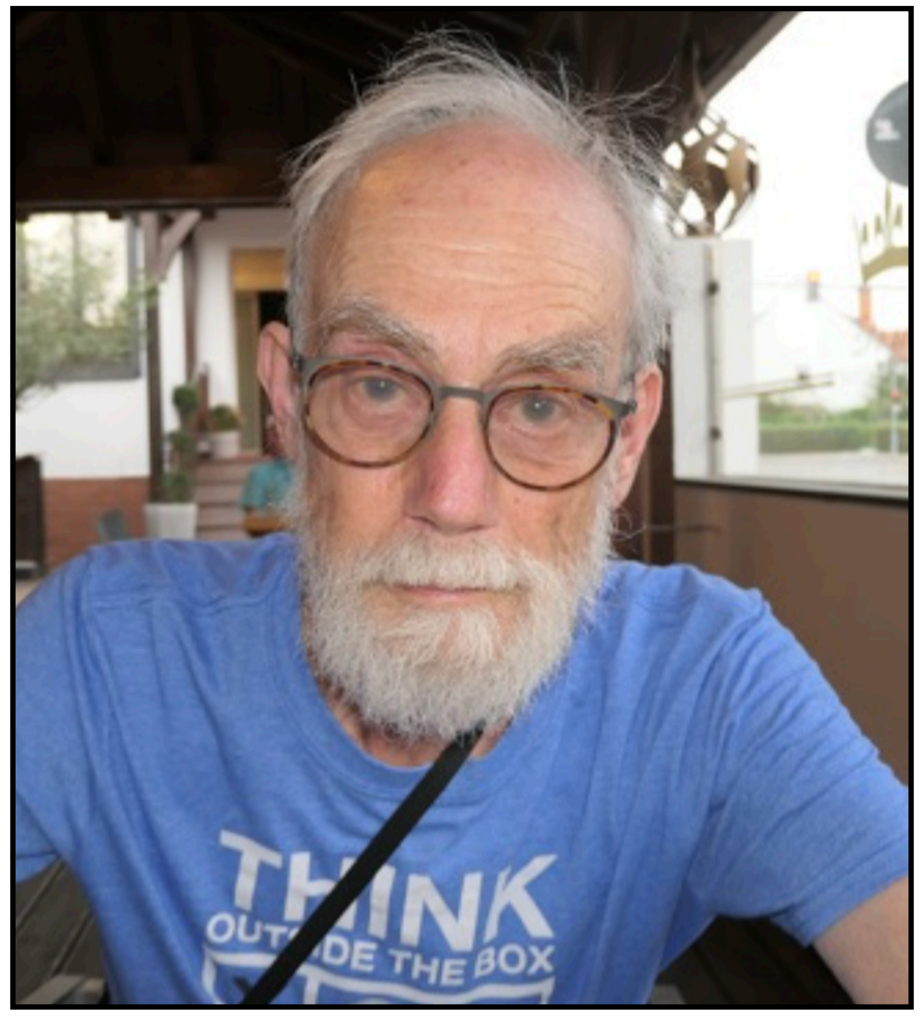
    Here, we deal with **integers from a bounded domain**.

Problem definition

- **Problem.** Maintain a dynamic, sorted, integer set $S \subseteq \{0,\ldots,U-1\}$, for some U.
- Notes:
  - We assume U fits in $O(1)$ memory words.
  - S is a set, hence **no duplicates are allowed**.
  - Balanced binary search trees (like AVL trees and RB trees) solve the problem in $O(\log n)$ time per op/query, where $n = |S|$. This holds for general keys.

    Here, we deal with **integers from a bounded domain**.

- The solution we are going to describe was first proposed by **Peter van Emde Boas** in 1975, and later refined in 1977.



Peter van Emde Boas

Plain bitvector

- **Idea.** Use a bitvector $B[0..U-1]$ of U bits where $B[x] = 1$ iff $x \in S$.
  (Sometimes called the *characteristic* bitvector of S.)

- Example for $U = 16$ and $S = \{2,4,7,13\}$.

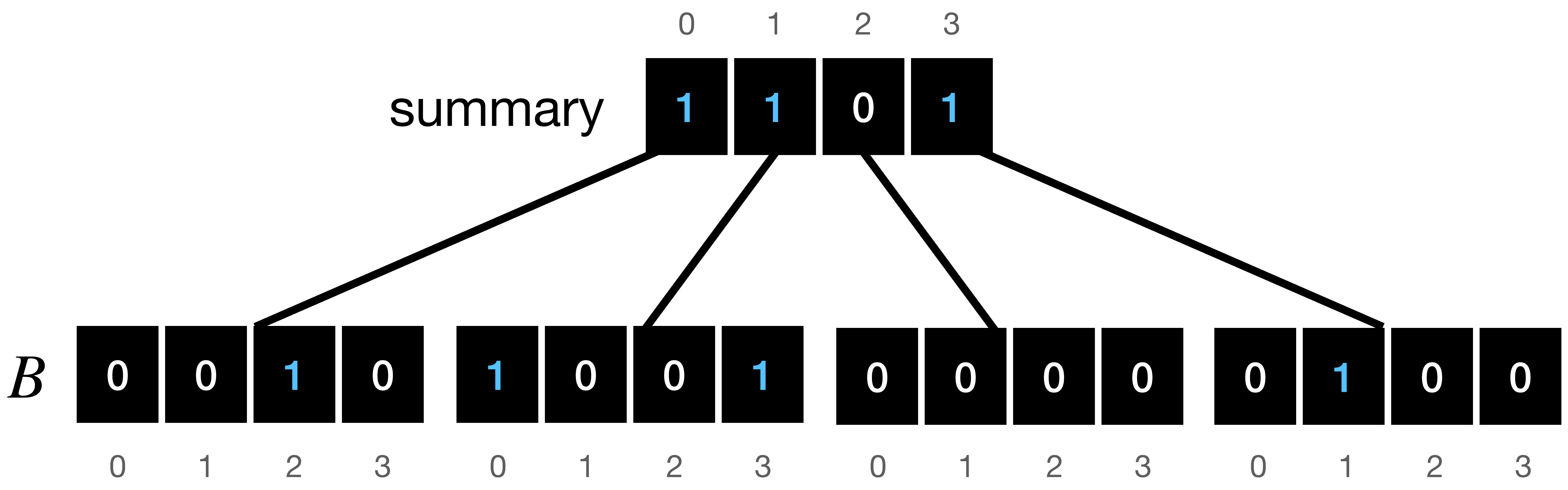
B	0	0	1	0	1	0	0	1	0	0	0	0	0	1	0	0
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

- Insert/Delete x: flip bit in position x.
- Member(x): check bit in position x.
- Runtime is $O(1)$.

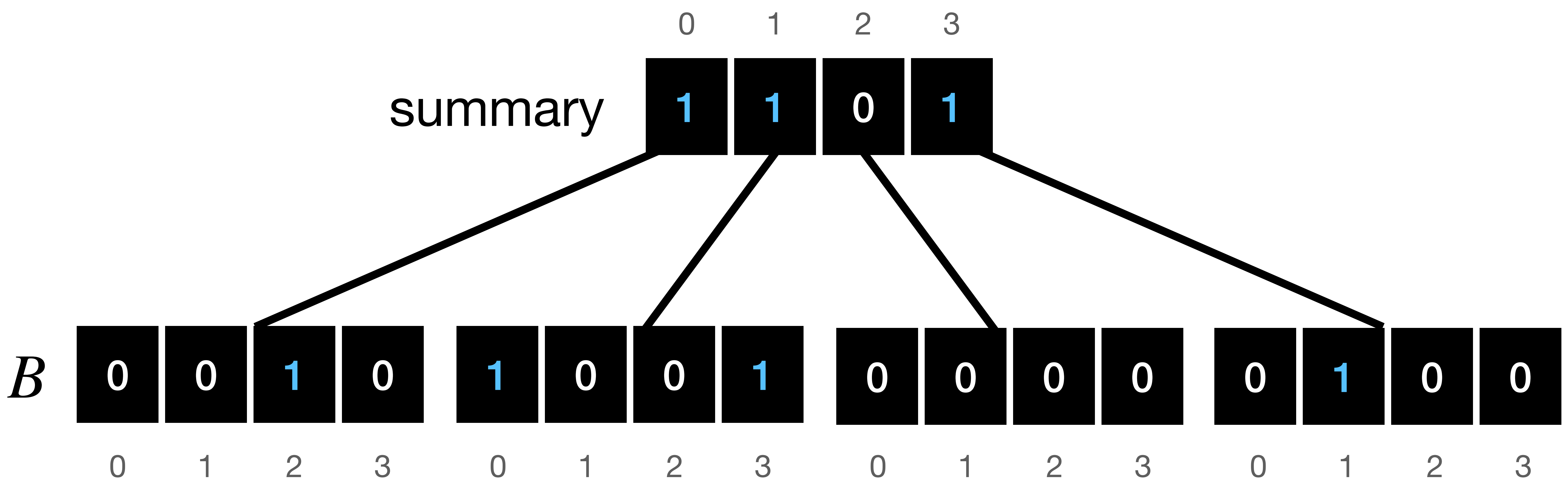
- But Min/Max, Predecessor/Successor are slow: $O(U)$ time by scanning B.

Chunking

- **Idea.** Split B into chunks of R bits. Define a new bitvector summary$[0..U/R-1]$ such that summary$[i] = 1$ iff chunk i is not empty. (We assume R divides U.)
- **Intuition.** Use the summary as a **shortcut**, avoiding to scan the entire bitvector.
- In general, ops/queries operate on the summary and at most two chunks: $O(U/R + R)$ time, which is minimized for $R = \sqrt{U}$.

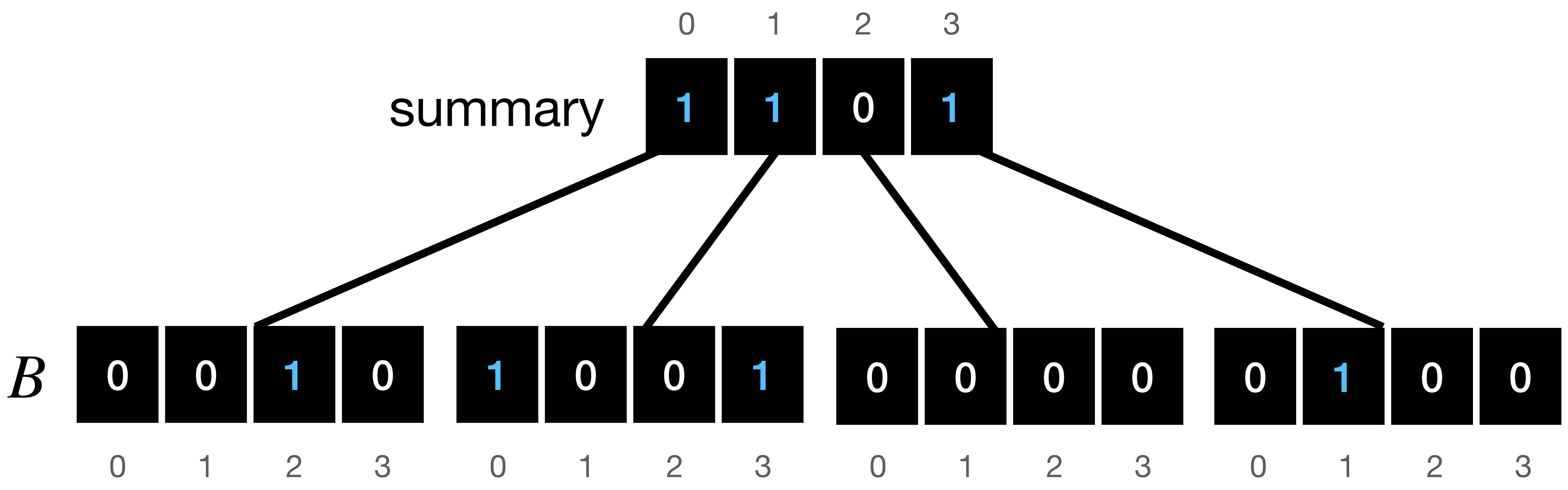


Chunking



Chunking

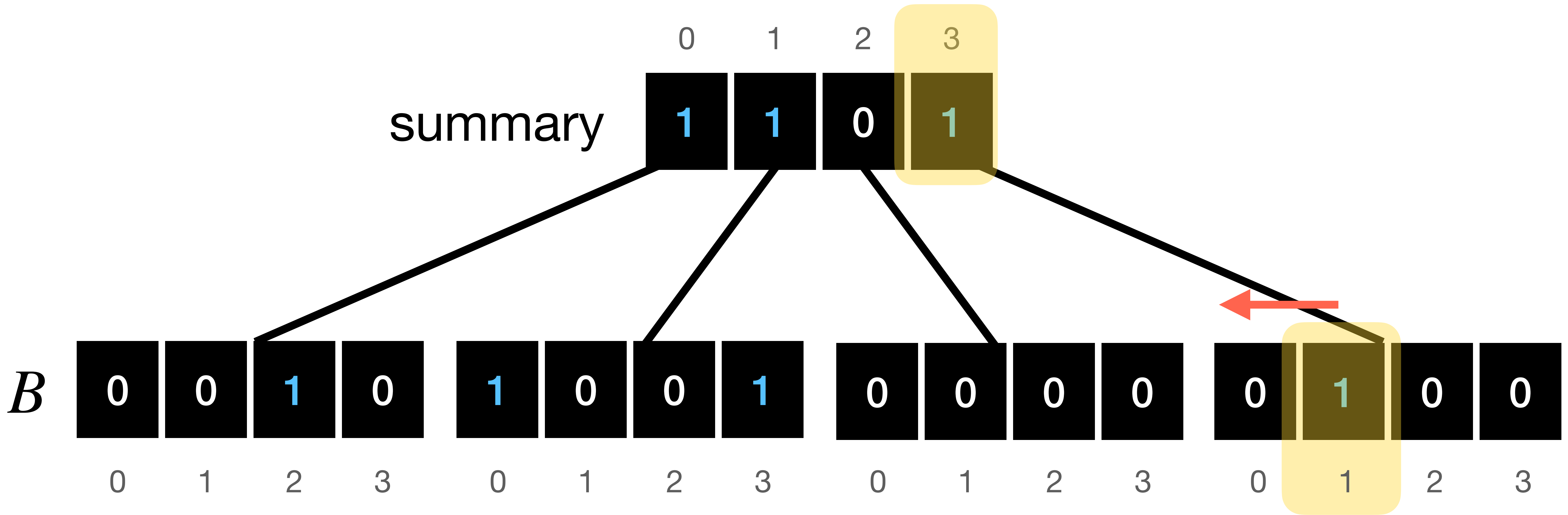
Predecessor(13):



Chunking

Predecessor(13):

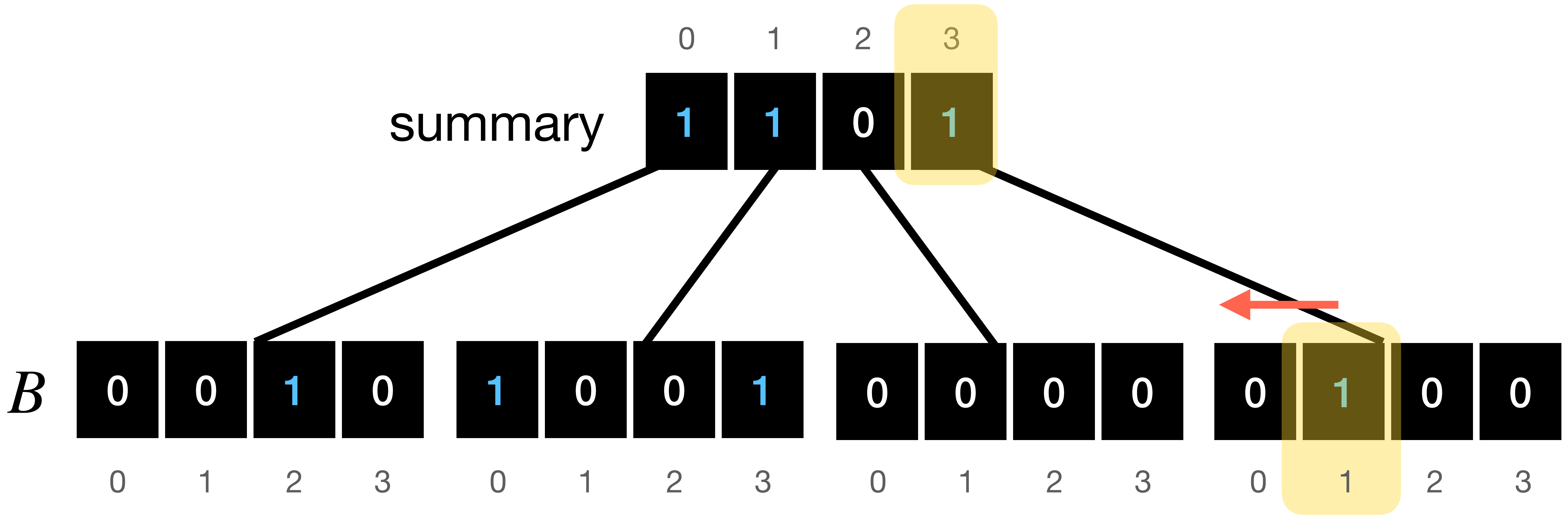
- Retrieve Min from chunk $\lfloor 13/4 \rfloor$.



Chunking

Predecessor(13):

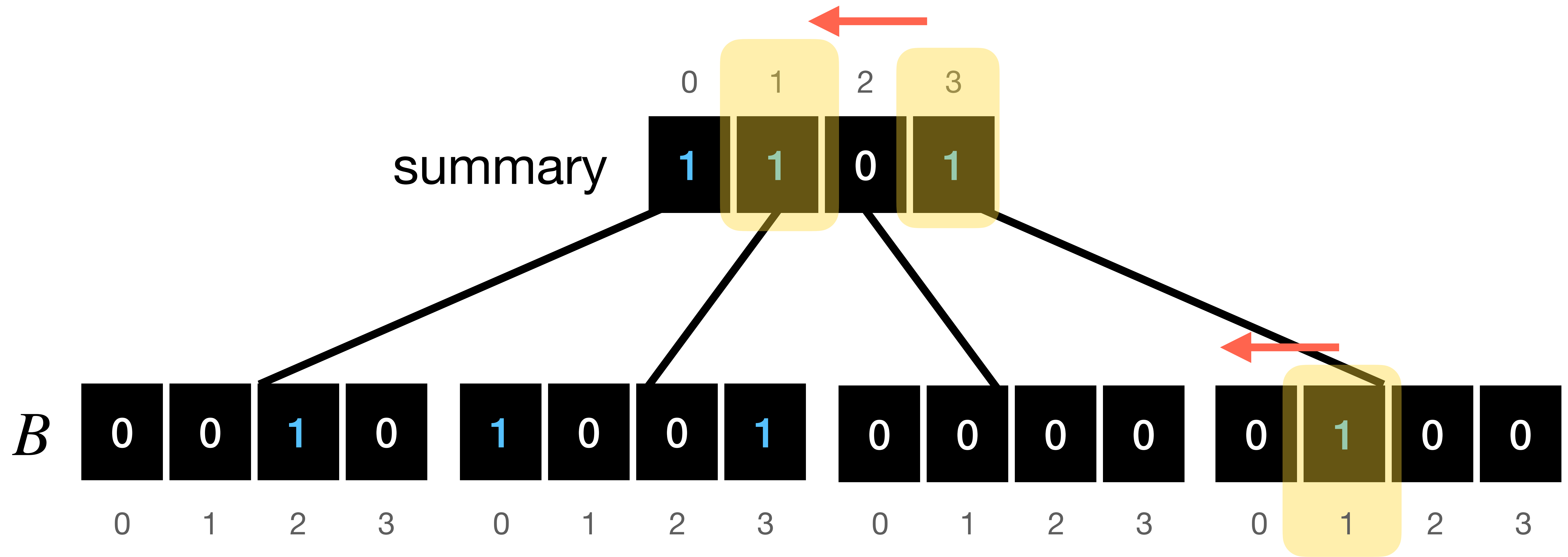
- Retrieve Min from chunk $\lfloor 13/4 \rfloor$.
- Is Min larger than $13 \bmod 4$?



Chunking

Predecessor(13):

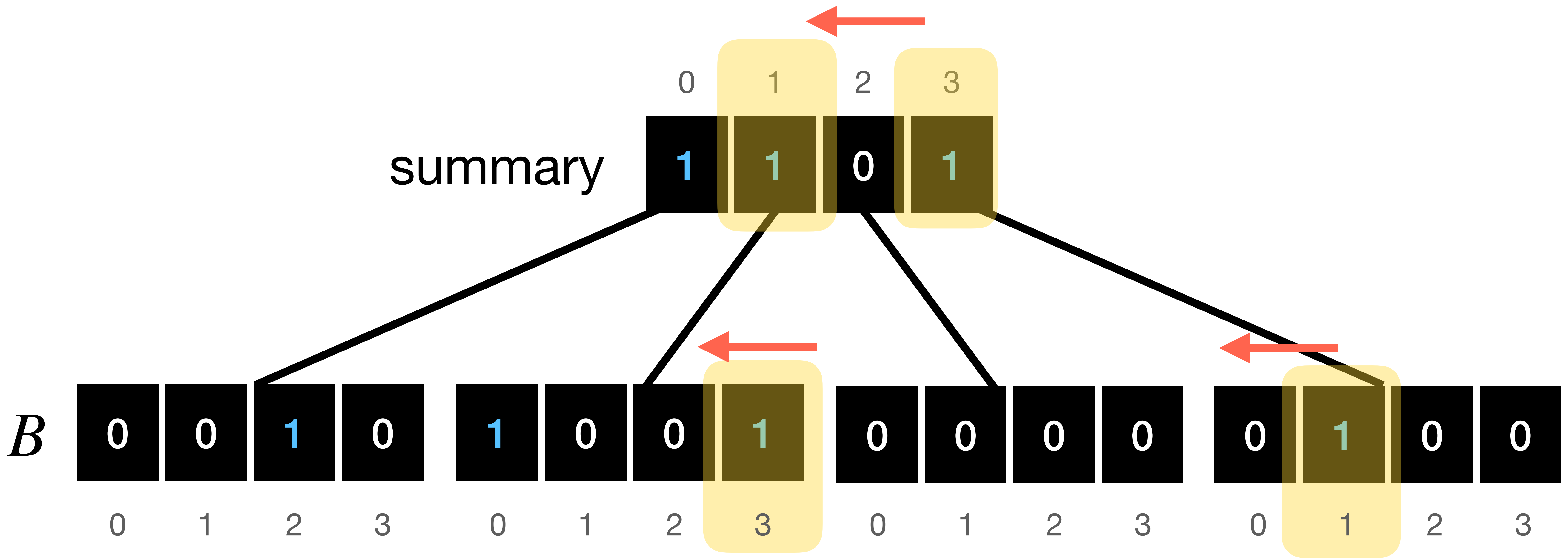
- Retrieve Min from chunk $\lfloor 13/4 \rfloor$.
- Is Min larger than $13 \bmod 4$?
- No, so answer Predecessor$(\lfloor 13/4 \rfloor) = 1$ on summary.



Chunking

Predecessor(13):

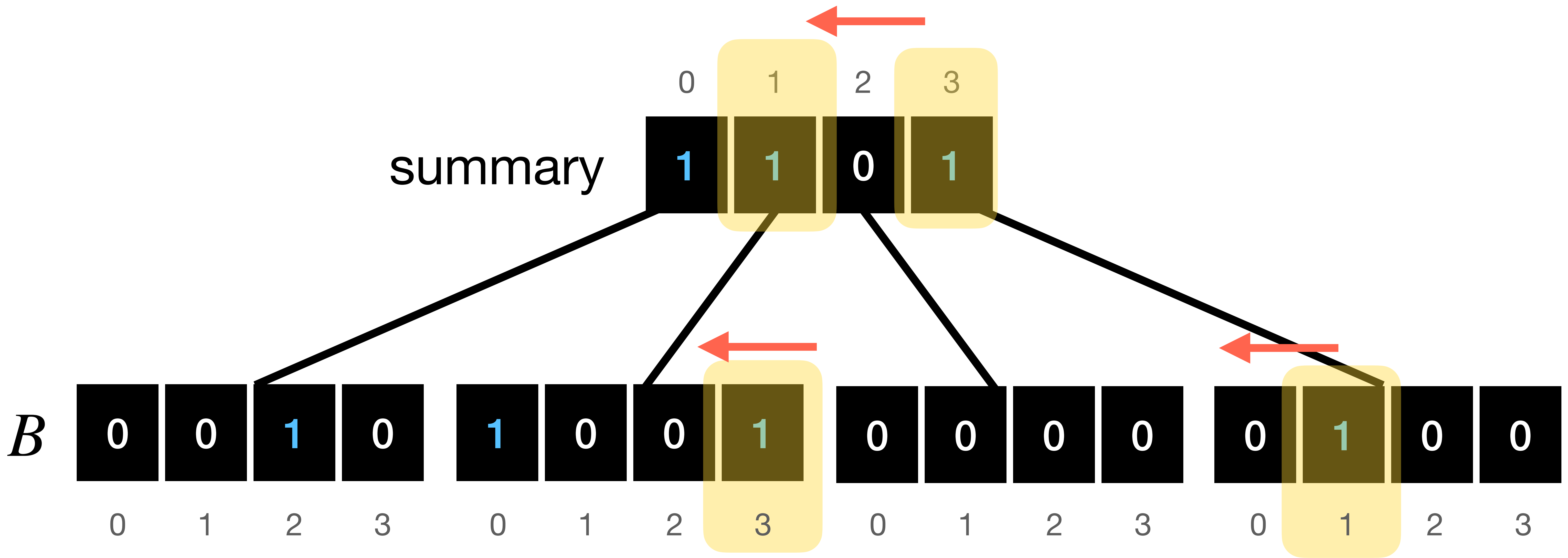
- Retrieve Min from chunk $\lfloor 13/4 \rfloor$.
- Is Min larger than $13 \bmod 4$?
- No, so answer Predecessor$(\lfloor 13/4 \rfloor) = 1$ on summary.
- Return Max from chunk 1, which is 3.



Chunking

Predecessor(13):

- Retrieve Min from chunk $\lfloor 13/4 \rfloor$.
- Is Min larger than $13 \bmod 4$?
- No, so answer Predecessor$(\lfloor 13/4 \rfloor) = 1$ on summary.
- Return Max from chunk 1, which is 3.
- Reconstruct answer: $3 + 1 \times 4 = 7$.

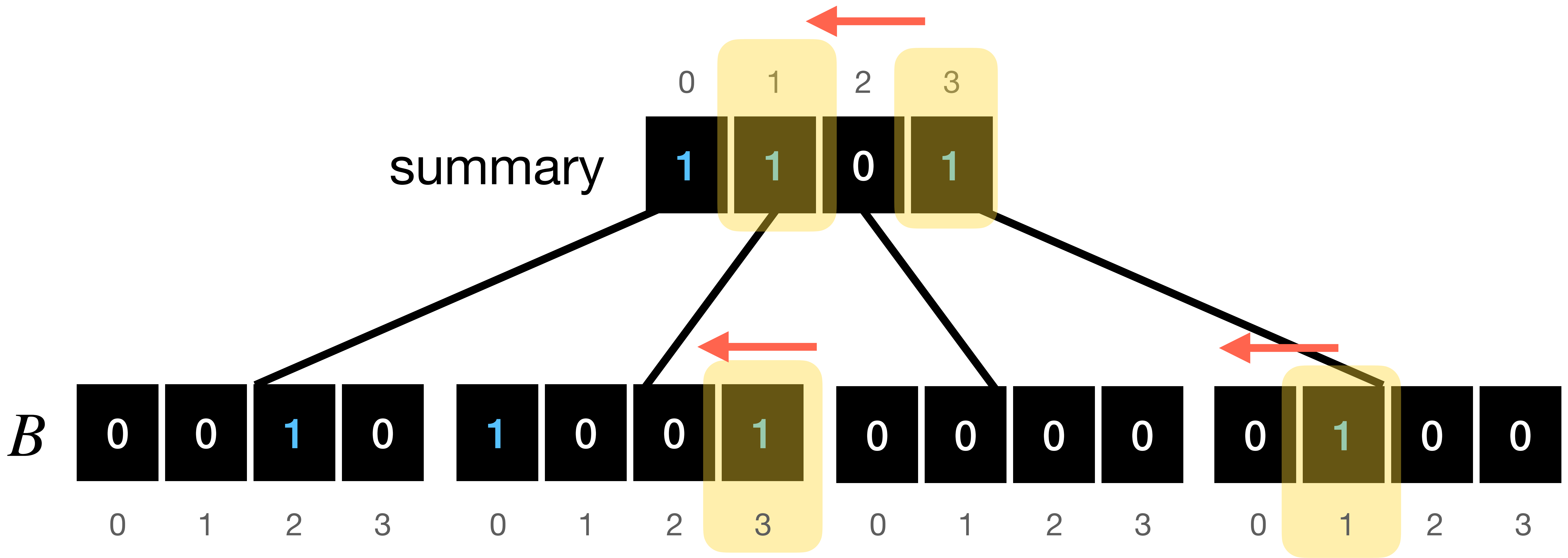


Chunking

Predecessor(13):

- Retrieve Min from chunk $\lfloor 13/4 \rfloor$.
- Is Min larger than $13 \bmod 4$?
- No, so answer Predecessor$(\lfloor 13/4 \rfloor) = 1$ on summary.
- Return Max from chunk 1, which is 3.
- Reconstruct answer: $3 + 1 \times 4 = 7$.

Insert(9):



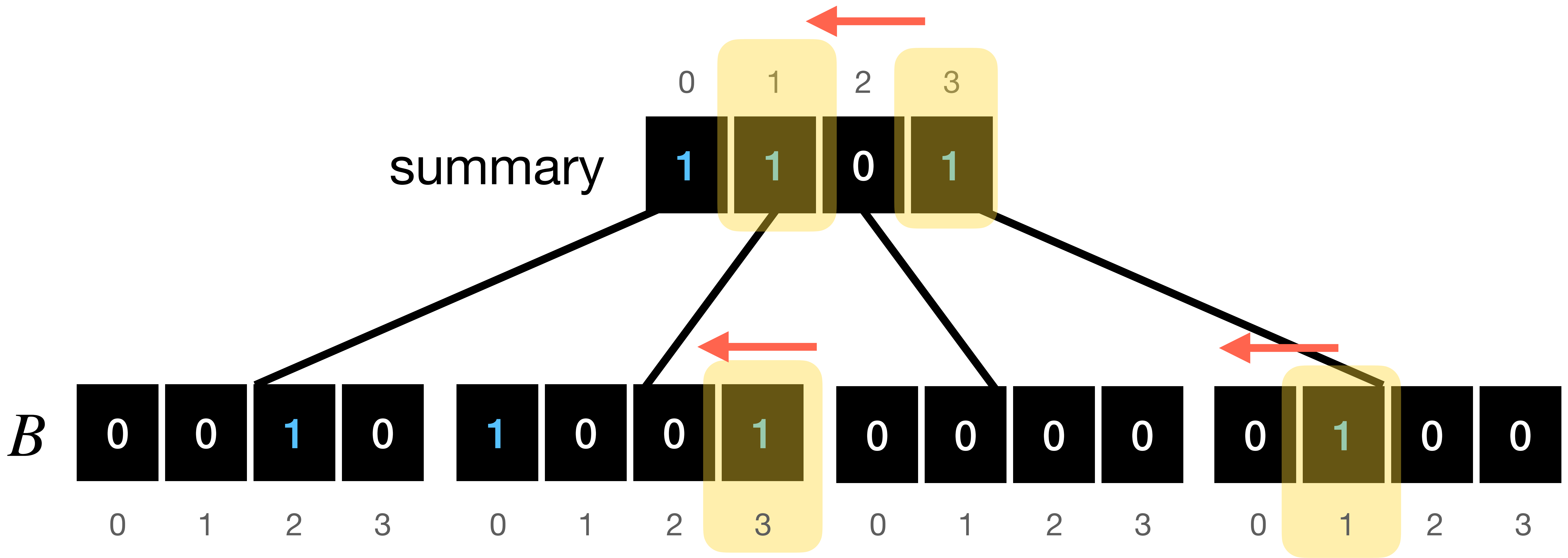
Chunking

Predecessor(13):

- Retrieve Min from chunk $\lfloor 13/4 \rfloor$.
- Is Min larger than $13 \bmod 4$?
- No, so answer Predecessor$(\lfloor 13/4 \rfloor) = 1$ on summary.
- Return Max from chunk 1, which is 3.
- Reconstruct answer: $3 + 1 \times 4 = 7$.

Insert(9):

- Is chunk $\lfloor 9/4 \rfloor$ empty? Yes.



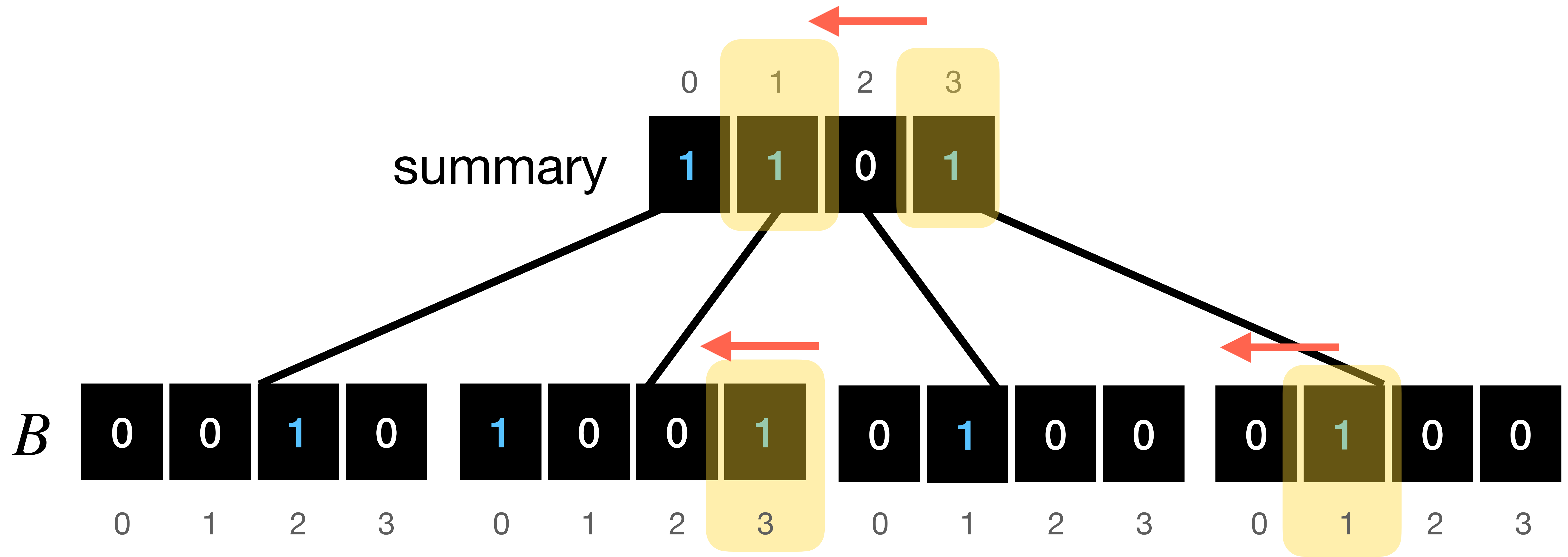
Chunking

Predecessor(13):

- Retrieve Min from chunk $\lfloor 13/4 \rfloor$.
- Is Min larger than $13 \bmod 4$?
- No, so answer Predecessor$(\lfloor 13/4 \rfloor) = 1$ on summary.
- Return Max from chunk 1, which is 3.
- Reconstruct answer: $3 + 1 \times 4 = 7$.

Insert(9):

- Is chunk $\lfloor 9/4 \rfloor$ empty? Yes.
- Insert$(9 \bmod 4)$ in chunk $\lfloor 9/4 \rfloor$.



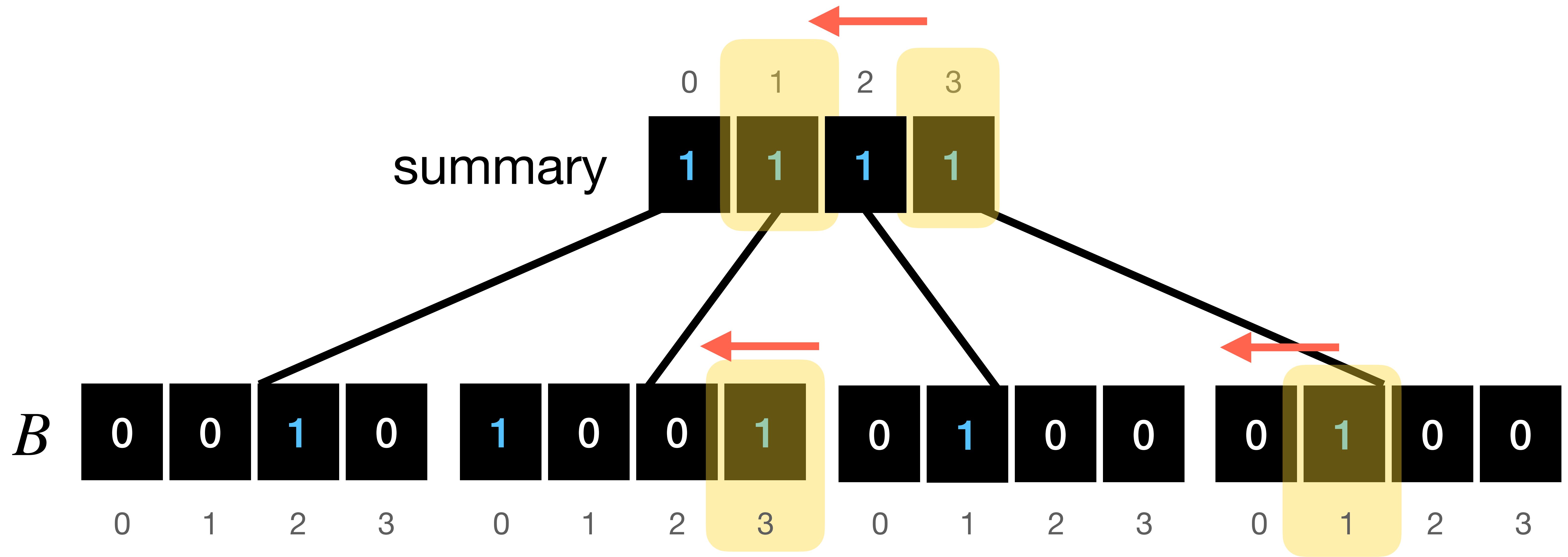
Chunking

Predecessor(13):

- Retrieve Min from chunk $\lfloor 13/4 \rfloor$.
- Is Min larger than $13 \bmod 4$?
- No, so answer Predecessor($\lfloor 13/4 \rfloor$) $= 1$ on summary.
- Return Max from chunk 1, which is 3.
- Reconstruct answer: $3 + 1 \times 4 = 7$.

Insert(9):

- Is chunk $\lfloor 9/4 \rfloor$ empty? Yes.
- Insert($9 \bmod 4$) in chunk $\lfloor 9/4 \rfloor$.
- Previous query returned "Yes", so Insert($\lfloor 9/4 \rfloor$) on summary.



Chunking

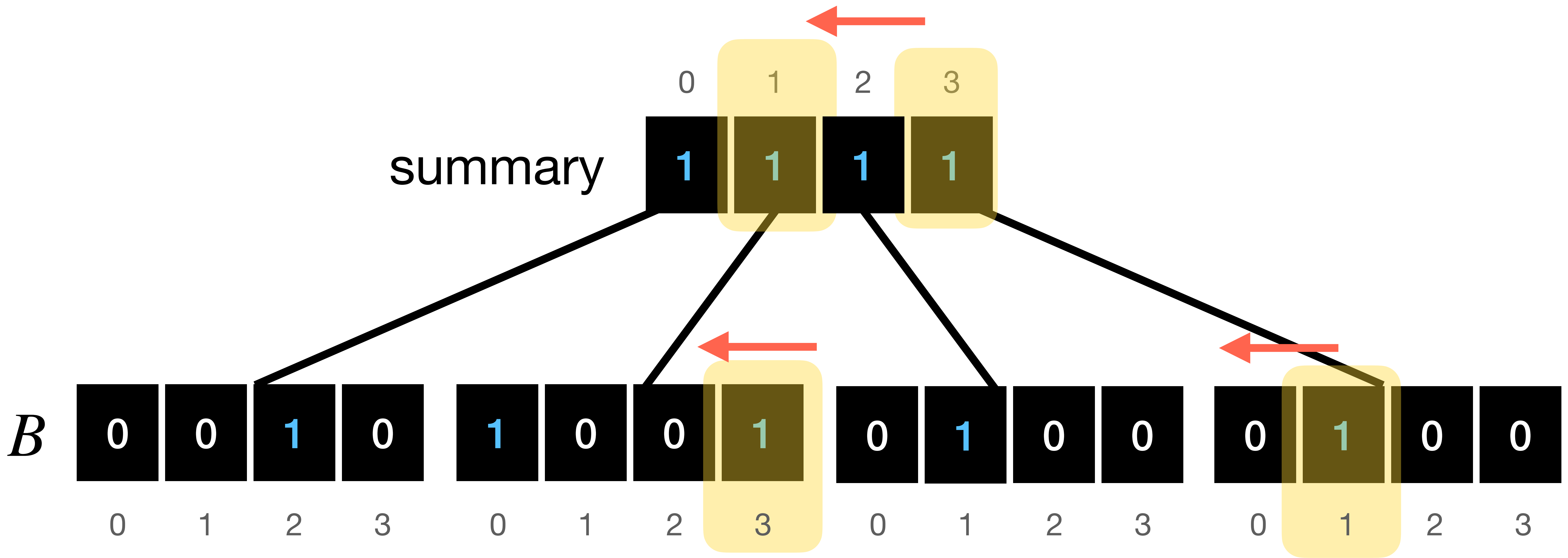
Predecessor(13):

- Retrieve Min from chunk $\lfloor 13/4 \rfloor$.
- Is Min larger than $13 \bmod 4$?
- No, so answer Predecessor$(\lfloor 13/4 \rfloor) = 1$ on summary.
- Return Max from chunk 1, which is 3.
- Reconstruct answer: $3 + 1 \times 4 = 7$.

Insert(9):

- Is chunk $\lfloor 9/4 \rfloor$ empty? Yes.
- Insert$(9 \bmod 4)$ in chunk $\lfloor 9/4 \rfloor$.
- Previous query returned "Yes", so Insert$(\lfloor 9/4 \rfloor)$ on summary.

Delete(9):



Chunking

Predecessor(13):

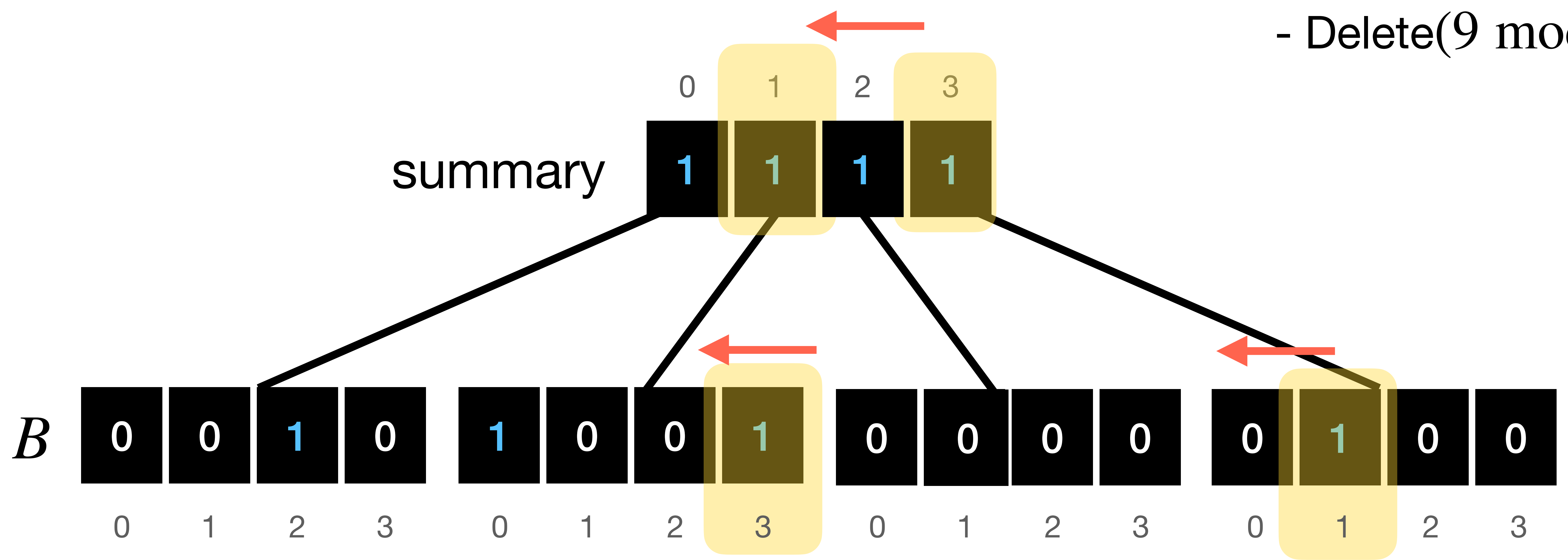
- Retrieve Min from chunk $\lfloor 13/4 \rfloor$.
- Is Min larger than $13 \bmod 4$?
- No, so answer Predecessor$(\lfloor 13/4 \rfloor) = 1$ on summary.
- Return Max from chunk 1, which is 3.
- Reconstruct answer: $3 + 1 \times 4 = 7$.

Insert(9):

- Is chunk $\lfloor 9/4 \rfloor$ empty? Yes.
- Insert$(9 \bmod 4)$ in chunk $\lfloor 9/4 \rfloor$.
- Previous query returned "Yes", so Insert$(\lfloor 9/4 \rfloor)$ on summary.

Delete(9):

- Delete$(9 \bmod 4)$ from chunk $\lfloor 9/4 \rfloor$.



Chunking

Predecessor(13):

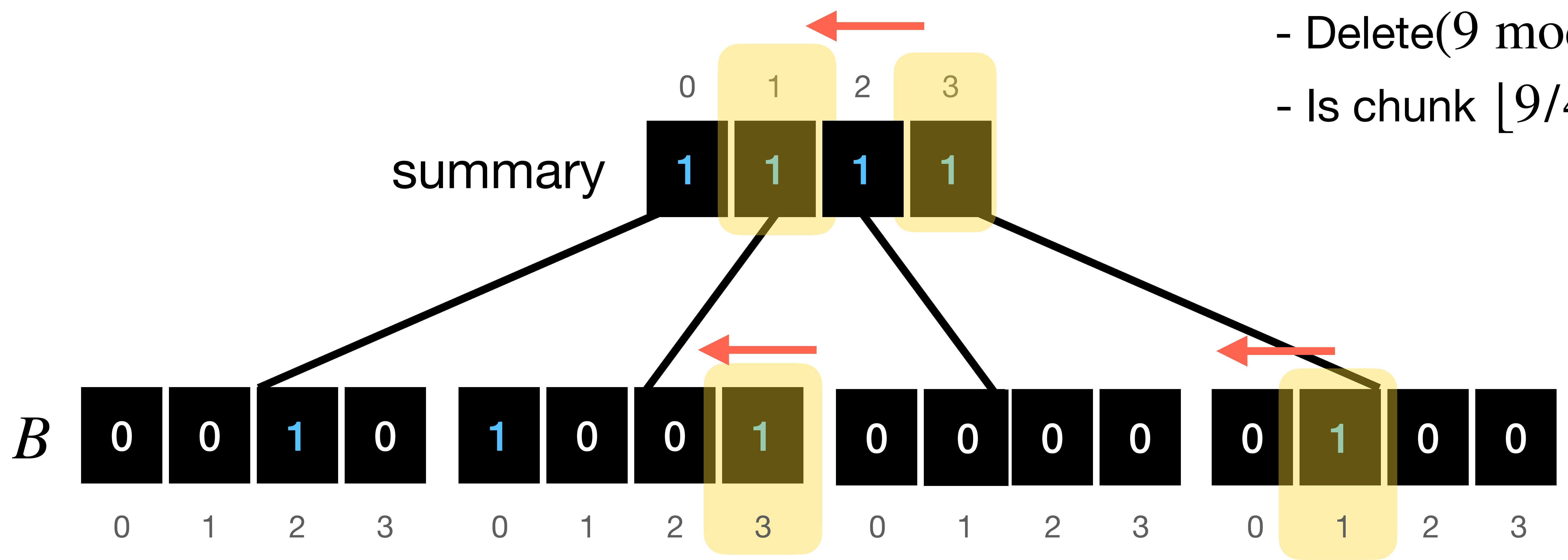
- Retrieve Min from chunk $\lfloor 13/4 \rfloor$.
- Is Min larger than $13 \bmod 4$?
- No, so answer Predecessor$(\lfloor 13/4 \rfloor) = 1$ on summary.
- Return Max from chunk 1, which is 3.
- Reconstruct answer: $3 + 1 \times 4 = 7$.

Insert(9):

- Is chunk $\lfloor 9/4 \rfloor$ empty? Yes.
- Insert$(9 \bmod 4)$ in chunk $\lfloor 9/4 \rfloor$.
- Previous query returned “Yes”, so Insert$(\lfloor 9/4 \rfloor)$ on summary.

Delete(9):

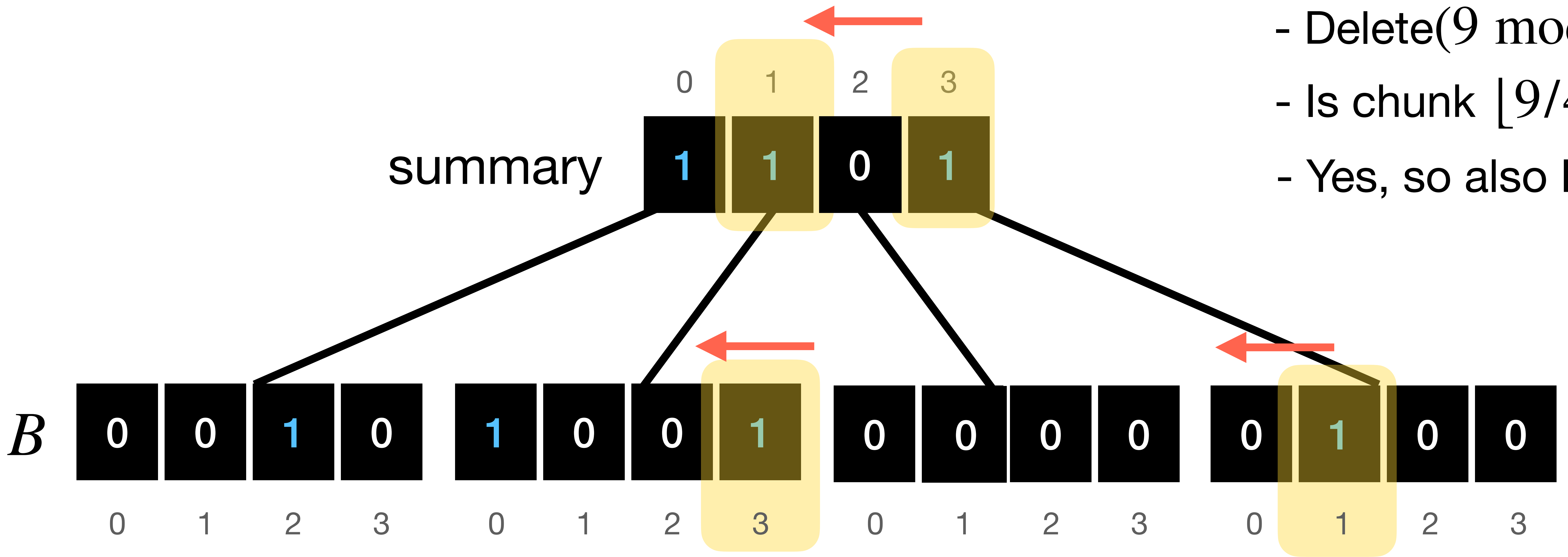
- Delete$(9 \bmod 4)$ from chunk $\lfloor 9/4 \rfloor$.
- Is chunk $\lfloor 9/4 \rfloor$ empty?



Chunking

Predecessor(13):

- Retrieve Min from chunk $\lfloor 13/4 \rfloor$.
- Is Min larger than $13 \bmod 4$?
- No, so answer Predecessor$(\lfloor 13/4 \rfloor) = 1$ on summary.
- Return Max from chunk 1, which is 3.
- Reconstruct answer: $3 + 1 \times 4 = 7$.



Insert(9):

- Is chunk $\lfloor 9/4 \rfloor$ empty? Yes.
- Insert$(9 \bmod 4)$ in chunk $\lfloor 9/4 \rfloor$.
- Previous query returned "Yes", so Insert$(\lfloor 9/4 \rfloor)$ on summary.

Delete(9):

- Delete$(9 \bmod 4)$ from chunk $\lfloor 9/4 \rfloor$.
- Is chunk $\lfloor 9/4 \rfloor$ empty?
- Yes, so also Delete$(\lfloor 9/4 \rfloor)$ from summary.

Chunking

Predecessor(13):

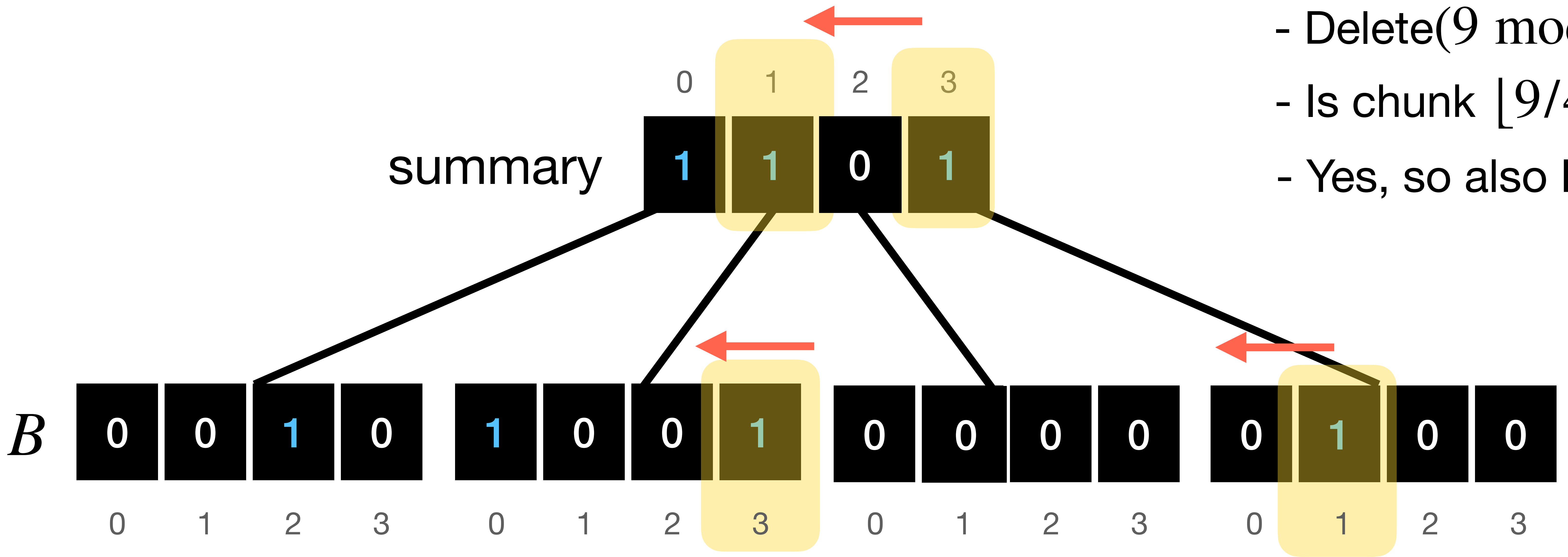
- Retrieve Min from chunk $\lfloor 13/4 \rfloor$.
- Is Min larger than $13 \bmod 4$?
- No, so answer Predecessor$(\lfloor 13/4 \rfloor) = 1$ on summary.
- Return Max from chunk 1, which is 3.
- Reconstruct answer: $3 + 1 \times 4 = 7$.

Insert(9):

- Is chunk $\lfloor 9/4 \rfloor$ empty? Yes.
- Insert$(9 \bmod 4)$ in chunk $\lfloor 9/4 \rfloor$.
- Previous query returned "Yes", so Insert$(\lfloor 9/4 \rfloor)$ on summary.

Delete(9):

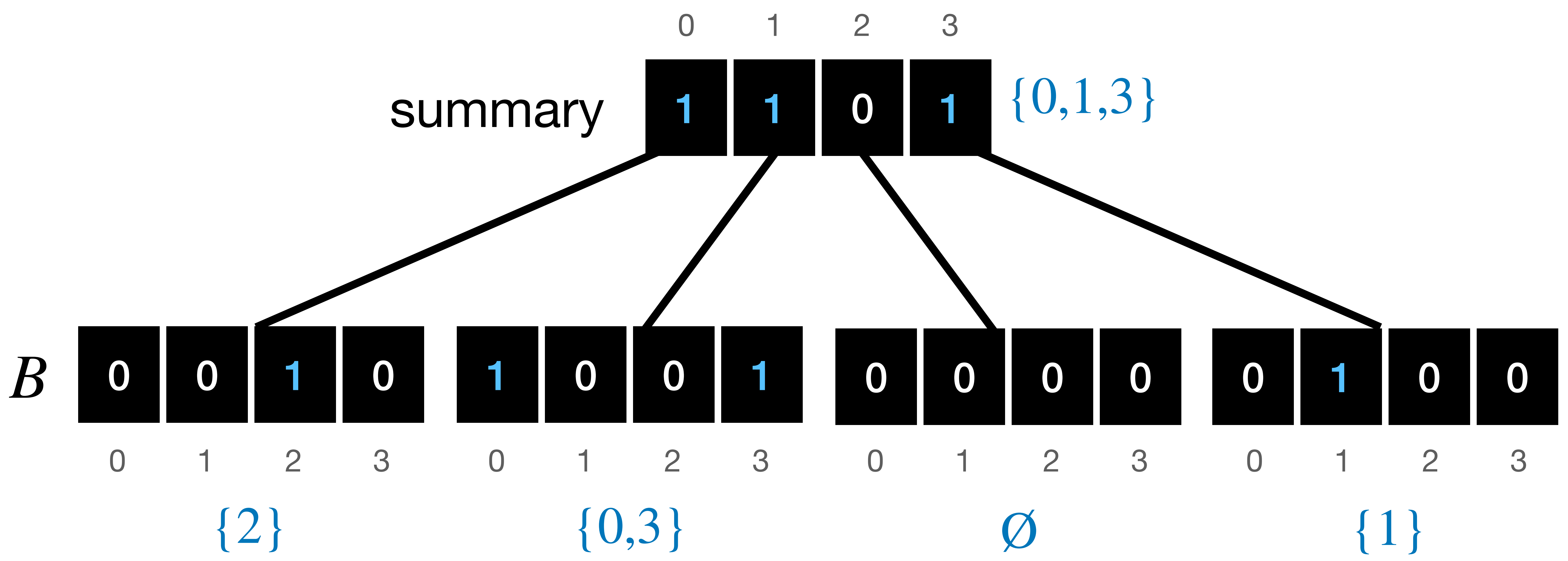
- Delete$(9 \bmod 4)$ from chunk $\lfloor 9/4 \rfloor$.
- Is chunk $\lfloor 9/4 \rfloor$ empty?
- Yes, so also Delete$(\lfloor 9/4 \rfloor)$ from summary.



In general, we recurse **twice**: on the summary **and** on a chunk.

Towards van Emde Boas trees — Recursion

- Now that we have a solution that runs in $O(\sqrt{U})$ time for a universe size U, we can use it for both the summary and the chunks, recursively, over a universe size $\sqrt{U}$.

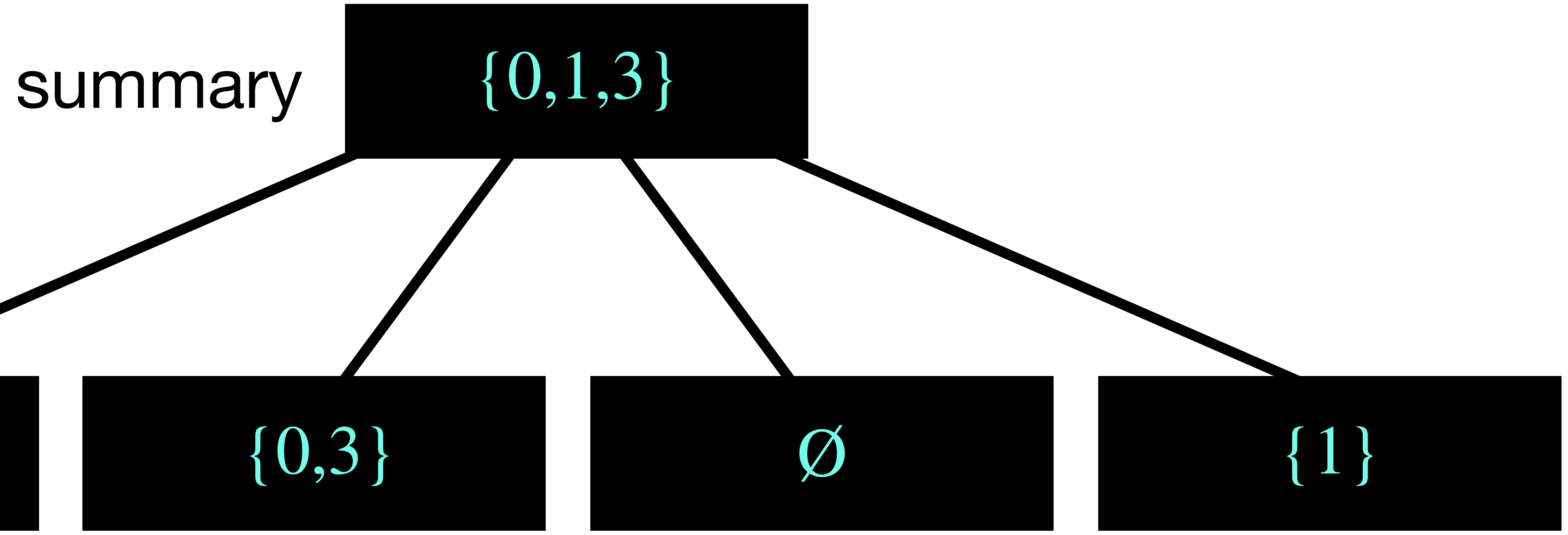


- We let high(x) $:= \lfloor x/\sqrt{U} \rfloor$ and low(x) $:= x \bmod \sqrt{U}$.

x	high(x)	low(x)
2	0	2
4	1	0
7	1	3
13	3	1

Towards van Emde Boas trees

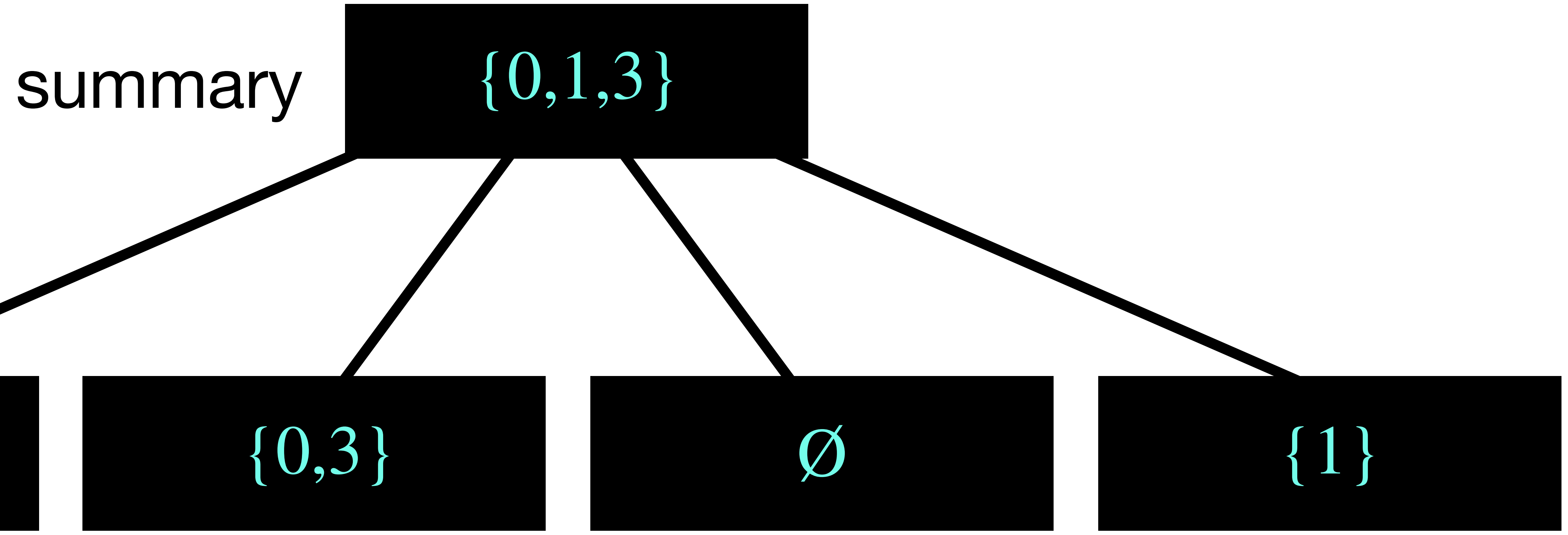
- Let's see how efficient this data structure is.



Towards van Emde Boas trees

- Let's see how efficient this data structure is.

Min():

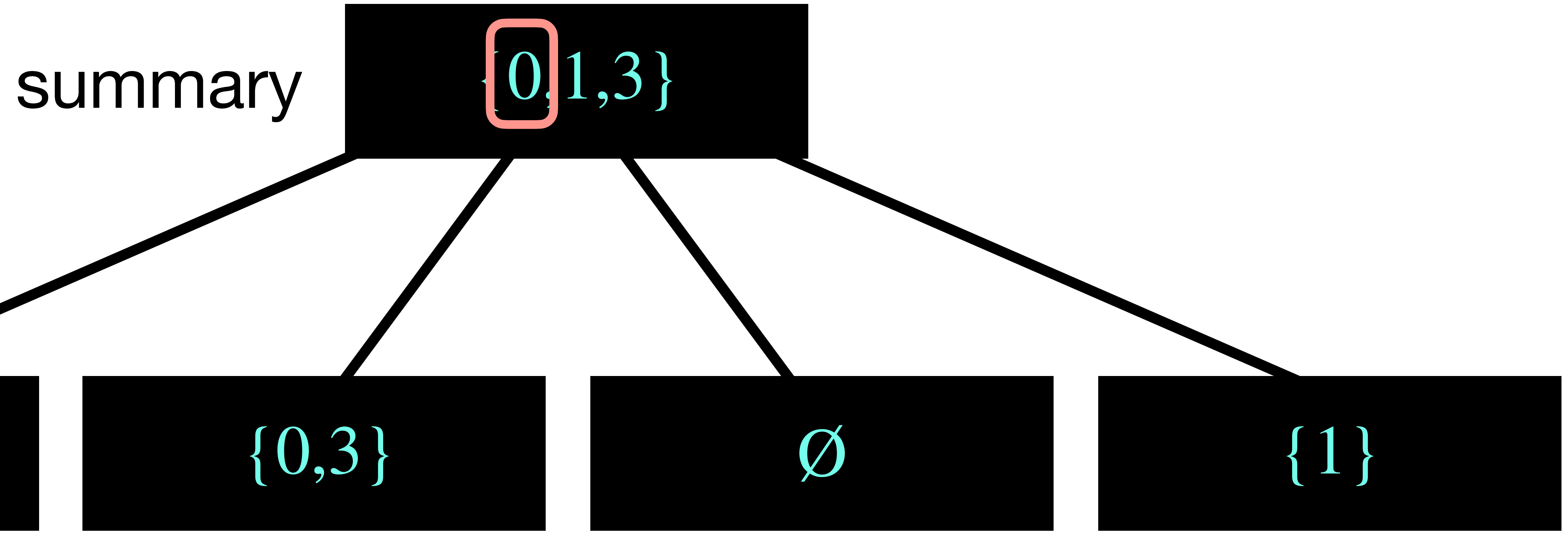


Towards van Emde Boas trees

- Let's see how efficient this data structure is.

Min():

- Answer $i =$ Min() on summary.

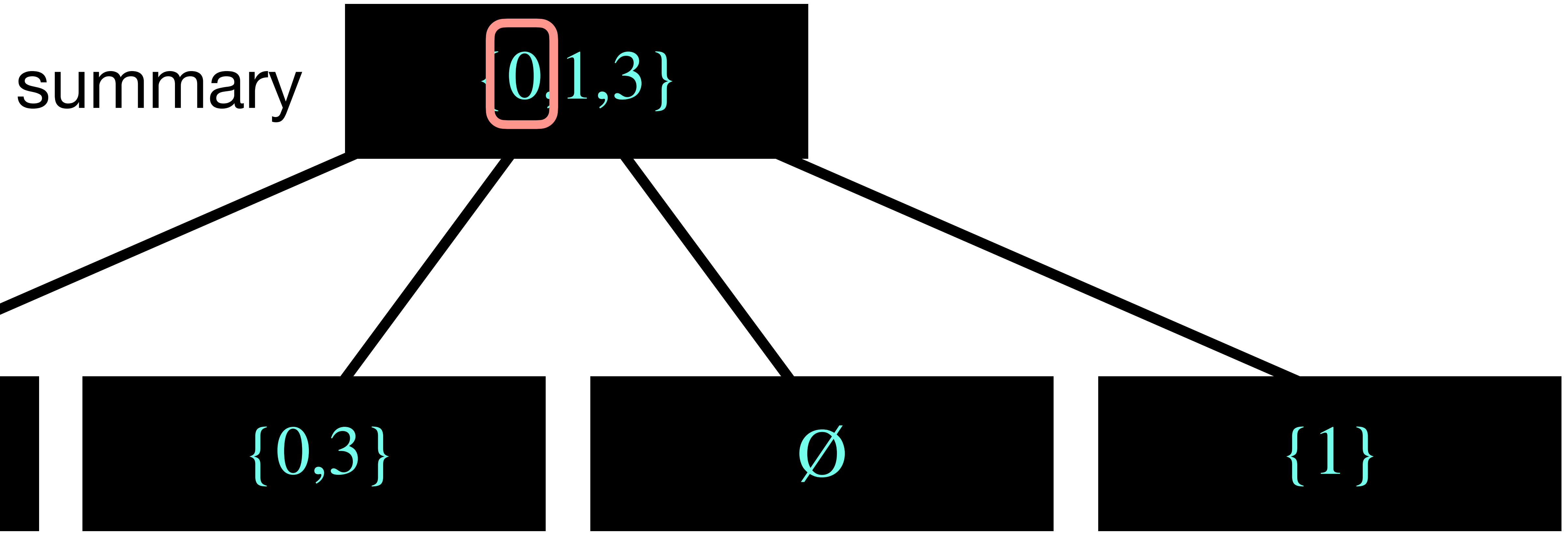


Towards van Emde Boas trees

- Let’s see how efficient this data structure is.

Min():

- Answer $i =$ Min() on summary.
- If $i = \perp$, return $\perp$.
- Otherwise, return Min() on child i.



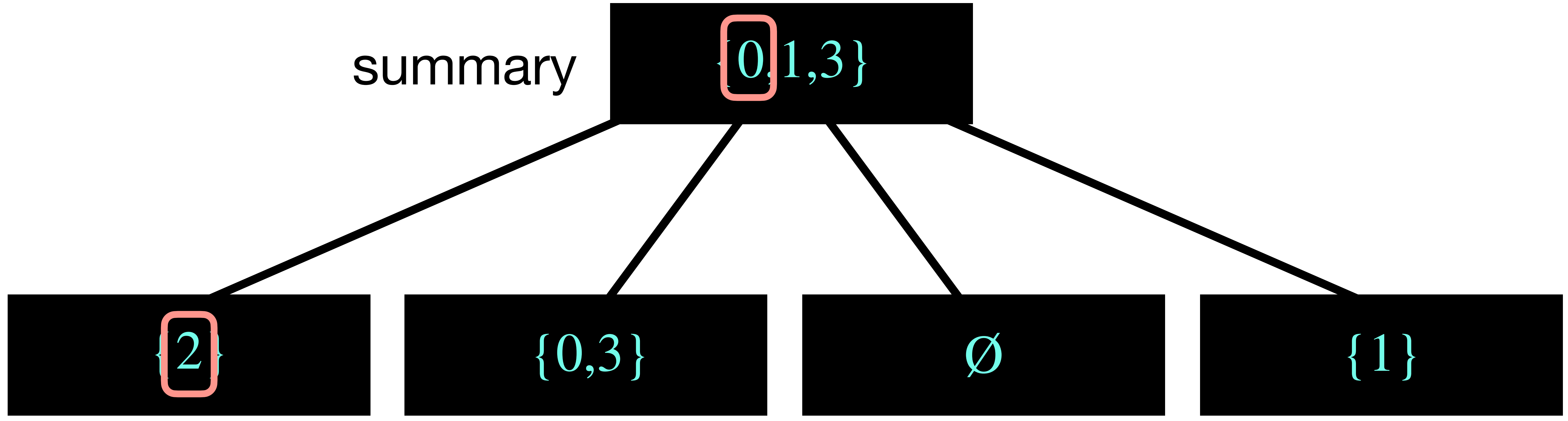
Towards van Emde Boas trees

- Let's see how efficient this data structure is.

Min():

- Answer $i =$ Min() on summary.
- If $i = \perp$, return $\perp$.
- Otherwise, return Min() on child i.

Runtime is
$T(U) \leq \Theta(1) + 2T(\sqrt{U}), T(2) = \Theta(1)$.



Towards van Emde Boas trees

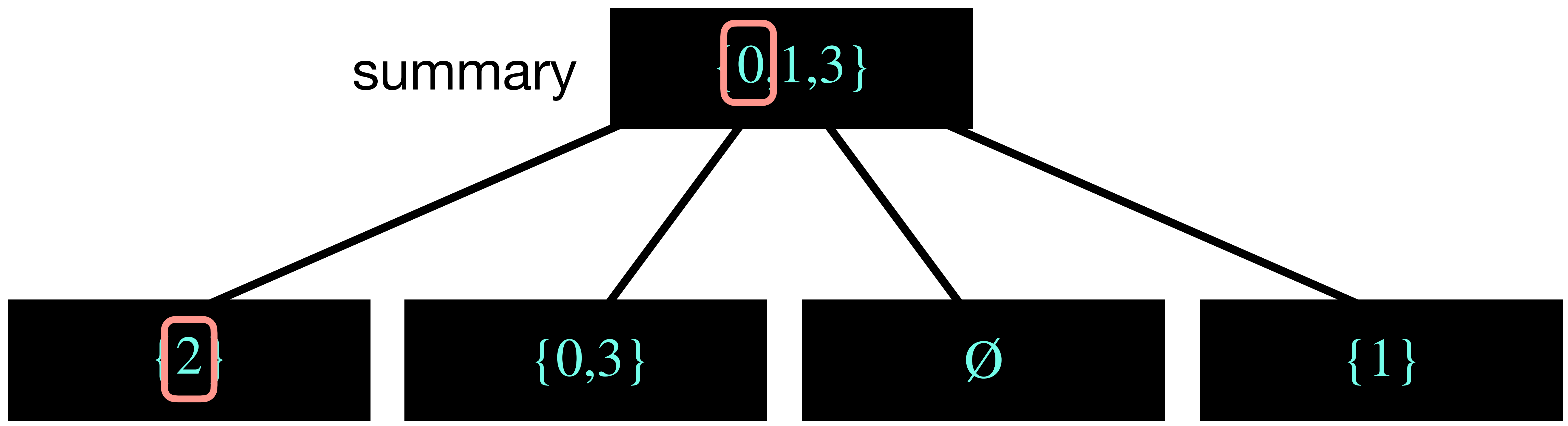
- Let's see how efficient this data structure is.

Min():

- Answer $i =$ Min() on summary.
- If $i = \perp$, return $\perp$.
- Otherwise, return Min() on child i.

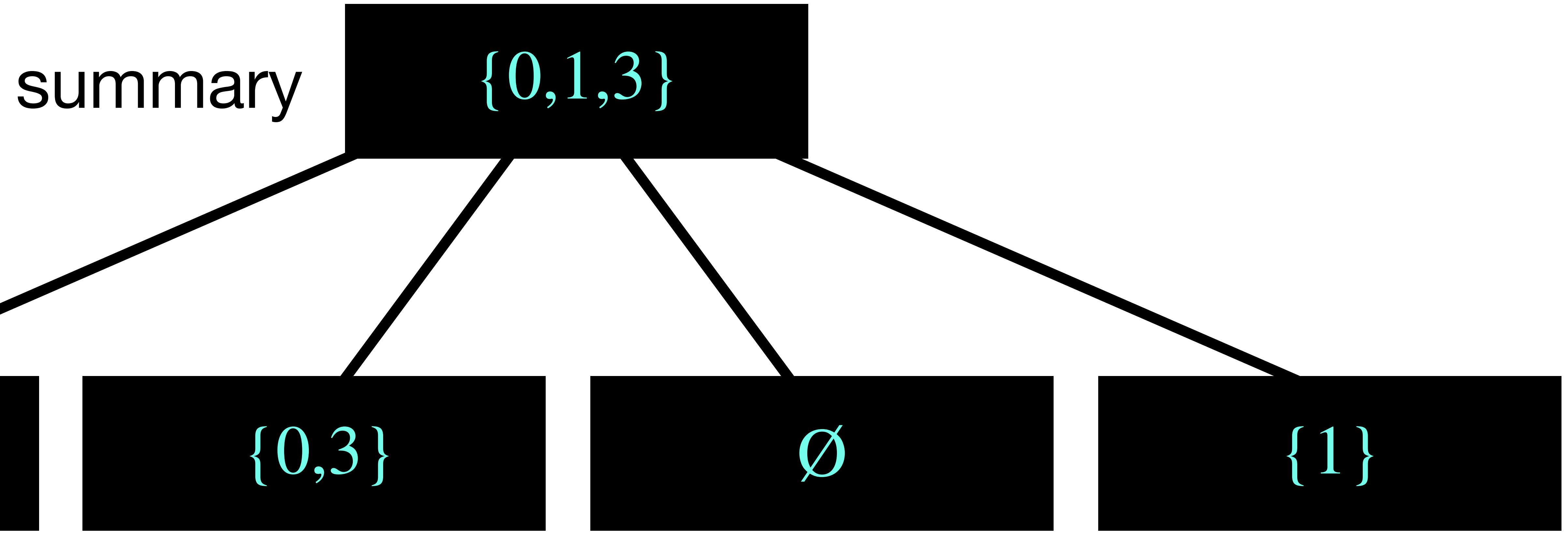
Runtime is
$T(U) \leq \Theta(1) + 2T(\sqrt{U}), T(2) = \Theta(1)$.

Letting $w = \log_2 U$ and $R(w) = T(2^w)$, we have
$R(w) \leq \Theta(1) + 2R(w/2) \to R(w) = O(w)$.
Hence $T(U) = O(\log U)$.



Towards van Emde Boas trees

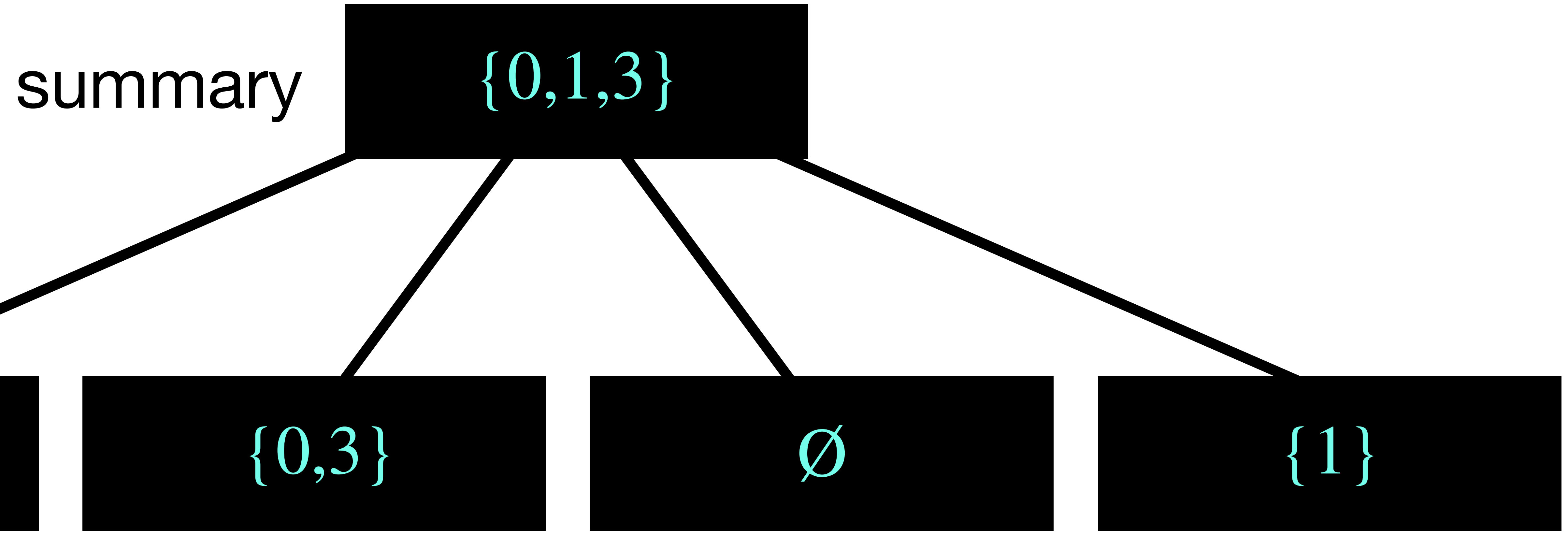
- Let’s see how efficient this data structure is.



Towards van Emde Boas trees

- Let's see how efficient this data structure is.

Insert(9):

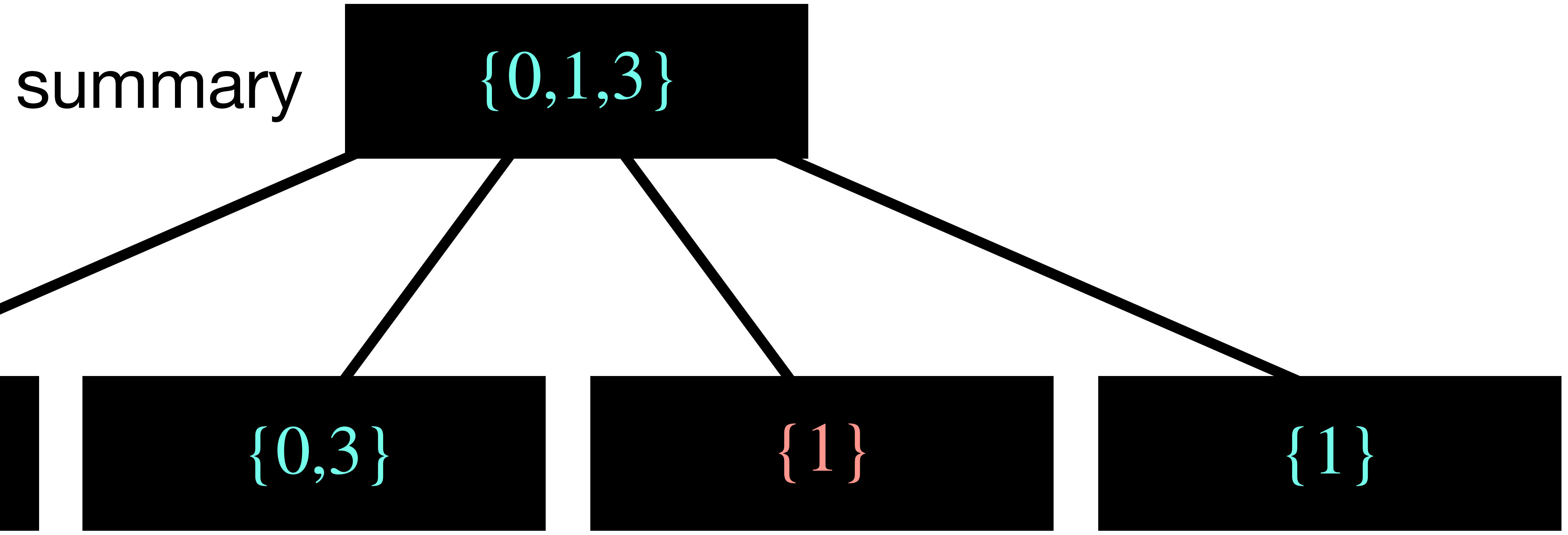


Towards van Emde Boas trees

- Let's see how efficient this data structure is.

Insert(9):

- Insert$(9 \bmod 4)$ in child $\lfloor 9/4 \rfloor$.

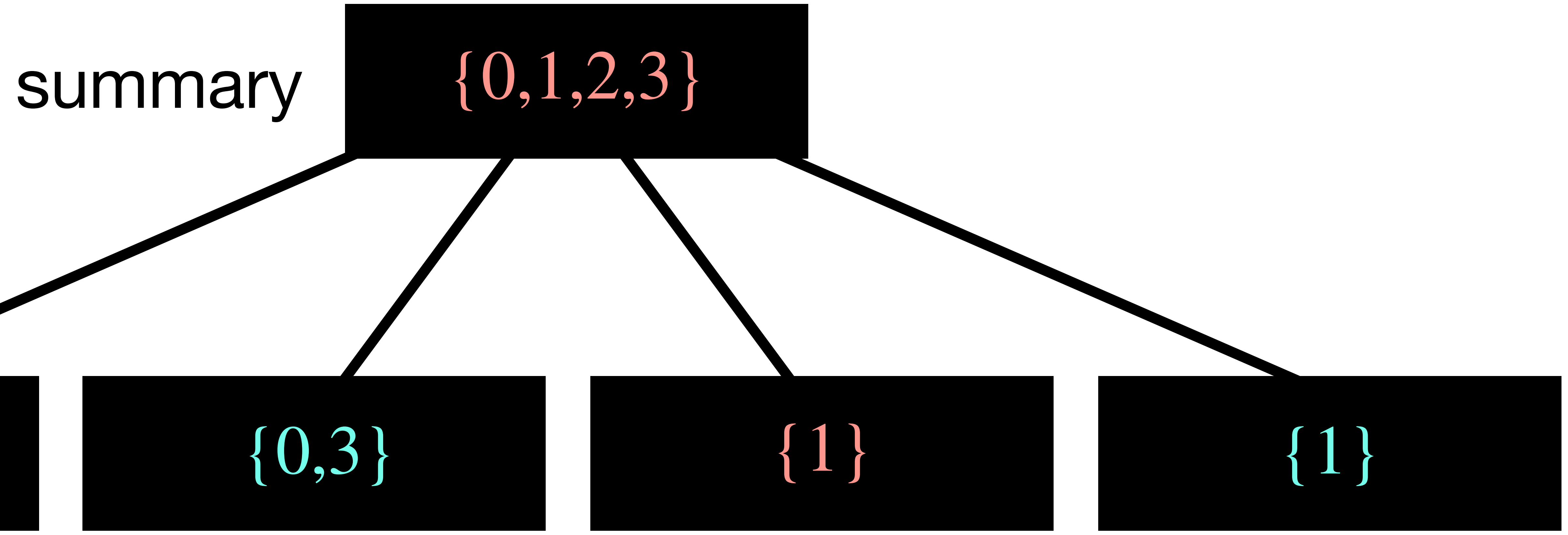


Towards van Emde Boas trees

- Let's see how efficient this data structure is.

Insert(9):

- Insert$(9 \bmod 4)$ in child $\lfloor 9/4 \rfloor$.
- Insert$(\lfloor 9/4 \rfloor)$ in summary.



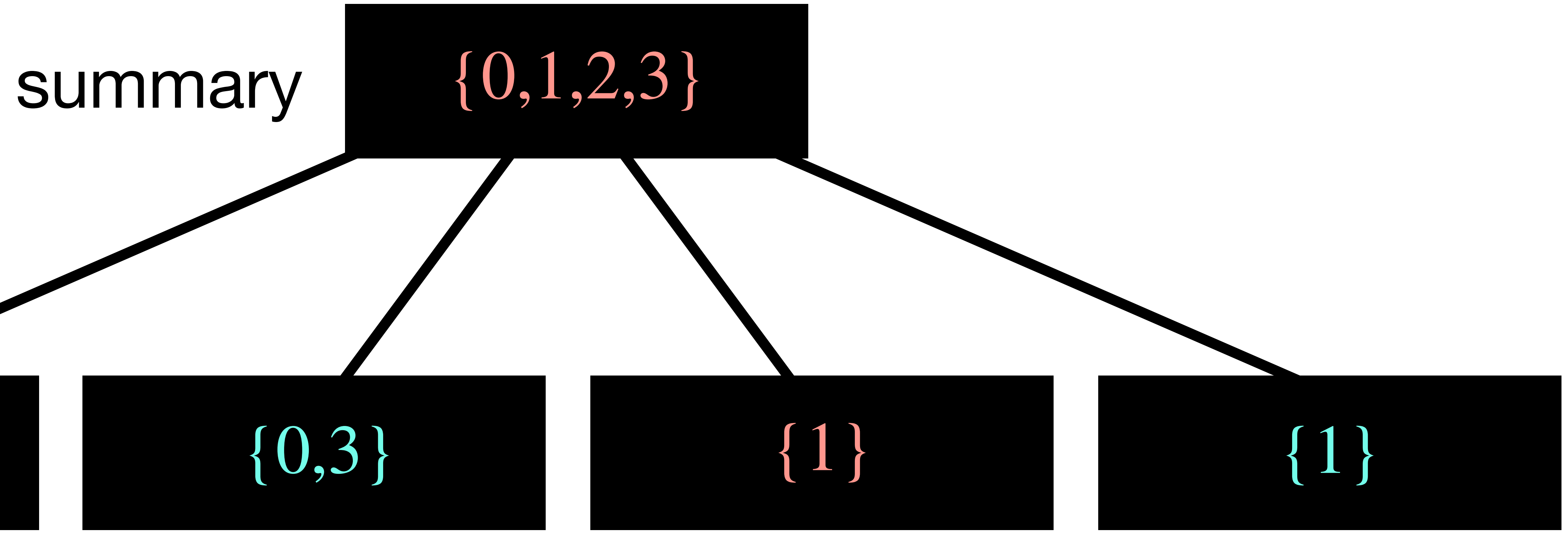
Towards van Emde Boas trees

- Let's see how efficient this data structure is.

Insert(9):

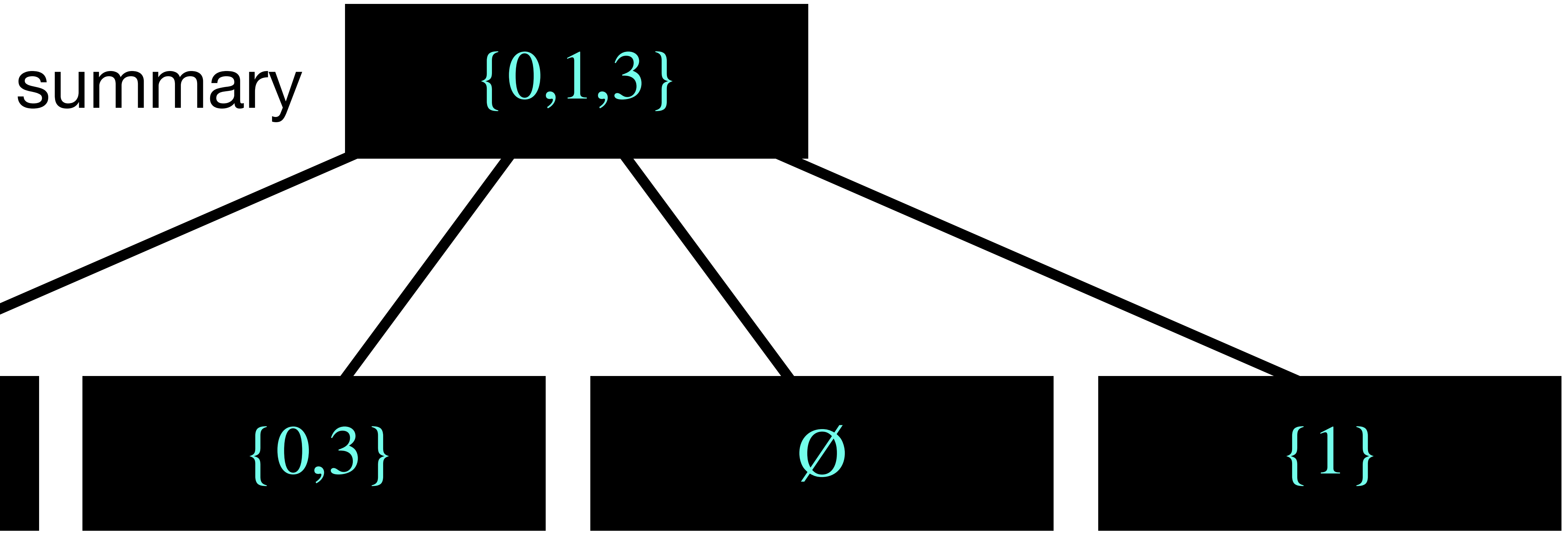
- Insert$(9 \bmod 4)$ in child $\lfloor 9/4 \rfloor$.
- Insert$(\lfloor 9/4 \rfloor)$ in summary.

Runtime is
$T(U) \leq \Theta(1) + 2T(\sqrt{U}), T(2) = \Theta(1) \ \rightarrow O(\log U)$.



Towards van Emde Boas trees

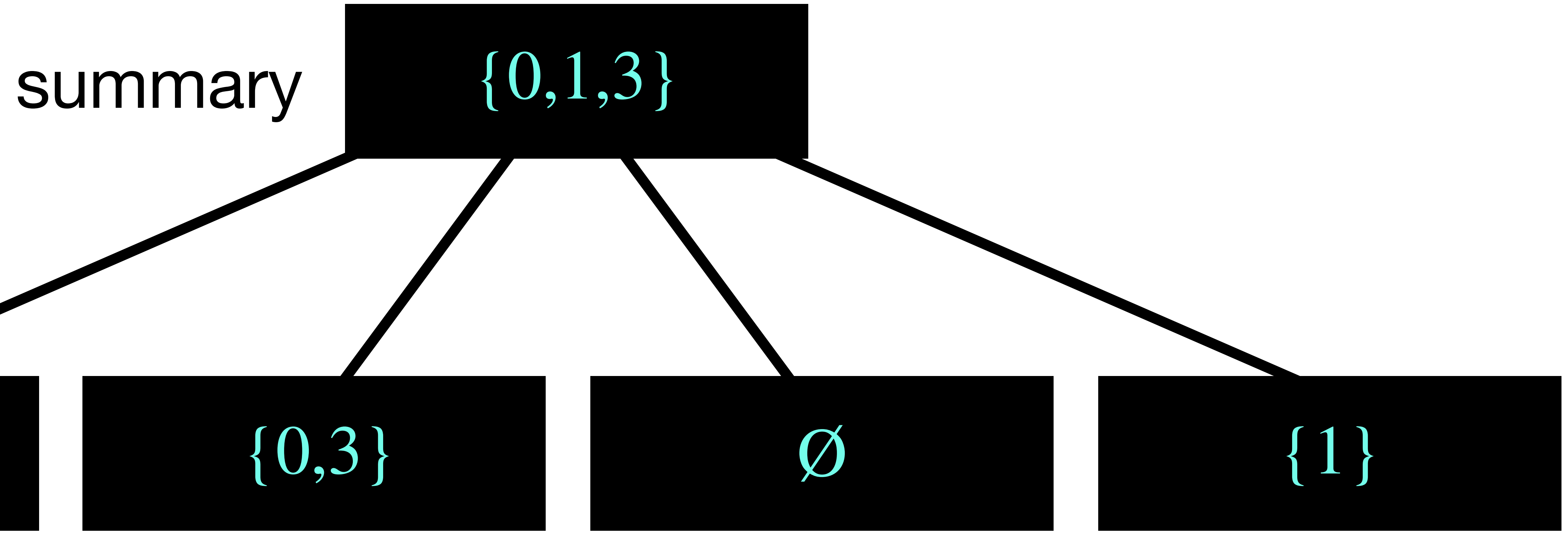
- Let’s see how efficient this data structure is.



Towards van Emde Boas trees

- Let's see how efficient this data structure is.

Predecessor(13):

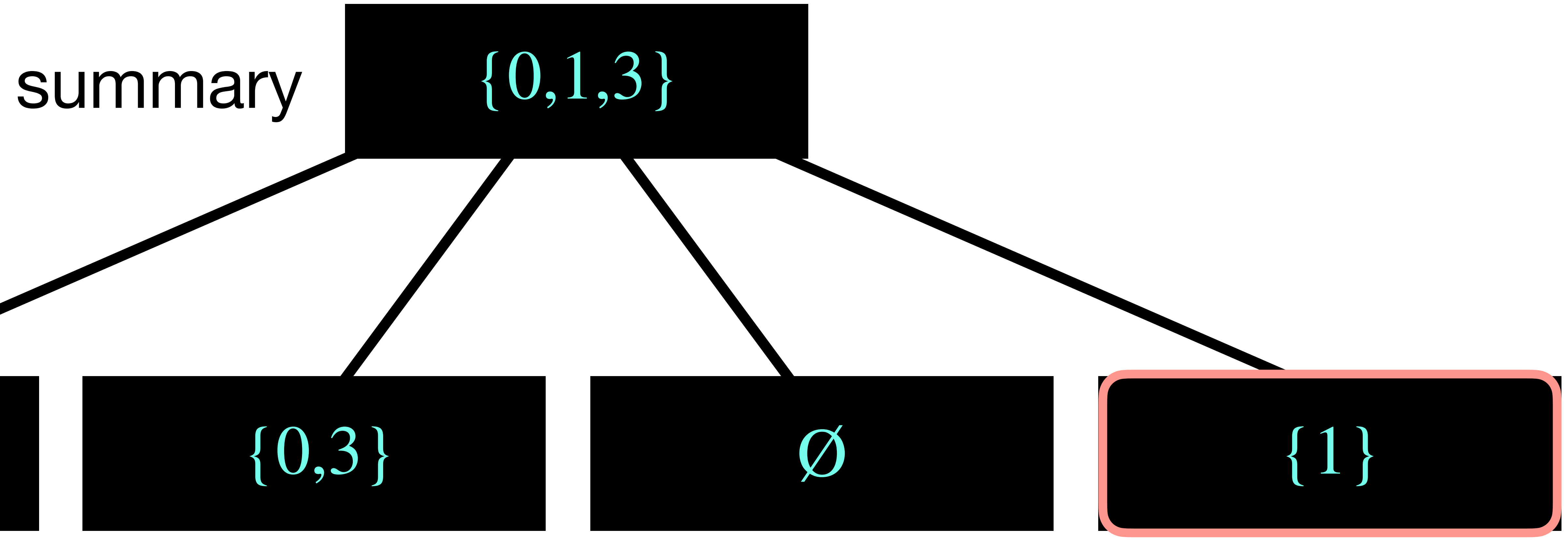


Towards van Emde Boas trees

- Let's see how efficient this data structure is.

Predecessor(13):

- Retrieve Min from child $\lfloor 13/4 \rfloor$.
- Is Min larger than $13 \bmod 4$?

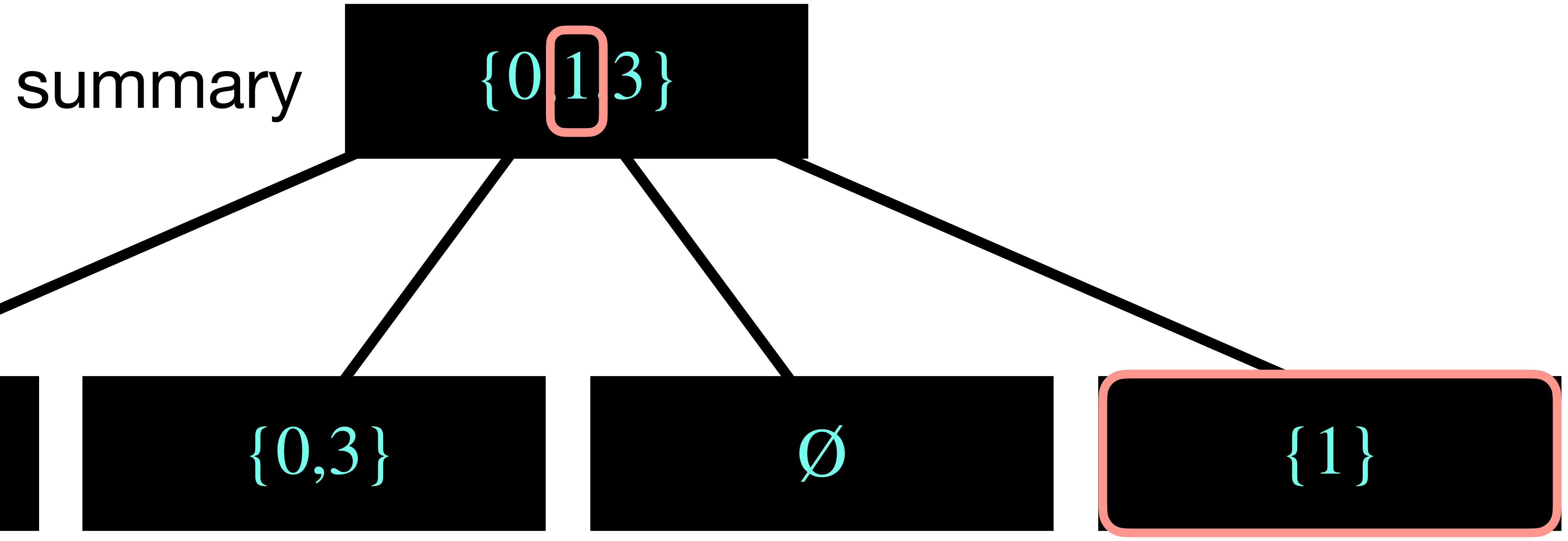


Towards van Emde Boas trees

- Let's see how efficient this data structure is.

Predecessor(13):

- Retrieve Min from child $\lfloor 13/4 \rfloor$.
- Is Min larger than $13 \bmod 4$?
- No, so answer Predecessor$(\lfloor 13/4 \rfloor) = 1$ on summary.

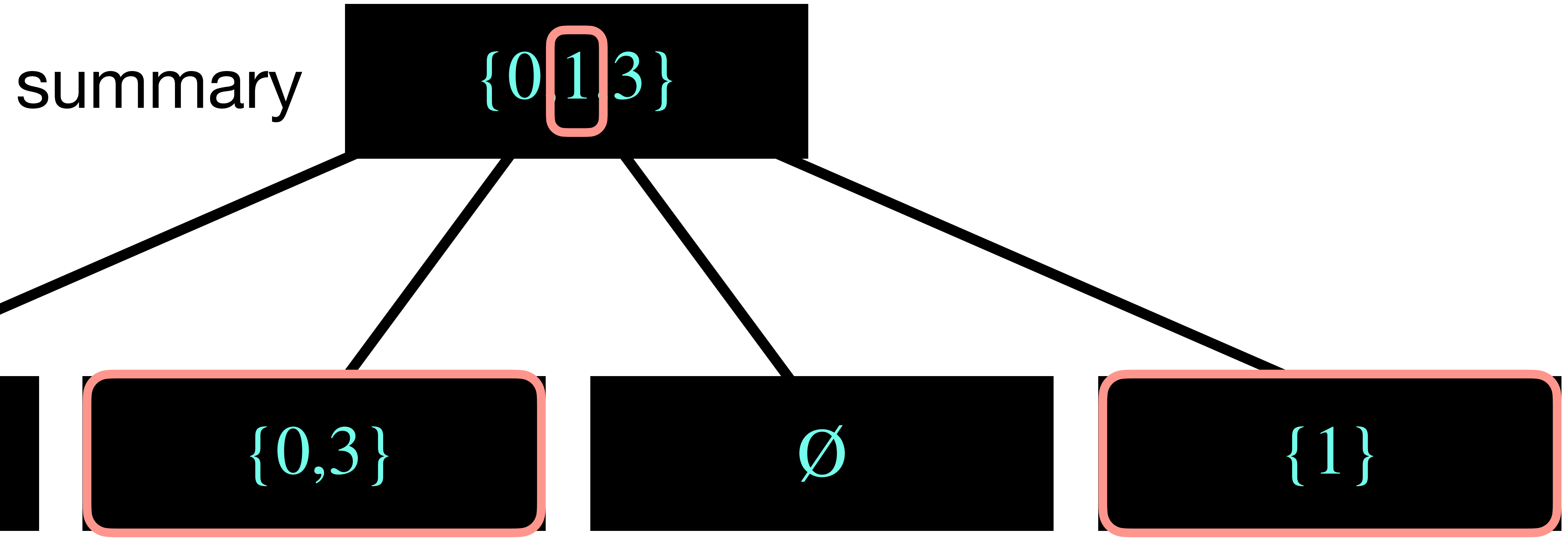


Towards van Emde Boas trees

- Let's see how efficient this data structure is.

Predecessor(13):

- Retrieve Min from child $\lfloor 13/4 \rfloor$.
- Is Min larger than $13 \bmod 4$?
- No, so answer Predecessor$(\lfloor 13/4 \rfloor) = 1$ on summary.
- Return Max from child 1, which is 3.
- Reconstruct answer: $3 + 1 \times 4 = 7$.



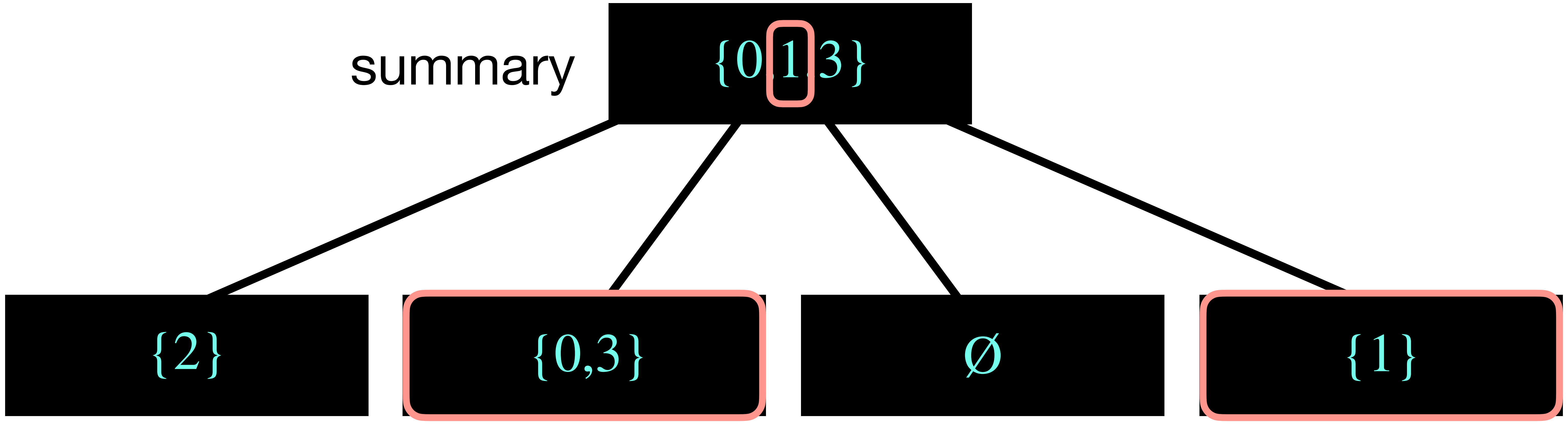
Towards van Emde Boas trees

- Let's see how efficient this data structure is.

Predecessor(13):

- Retrieve Min from child $\lfloor 13/4 \rfloor$.
- Is Min larger than $13 \bmod 4$?
- No, so answer Predecessor$(\lfloor 13/4 \rfloor) = 1$ on summary.
- Return Max from child 1, which is 3.
- Reconstruct answer: $3 + 1 \times 4 = 7$.

Runtime is
$T(U) \leq O(\log U) + T(\sqrt{U}), T(2) = \Theta(1)$.



Towards van Emde Boas trees

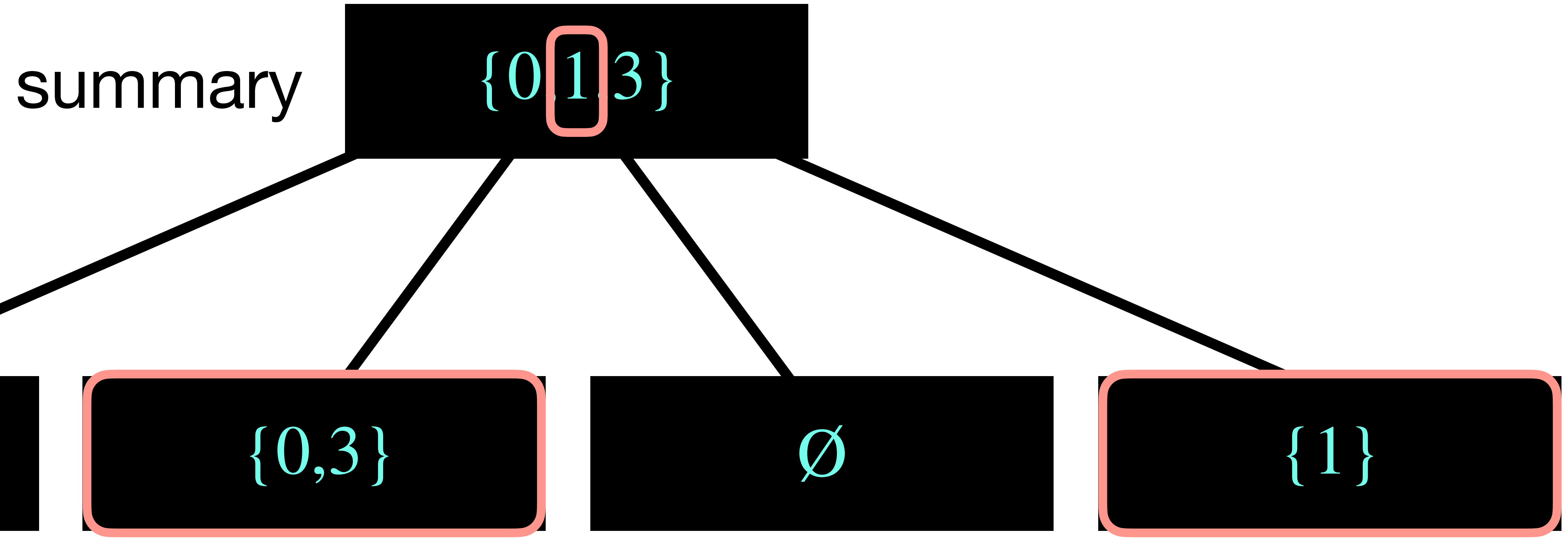
- Let’s see how efficient this data structure is.

Predecessor(13):

- Retrieve Min from child $\lfloor 13/4 \rfloor$.
- Is Min larger than $13 \bmod 4$?
- No, so answer Predecessor$(\lfloor 13/4 \rfloor) = 1$ on summary.
- Return Max from child 1, which is 3.
- Reconstruct answer: $3 + 1 \times 4 = 7$.

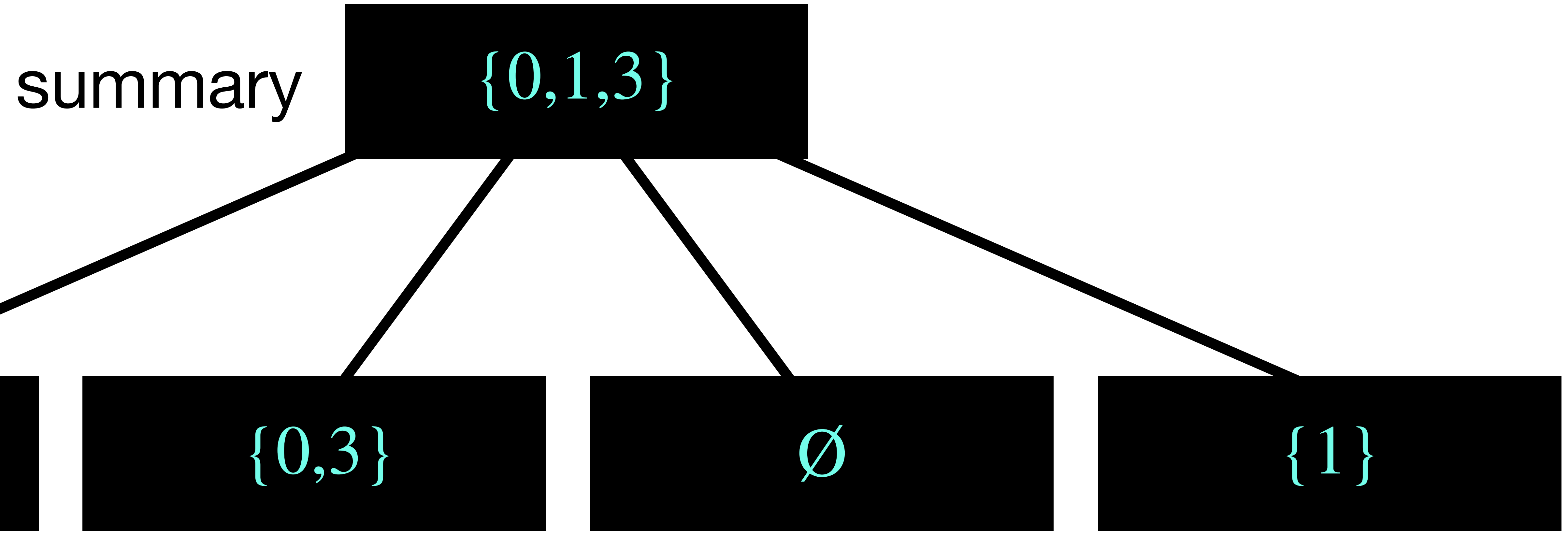
Runtime is
$T(U) \leq O(\log U) + T(\sqrt{U}), T(2) = \Theta(1)$.

Letting $w = \log_2 U$ and $R(w) = T(2^w)$, we have
$R(w) \leq O(w) + R(w/2) \to R(w) = O(w)$.
Hence $T(U) = O(\log U)$.



Towards van Emde Boas trees

- Let’s see how efficient this data structure is.

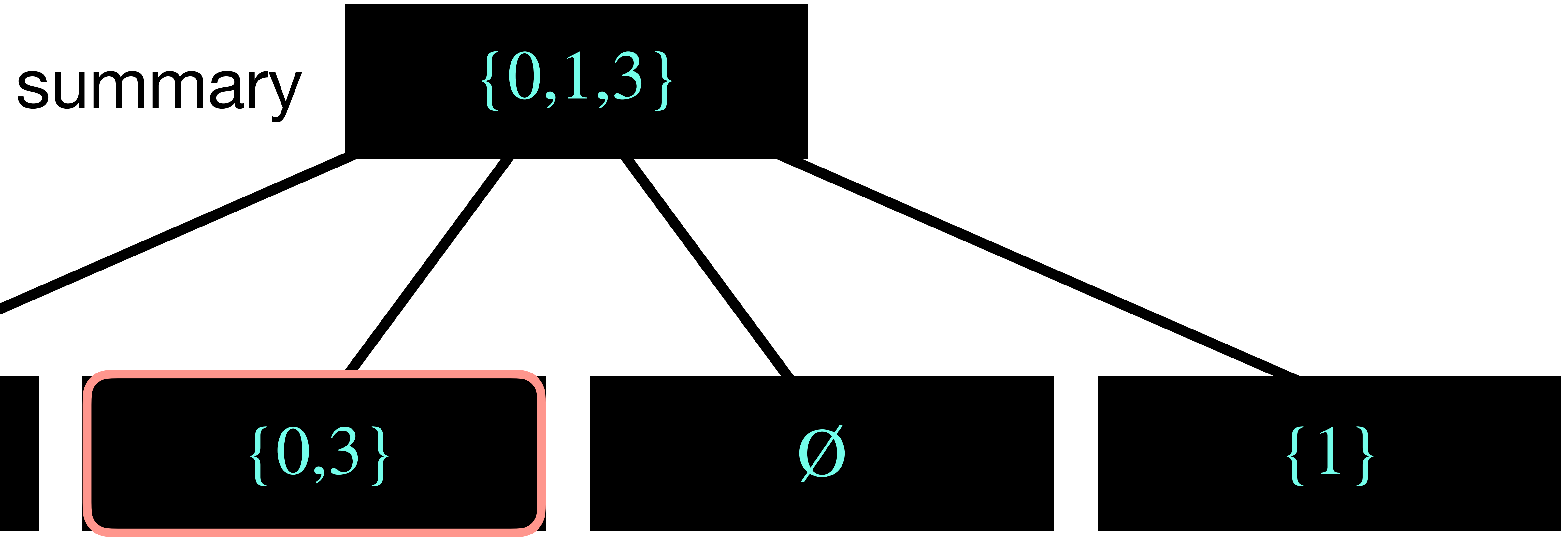


Towards van Emde Boas trees

- Let's see how efficient this data structure is.

Member(7):

- Answer Member$(7 \bmod 4)$ from chunk $\lfloor 7/4 \rfloor$.



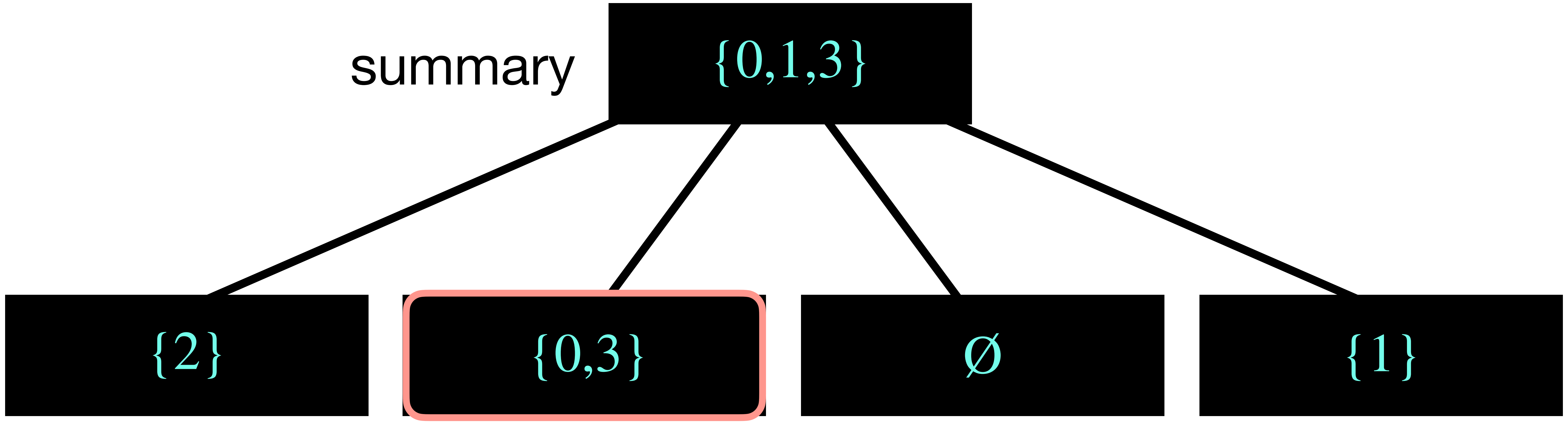
Towards van Emde Boas trees

- Let's see how efficient this data structure is.

Member(7):

- Answer Member($7 \bmod 4$) from chunk $\lfloor 7/4 \rfloor$.

Runtime is
$T(U) \leq \Theta(1) + T(\sqrt{U})$, $T(2) = \Theta(1)$.



Towards van Emde Boas trees

- Let's see how efficient this data structure is.

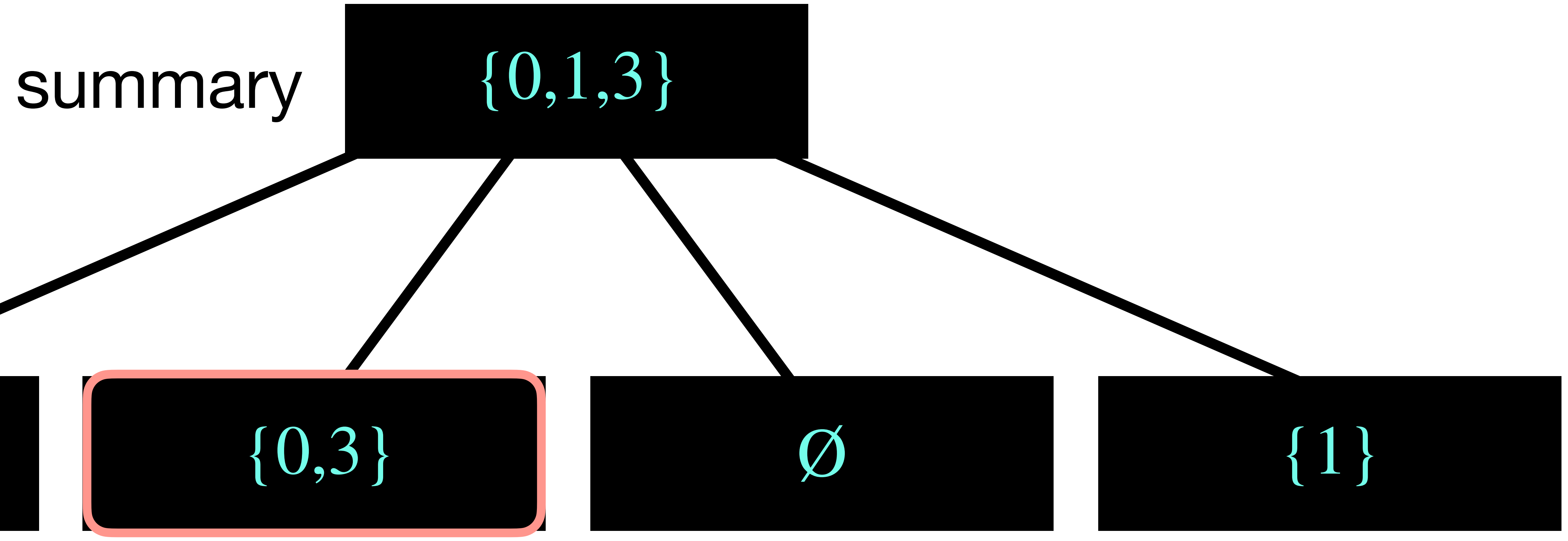
Member(7):

- Answer Member($7 \bmod 4$) from chunk $\lfloor 7/4 \rfloor$.

Runtime is
$T(U) \leq \Theta(1) + T(\sqrt{U})$, $T(2) = \Theta(1)$.

Letting $w = \log_2 U$ and $R(w) = T(2^w)$, we have
$R(w) \leq \Theta(1) + R(w/2) \to R(w) = O(\log w)$.

Hence $T(U) = O(\log \log U)$.



The story so far

- **Chunking**: $O(\sqrt{U})$ runtime.
- **Recursive chunking**: $O(\log U)$ runtime for most operations but membership in $O(\log \log U)$.

The story so far

- **Chunking**: $O(\sqrt{U})$ runtime.
- **Recursive chunking**: $O(\log U)$ runtime for most operations but membership in $O(\log\log U)$.

- … But imposing a complete binary tree on B gives a solution with $O(\log U)$ time for **all** operations and queries.

- **Our goal**: $O(\log\log U)$ time for everything.
- **Q.** Where is the problem?

The story so far

- **Chunking**: $O(\sqrt{U})$ runtime.
- **Recursive chunking**: $O(\log U)$ runtime for most operations but membership in $O(\log \log U)$.

- … But imposing a complete binary tree on B gives a solution with $O(\log U)$ time for **all** operations and queries.

- **Our goal**: $O(\log \log U)$ time for everything.
- **Q.** Where is the problem?

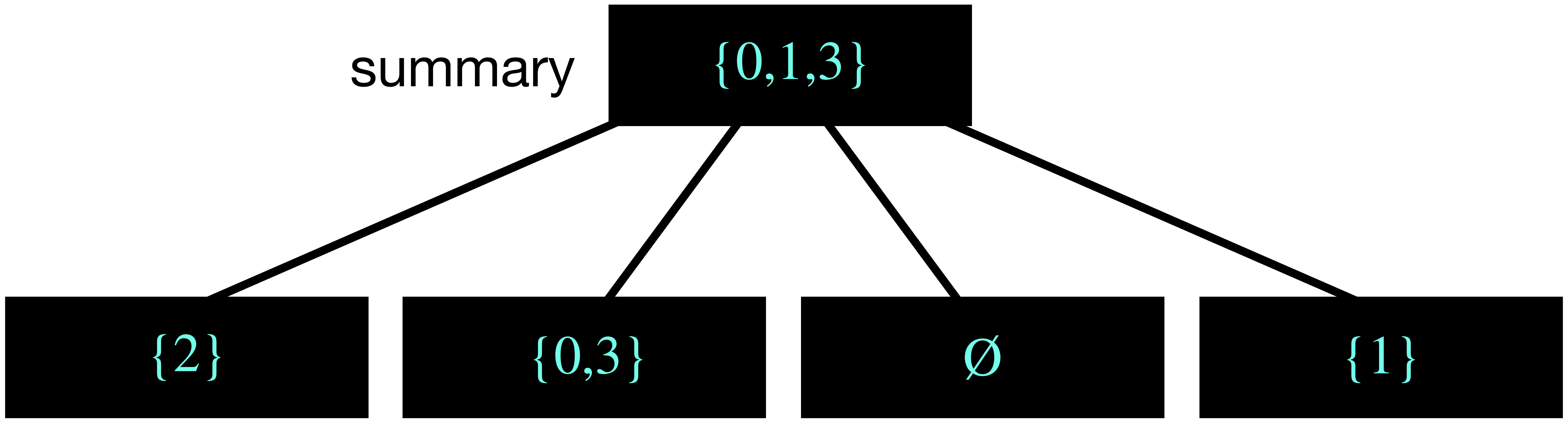
- **A.** We recurse **twice** (or we do not spend $O(1)$ per level):

$$T(U) \leq \Theta(1) + 2T(\sqrt{U}) \rightarrow O(\log U)$$

whereas $$T(U) \leq \Theta(1) + T(\sqrt{U}) \rightarrow O(\log \log U)$$

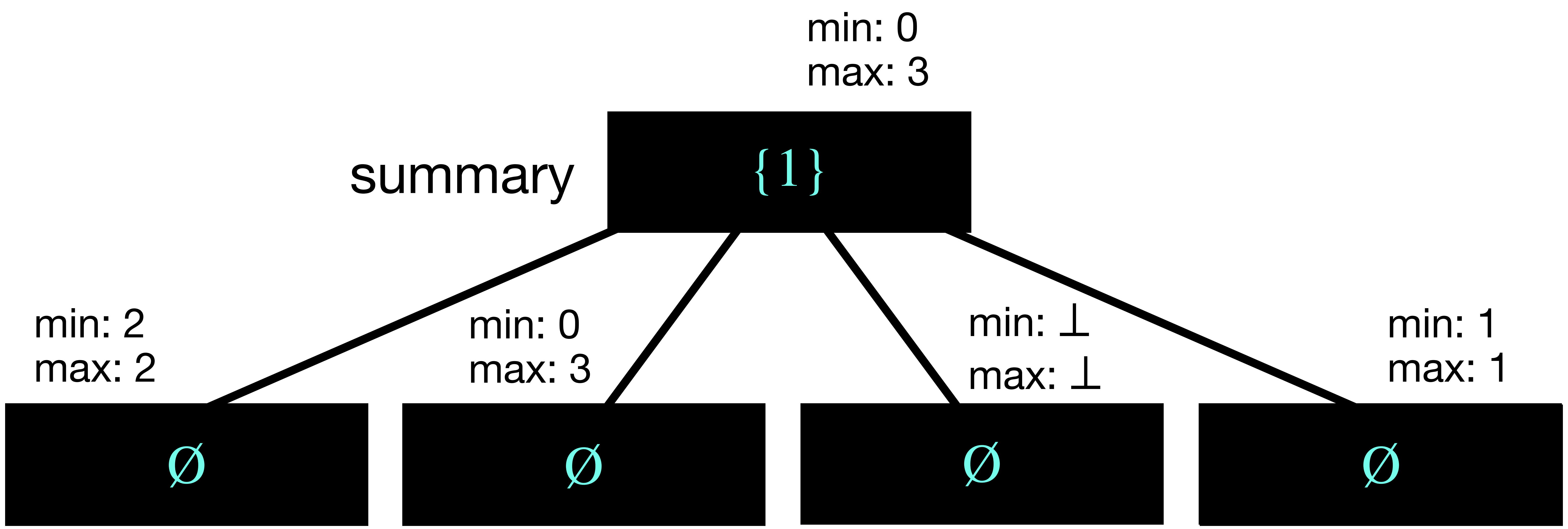
Towards van Emde Boas trees — min/max out

- Take min and max elements out from every child and summary, without representing them recursively.
- **Intuition.** Now Min() and Max(), as well as determining if a tree is empty, Insert/Delete into/from an empty tree, become constant-time ops/queries.
  With this change, we can **always** get the nice recurrence $T(U) \leq \Theta(1) + T(\sqrt{U})$, $T(2) = \Theta(1)$.



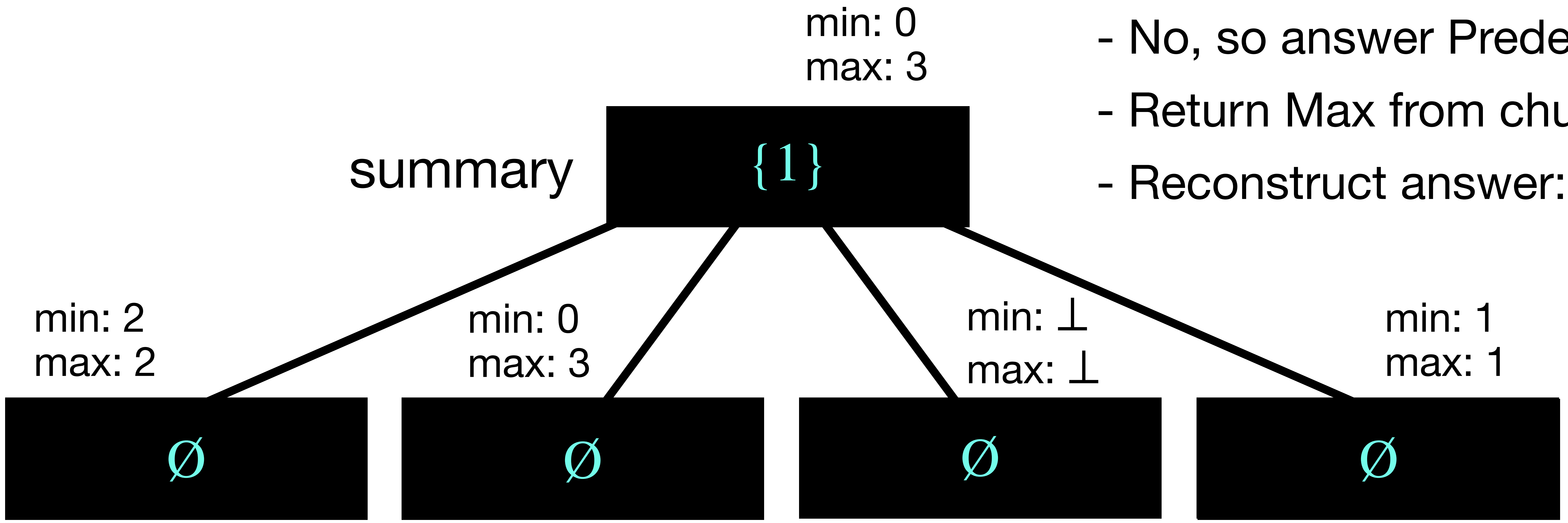
Towards van Emde Boas trees — min/max out

- Take min and max elements out from every child and summary, without representing them recursively.
- **Intuition.** Now Min() and Max(), as well as determining if a tree is empty, Insert/Delete into/from an empty tree, become constant-time ops/queries.
  With this change, we can **always** get the nice recurrence $T(U) \leq \Theta(1) + T(\sqrt{U})$, $T(2) = \Theta(1)$.



Towards van Emde Boas trees — min/max out

- Take min and max elements out from every child and summary, without representing them recursively.
- **Intuition.** Now Min() and Max(), as well as determining if a tree is empty, Insert/Delete into/from an empty tree, become constant-time ops/queries.
  With this change, we can **always** get the nice recurrence $T(U) \leq \Theta(1) + T(\sqrt{U})$, $T(2) = \Theta(1)$.

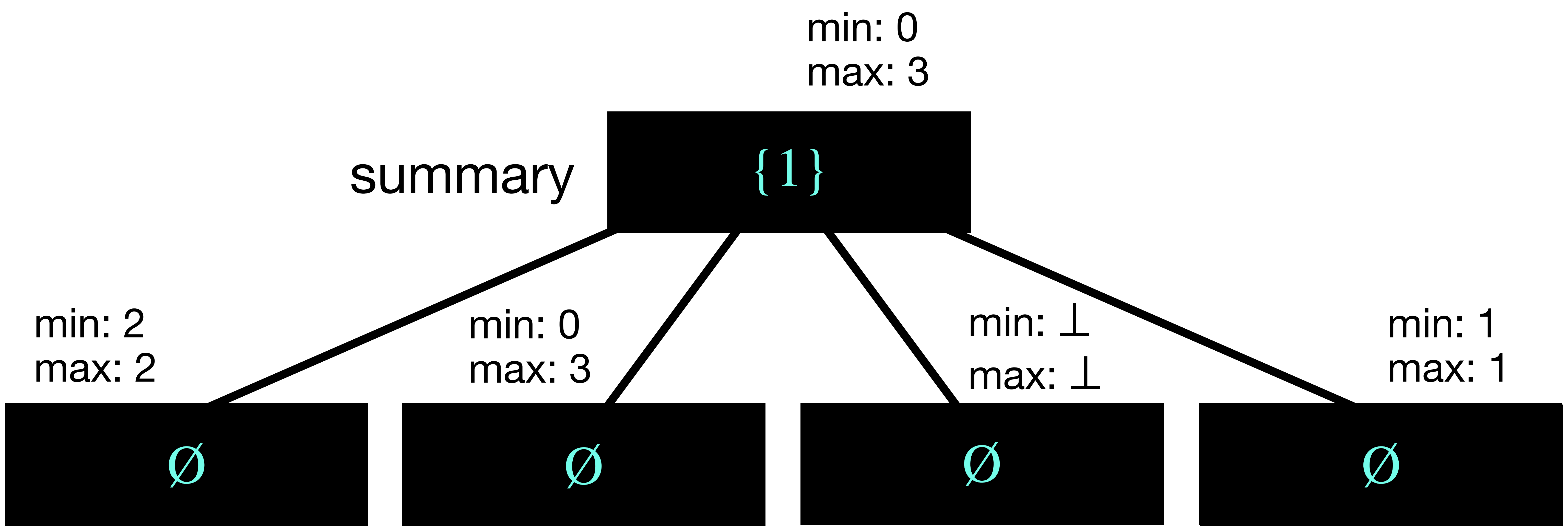


Predecessor(13):

- Retrieve Min from chunk $\lfloor 13/4 \rfloor$. Now in $\Theta(1)$.
- Is Min larger than $13 \bmod 4$?
- No, so answer Predecessor$(\lfloor 13/4 \rfloor) = 1$ on summary.
- Return Max from chunk 1, which is 3. Now in $\Theta(1)$.
- Reconstruct answer: $3 + 1 \times 4 = 7$.

Towards van Emde Boas trees — min/max out

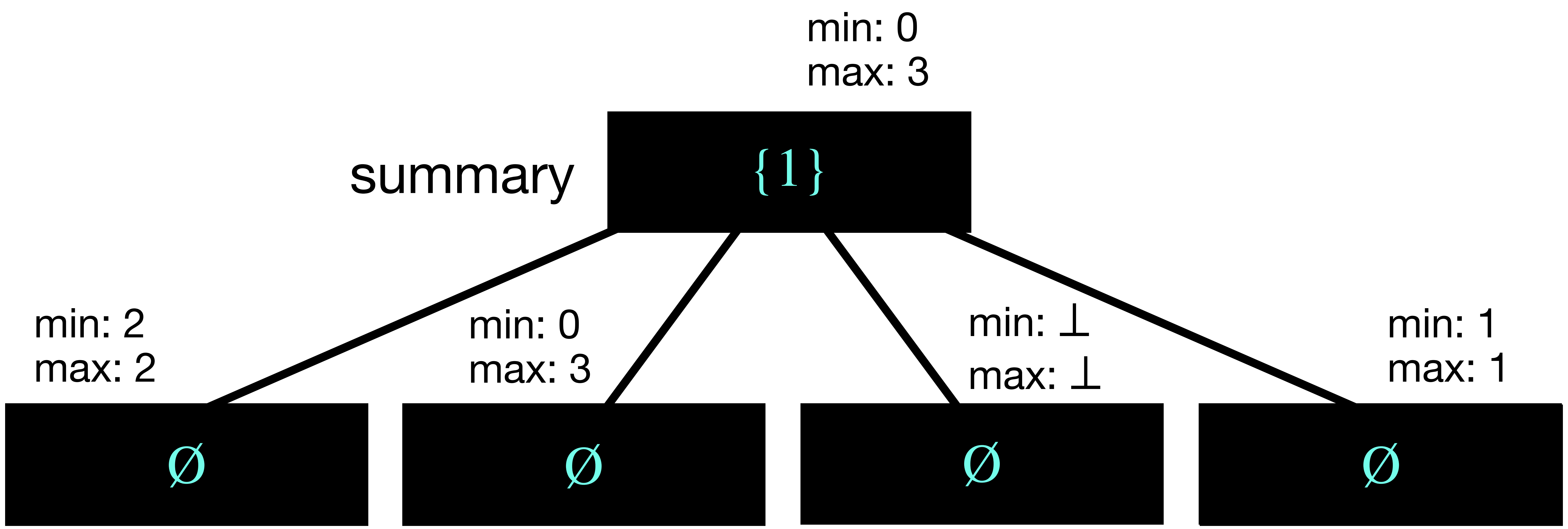
- Take min and max elements out from every child and summary, without representing them recursively.
- **Intuition.** Now Min() and Max(), as well as determining if a tree is empty, Insert/Delete into/from an empty tree, become constant-time ops/queries.
  With this change, we can **always** get the nice recurrence $T(U) \leq \Theta(1) + T(\sqrt{U})$, $T(2) = \Theta(1)$.



Towards van Emde Boas trees — min/max out

- Take min and max elements out from every child and summary, without representing them recursively.
- **Intuition.** Now Min() and Max(), as well as determining if a tree is empty, Insert/Delete into/from an empty tree, become constant-time ops/queries.
  With this change, we can **always** get the nice recurrence $T(U) \leq \Theta(1) + T(\sqrt{U})$, $T(2) = \Theta(1)$.

Insert(9):

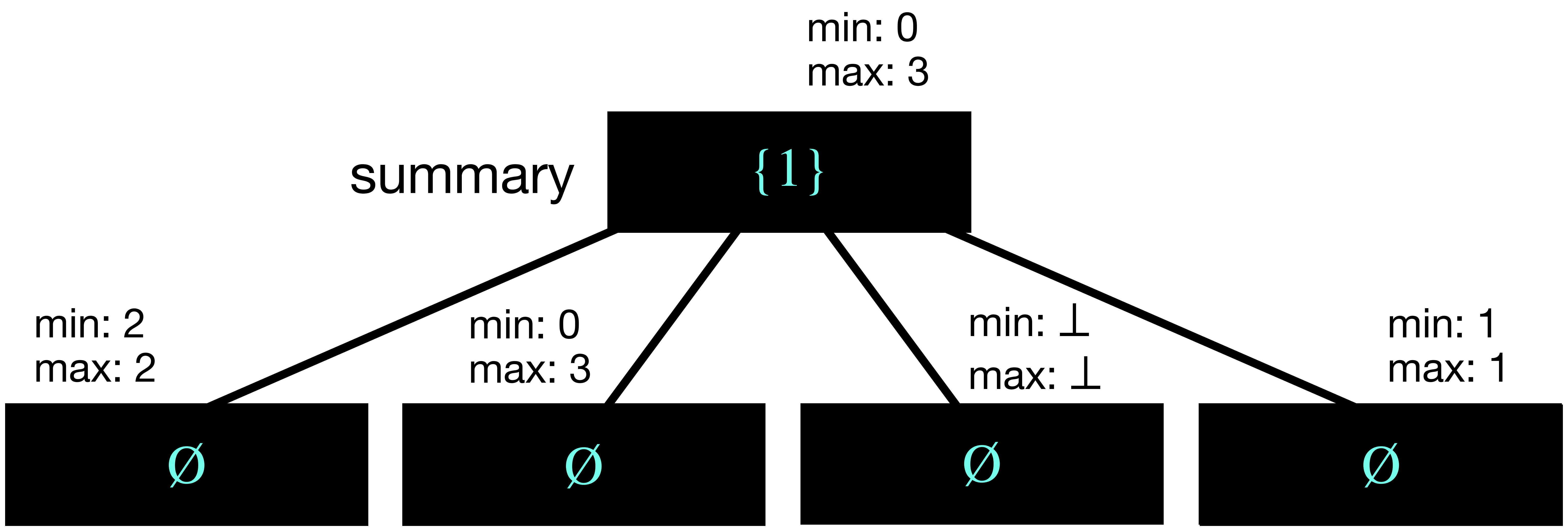


Towards van Emde Boas trees — min/max out

- Take min and max elements out from every child and summary, without representing them recursively.
- **Intuition.** Now Min() and Max(), as well as determining if a tree is empty, Insert/Delete into/from an empty tree, become constant-time ops/queries.
  With this change, we can **always** get the nice recurrence $T(U) \leq \Theta(1) + T(\sqrt{U})$, $T(2) = \Theta(1)$.

Insert(9):

- Is chunk $\lfloor 9/4 \rfloor$ empty? Yes. Now in $\Theta(1)$.

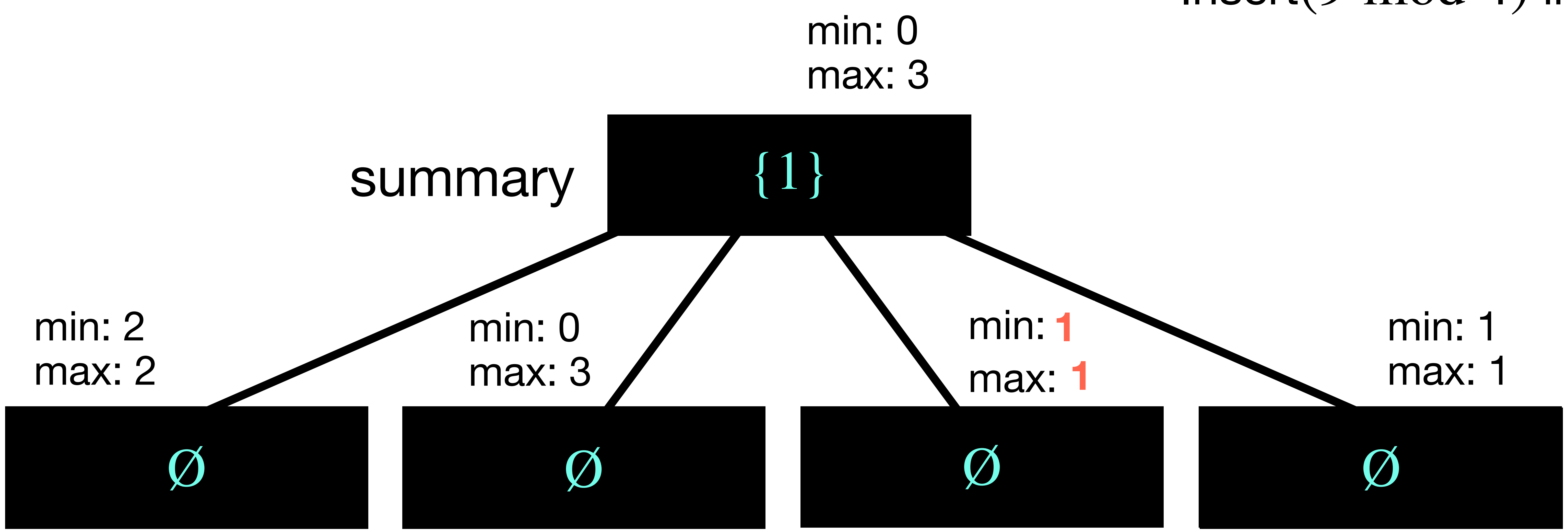


Towards van Emde Boas trees — min/max out

- Take min and max elements out from every child and summary, without representing them recursively.
- **Intuition.** Now Min() and Max(), as well as determining if a tree is empty, Insert/Delete into/from an empty tree, become constant-time ops/queries.
  With this change, we can **always** get the nice recurrence $T(U) \leq \Theta(1) + T(\sqrt{U})$, $T(2) = \Theta(1)$.

Insert(9):

- Is chunk $\lfloor 9/4 \rfloor$ empty? Yes. Now in $\Theta(1)$.
- Insert$(9 \bmod 4)$ in chunk $\lfloor 9/4 \rfloor$. Now in $\Theta(1)$.

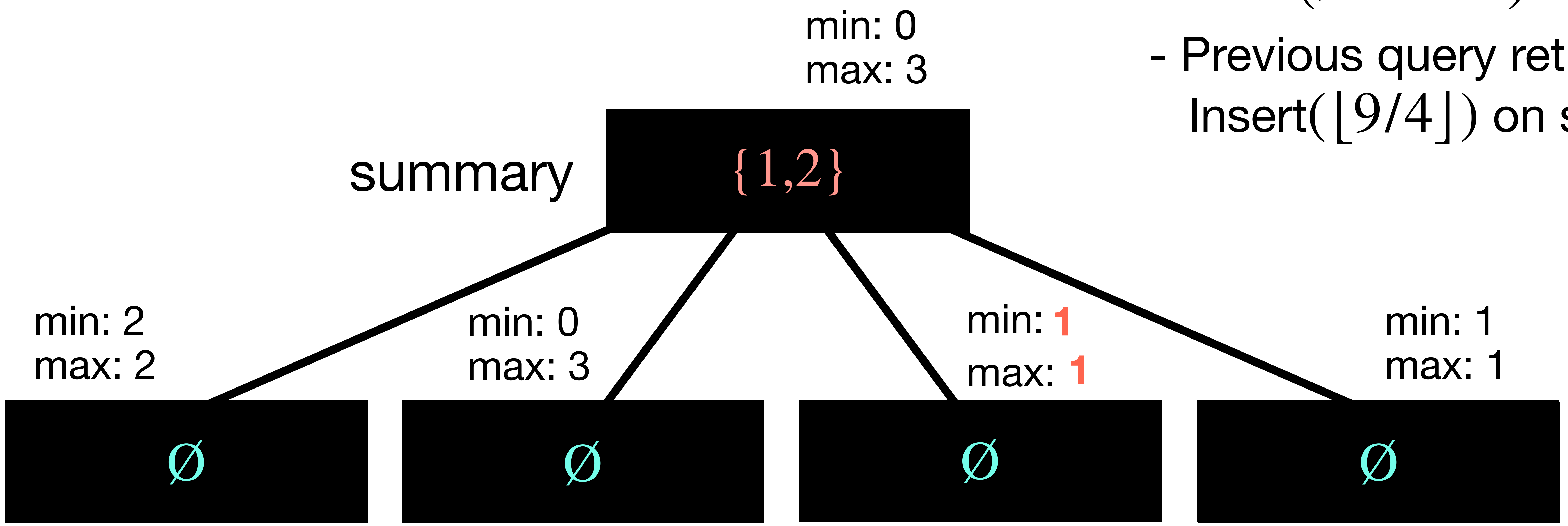


Towards van Emde Boas trees — min/max out

- Take min and max elements out from every child and summary, without representing them recursively.
- **Intuition.** Now Min() and Max(), as well as determining if a tree is empty, Insert/Delete into/from an empty tree, become constant-time ops/queries.
  With this change, we can **always** get the nice recurrence $T(U) \leq \Theta(1) + T(\sqrt{U})$, $T(2) = \Theta(1)$.

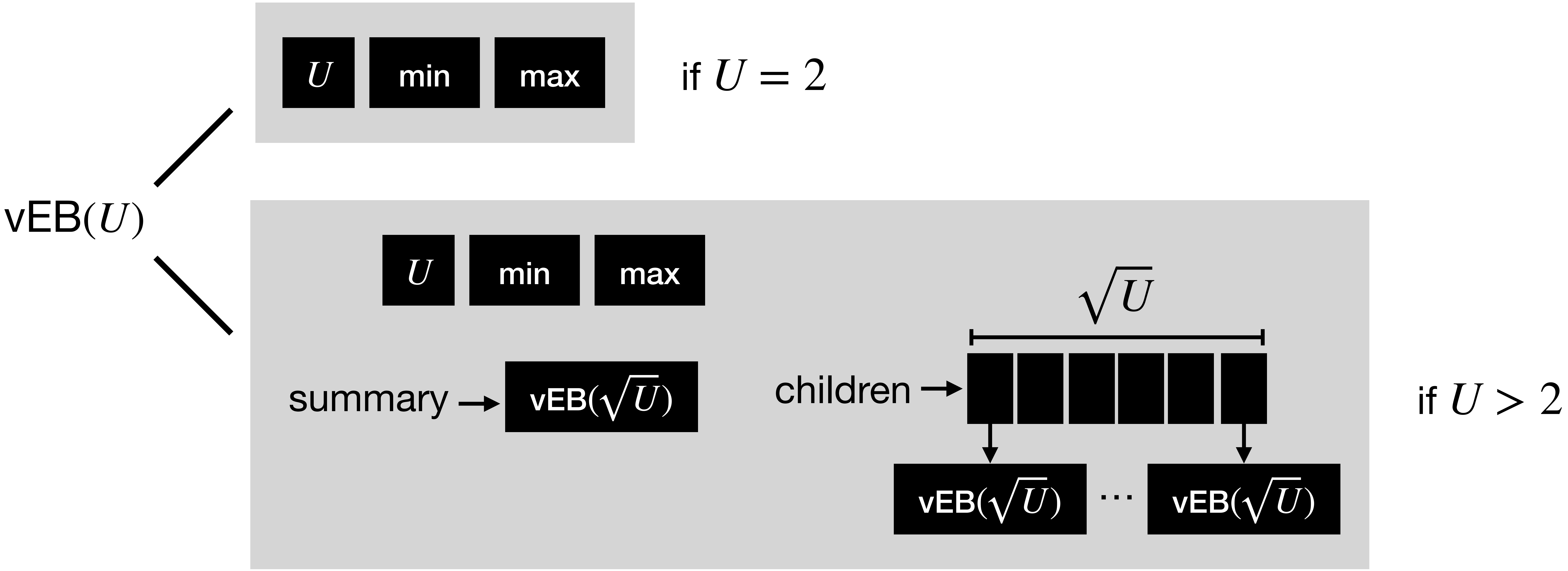
Insert(9):

- Is chunk $\lfloor 9/4 \rfloor$ empty? Yes. Now in $\Theta(1)$.
- Insert$(9 \bmod 4)$ in chunk $\lfloor 9/4 \rfloor$. Now in $\Theta(1)$.
- Previous query returned "Yes", so Insert$(\lfloor 9/4 \rfloor)$ on summary.



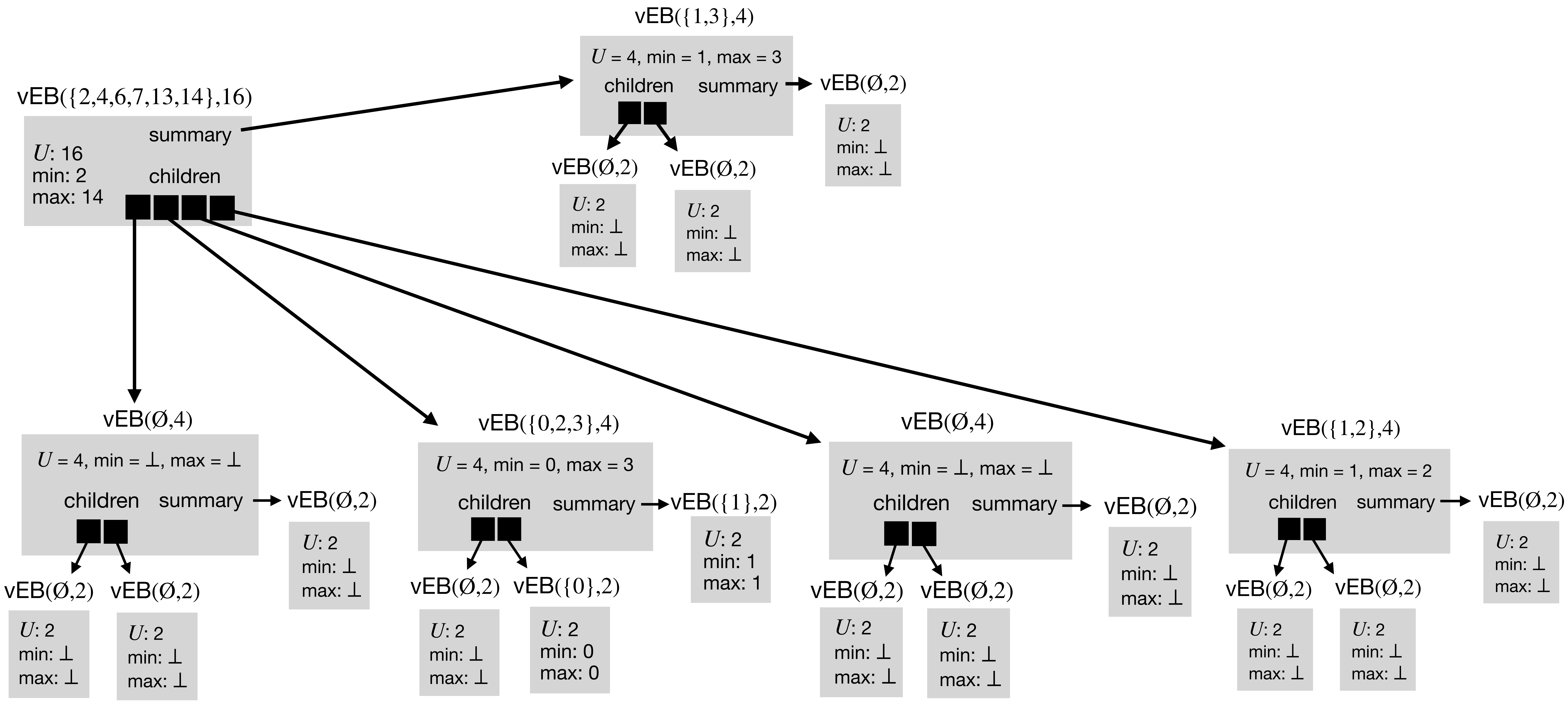
van Emde Boas trees — Rec. definition (graphical)

- A van Emde Boas tree for a universe U, vEB(U), is:



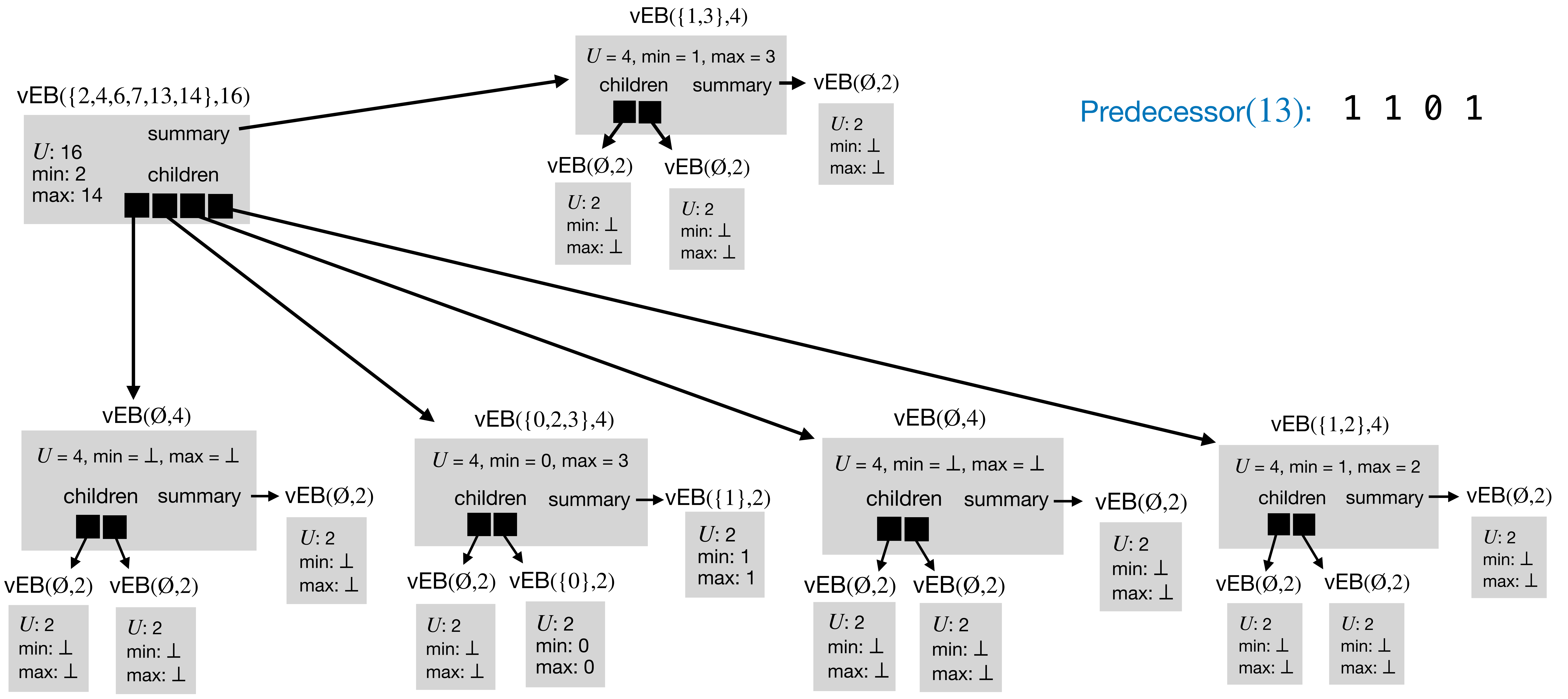
Important note: the min/max elements are not recursively represented in the children.

van Emde Boas trees — Example

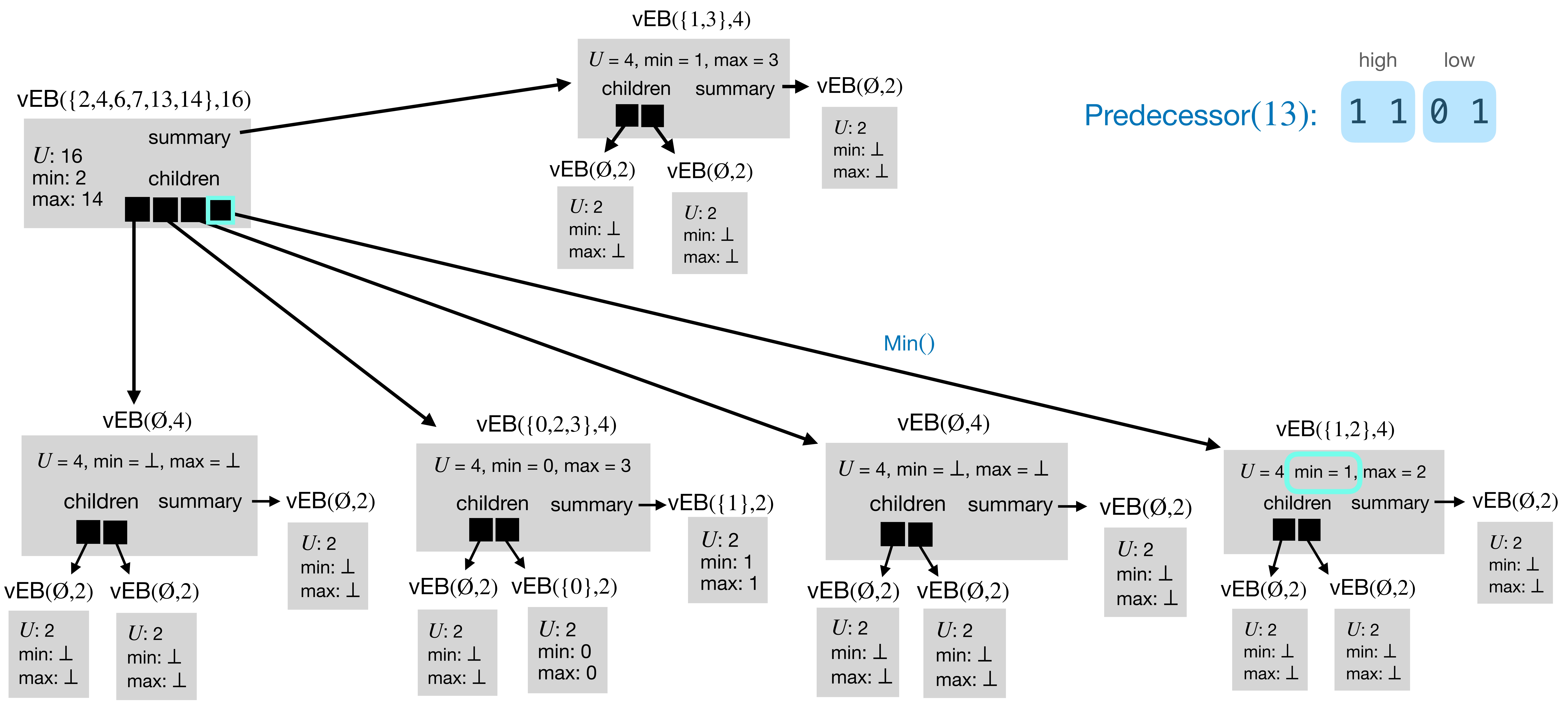


van Emde Boas trees — Example

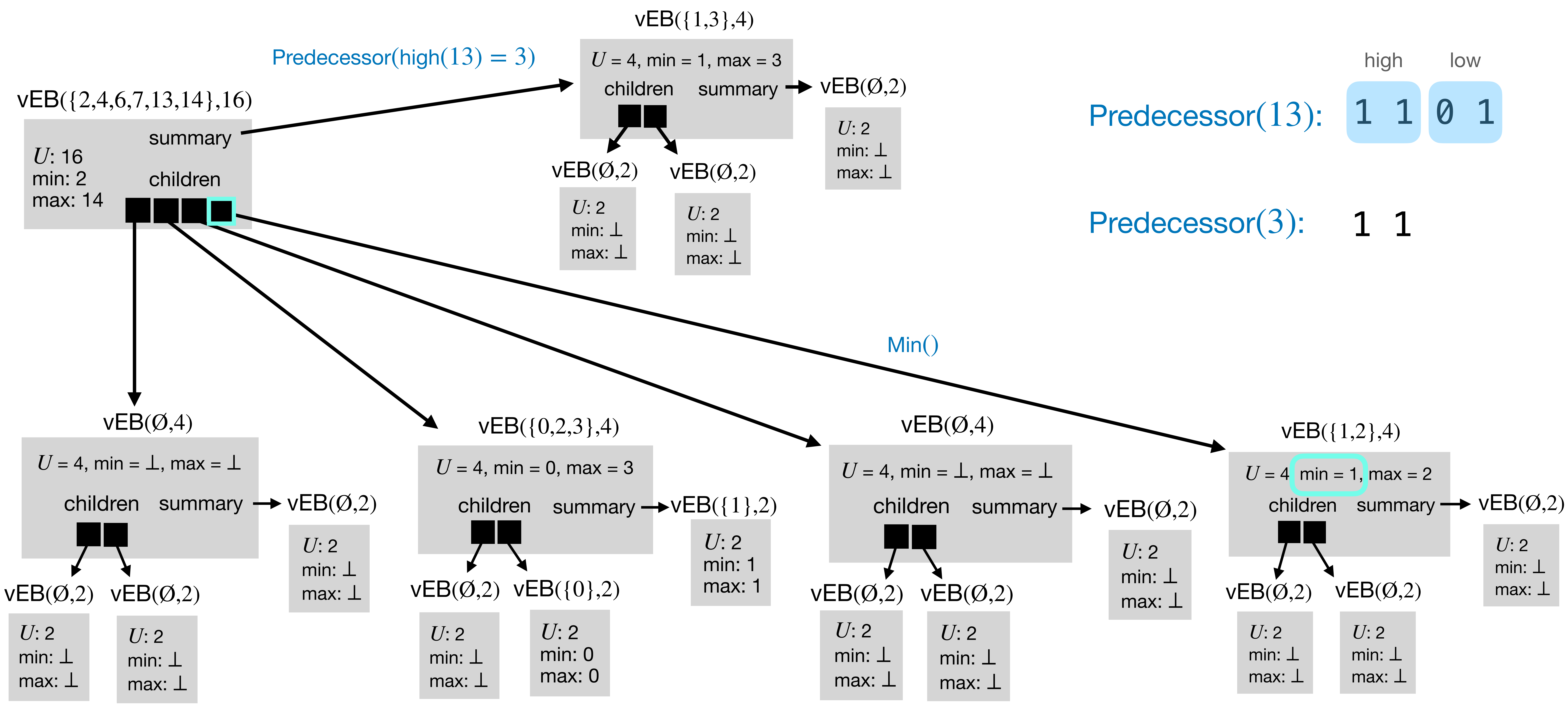
Predecessor(13): 1 1 0 1



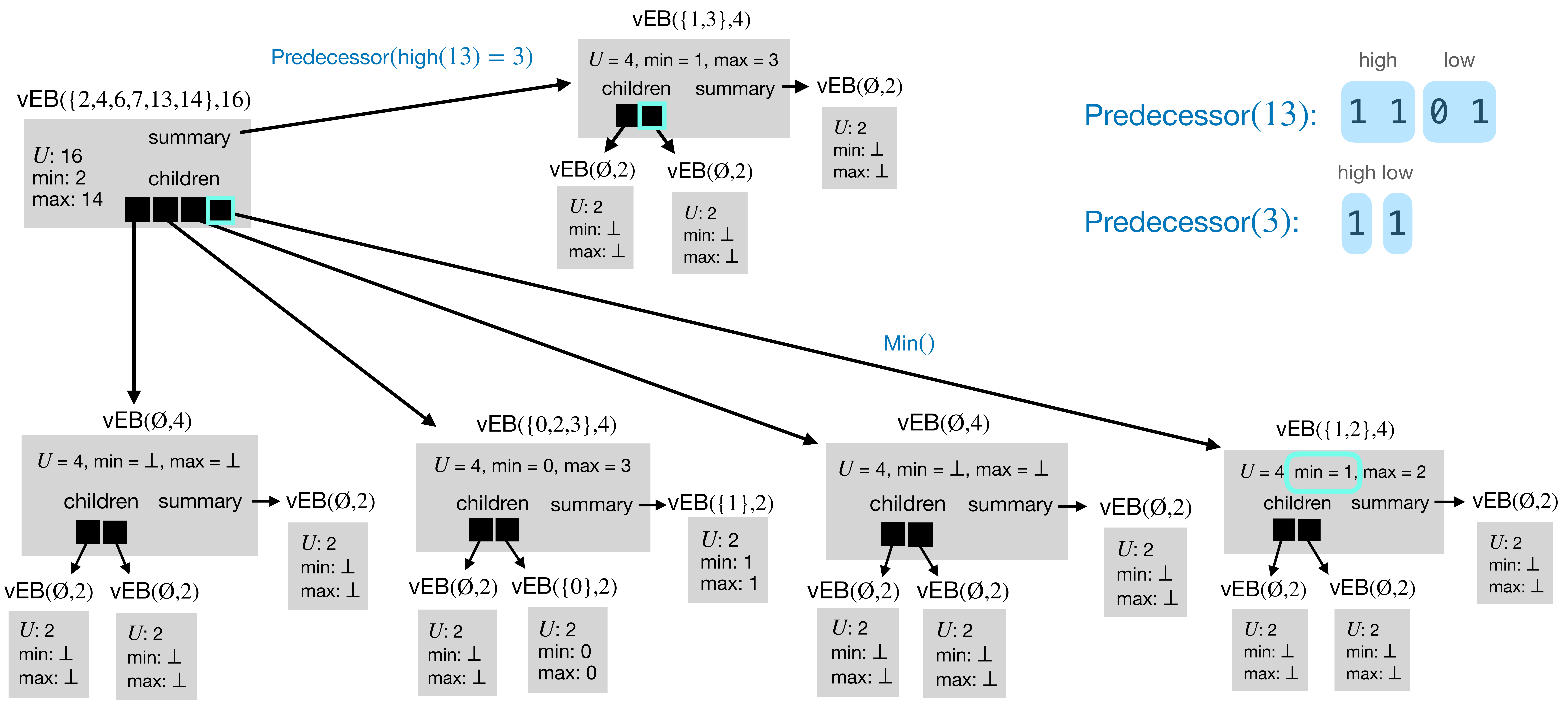
van Emde Boas trees — Example



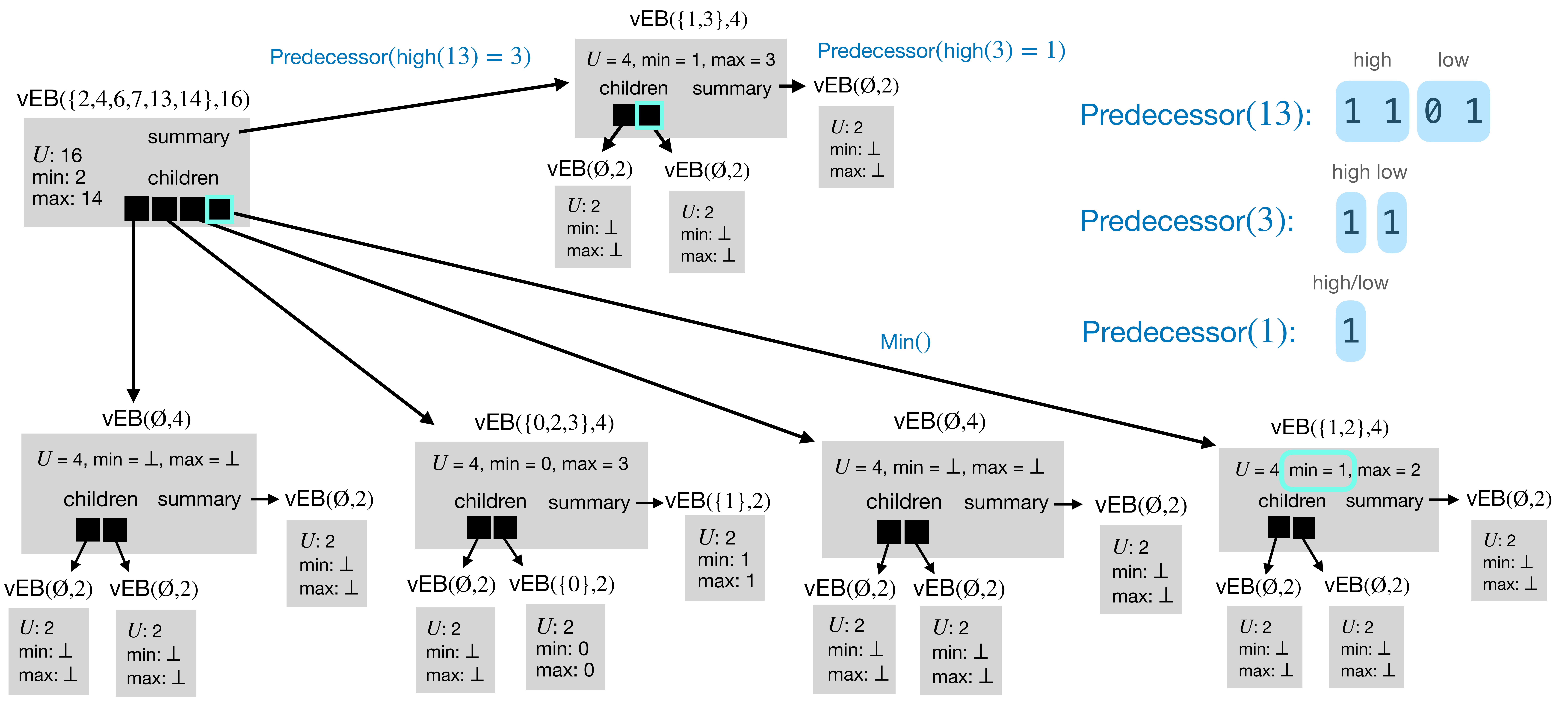
van Emde Boas trees — Example



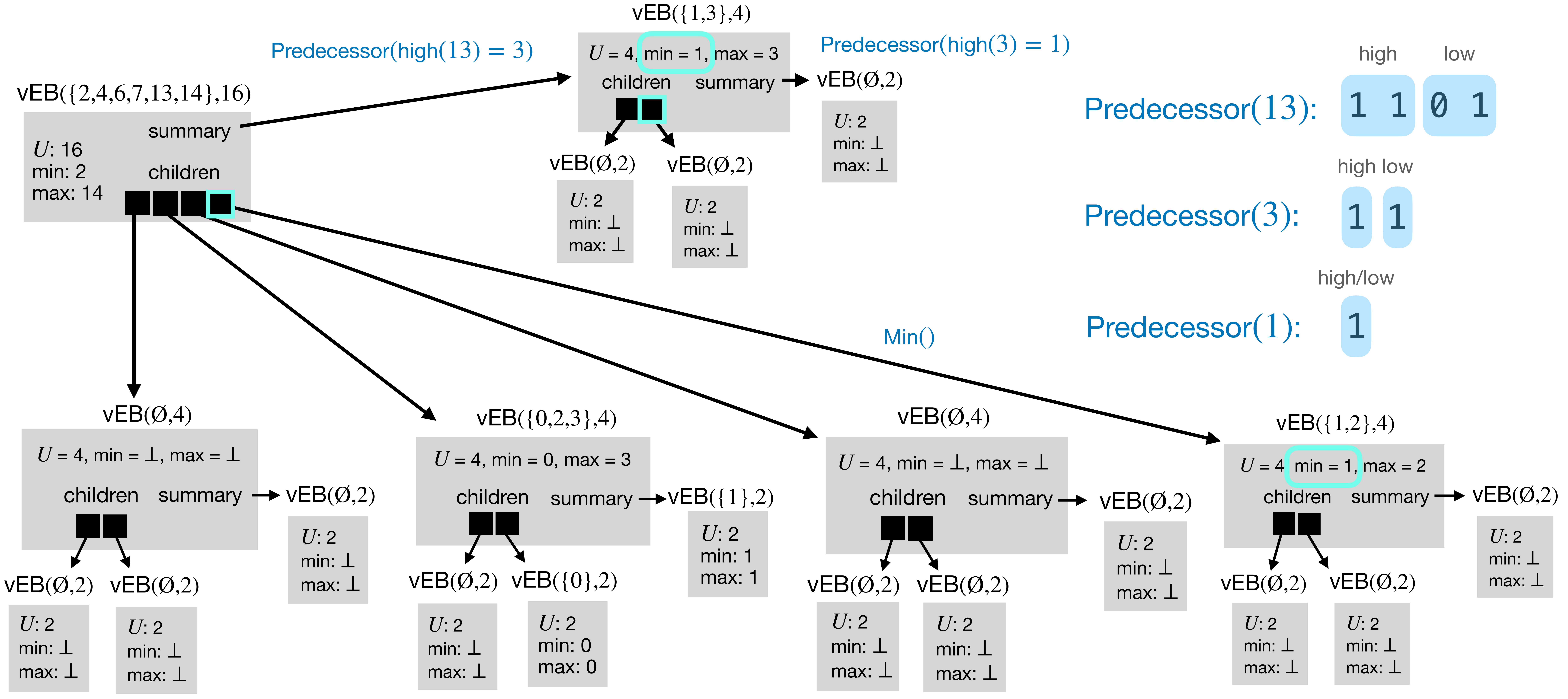
van Emde Boas trees — Example



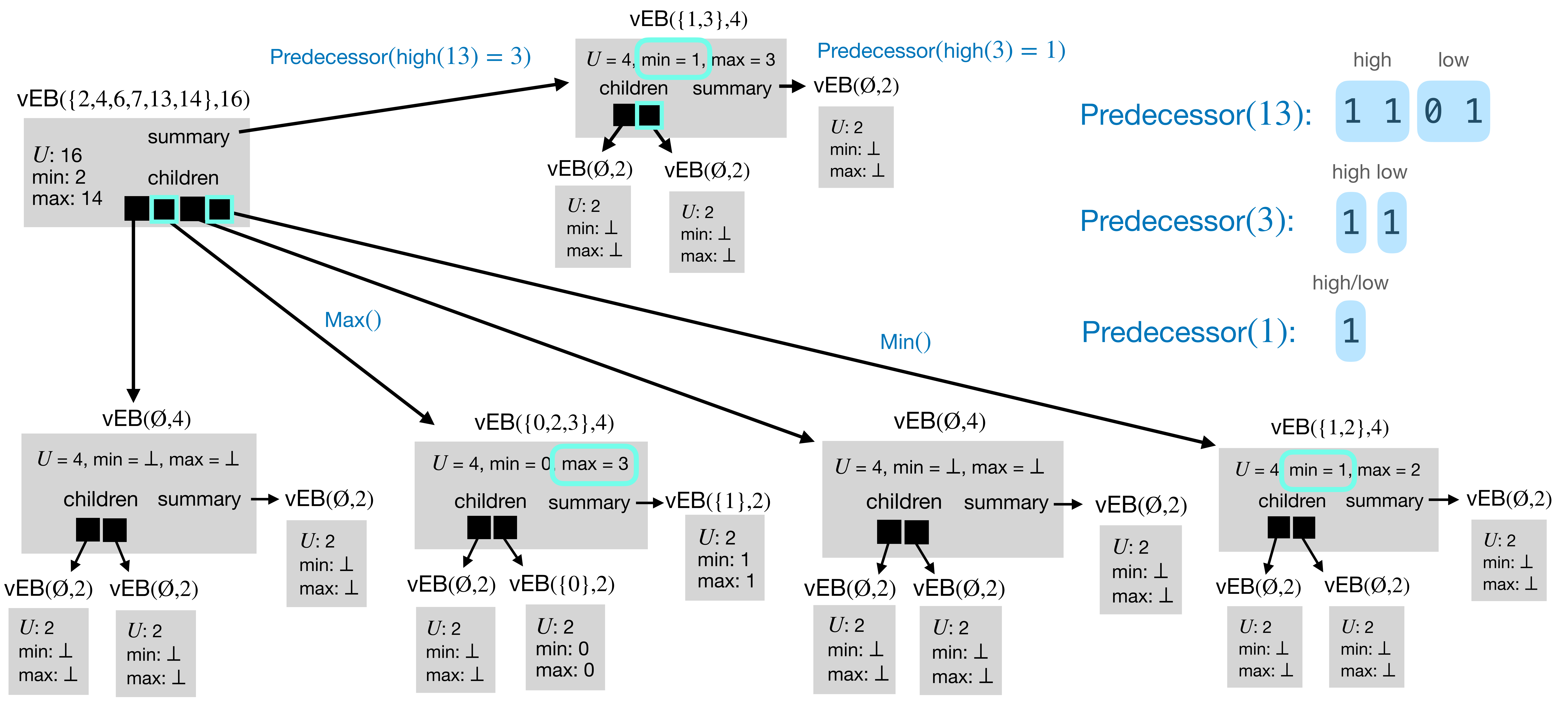
van Emde Boas trees — Example



van Emde Boas trees — Example

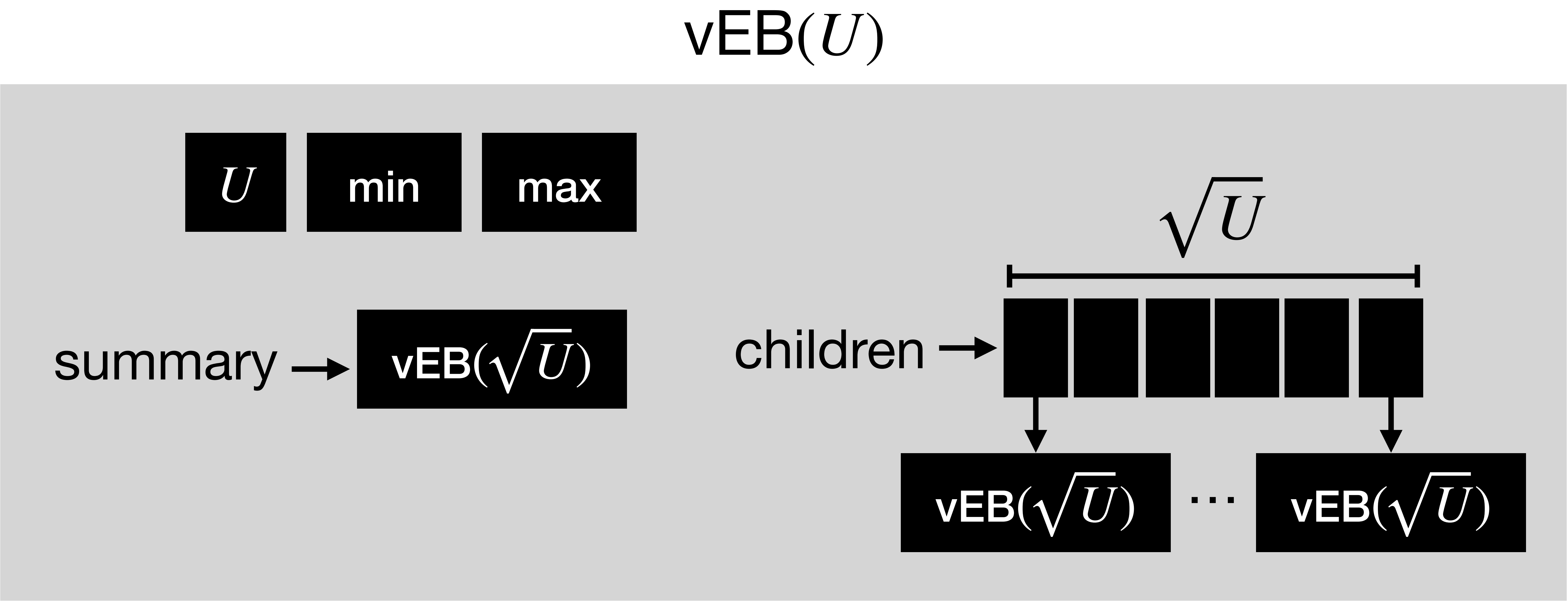


van Emde Boas trees — Example



van Emde Boas trees — Space

- Recall what a vEB(U) stores.



- $S(U) = \underbrace{\Theta(1)}_{U,\ \text{min, max}} + \underbrace{\Theta(\sqrt{U})}_{\text{pointers}} + \underbrace{\sqrt{U} \cdot S(\sqrt{U})}_{\text{children}} + \underbrace{S(\sqrt{U})}_{\text{summary}}$, and $S(2) = \Theta(1)$.

van Emde Boas trees — Space

- $S(U) \simeq \Theta(\sqrt{U}) + \sqrt{U} \cdot S(\sqrt{U})$, and $S(2) = \Theta(1)$.
- Letting $w = \log_2 U$, we have:

by induction

truncated **Kempner** number, $\approx 0.81642.$

$$S(2^w) = \Theta(2^{w/2}) + 2^{w/2} S(2^{w/2}) =$$
$$= \Theta(2^{w/2}) + \Theta(2^{3w/4}) + 2^{3w/4} S(2^{w/4}) =$$
$$= \Theta(2^{w/2}) + \Theta(2^{3w/4}) + \Theta(2^{7w/8}) + 2^{7w/8} S(2^{w/8}) = \dots$$
$$= \Theta\left(\sum_{i=1}^{\log_2 w} 2^{\frac{2^i - 1}{2^i} w}\right) = \Theta\left(2^w \cdot \sum_{i=1}^{\log_2 w} 2^{-w/2^i}\right) = \Theta\left(2^w \cdot \boxed{\sum_{j=0}^{\log_2 w - 1} 2^{-2^j}}\right) = \Theta(2^w) = \Theta(U)\,.$$

by linearity

change variable $j = \log_2 w - i$

Summary

- The vEB tree maintains a sorted integer set, whose elements are less than a known quantity U, in worst-case time $O(\log\log U)$ and space $\Theta(U)$. (It can be built in $O(U)$ time: solution to the recurrence $T(U) = T(\sqrt{U}) + \sqrt{U} \cdot T(\sqrt{U})$, with $T(2) = \Theta(1)$.)

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

Summary

- The vEB tree maintains a sorted integer set, whose elements are less than a known quantity U, in worst-case time $O(\log \log U)$ and space $\Theta(U)$. (It can be built in $O(U)$ time: solution to the recurrence $T(U) = T(\sqrt{U}) + \sqrt{U} \cdot T(\sqrt{U})$, with $T(2) = \Theta(1)$.)

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

- **Q.** Is this better than a balanced search tree?

Summary

- The vEB tree maintains a sorted integer set, whose elements are less than a known quantity U, in worst-case time $O(\log\log U)$ and space $\Theta(U)$. (It can be built in $O(U)$ time: solution to the recurrence $T(U) = T(\sqrt{U}) + \sqrt{U} \cdot T(\sqrt{U})$, with $T(2) = \Theta(1)$.)

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

- **Q.** Is this better than a balanced search tree?

- **A.** It depends on the relationship between n (number of keys currently in the set) and U.

Summary

- The vEB tree maintains a sorted integer set, whose elements are less than a known quantity U, in worst-case time $O(\log\log U)$ and space $\Theta(U)$. (It can be built in $O(U)$ time: solution to the recurrence $T(U) = T(\sqrt{U}) + \sqrt{U} \cdot T(\sqrt{U})$, with $T(2) = \Theta(1)$.)

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

- **Q.** Is this better than a balanced search tree?

- **A.** It depends on the relationship between n (number of keys currently in the set) and U.

- For example, if $U = O(2^n)$ then $\log_2 \log_2 U = O(\log n)$ and vEB trees are **not** asymptotically faster than AVL nor RB trees.

Summary

- The vEB tree maintains a sorted integer set, whose elements are less than a known quantity U, in worst-case time $O(\log\log U)$ and space $\Theta(U)$. (It can be built in $O(U)$ time: solution to the recurrence $T(U) = T(\sqrt{U}) + \sqrt{U} \cdot T(\sqrt{U})$, with $T(2) = \Theta(1)$.)

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

- **Q.** Is this better than a balanced search tree?

- **A.** It depends on the relationship between n (number of keys currently in the set) and U.

- For example, if $U = O(2^n)$ then $\log_2 \log_2 U = O(\log n)$ and vEB trees are **not** asymptotically faster than AVL nor RB trees.

- However, if $U = n^c$ 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.

Optimal predecessor search

- In 2006, Pătrașcu and Thorup published a landmark result

  Time-Space Trade-Offs for Predecessor Search, STOC 2006

  showing that **vEB is optimal for predecessor search** for polynomial universes ($U = n^c, c \geq 1$).

- Intuition: if the set is dense (n is close to U), it is faster to search over the binary representation of the integers directly, like vEB. If, instead, the set is sparse, we should just search the keys.

Mihai Pătrașcu

Mikkel Thorup

Caveats and issues

- Caveat: U must be known in advance and cannot change.
- **Issue: Space usage $\Theta(U)$.**

  Not practical…

Caveats and issues

- Caveat: U must be known in advance and cannot change.
- **Issue: Space usage $\Theta(U)$.**

  Not practical…

  But its principles inspired other data structures.

Caveats and issues

- Caveat: U must be known in advance and cannot change.
- **Issue: Space usage $\Theta(U)$.**

  Not practical…

  But its principles inspired other data structures.

- Next week (10 Feb) spoiler:
  ***y*-fast tries** combines $O(\log \log U)$ worst-case query time with $\Theta(n)$ space.
  (Insert/Delete is $O(\log \log U)$ amortised.)

References

- 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.

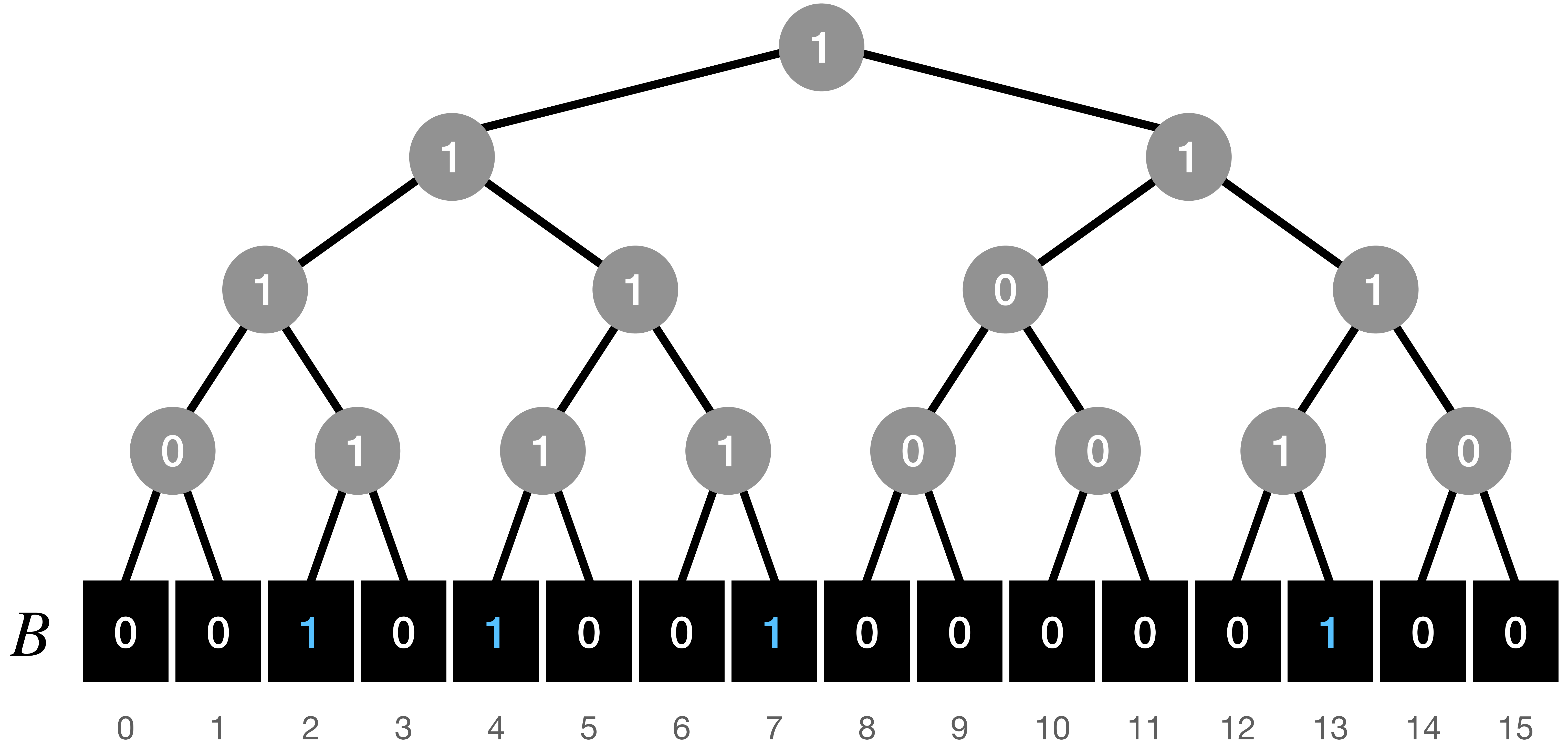
- M. Pătraşcu, and M. Thorup. *Time-space trade-offs for predecessor search*, STOC, 2006.

Bonus slides

Imposing a binary tree

- **Idea.** Impose a complete binary tree on top of B. The leaves of the tree correspond to the bits of B; an internal node stores the logical OR between the bits of its children.

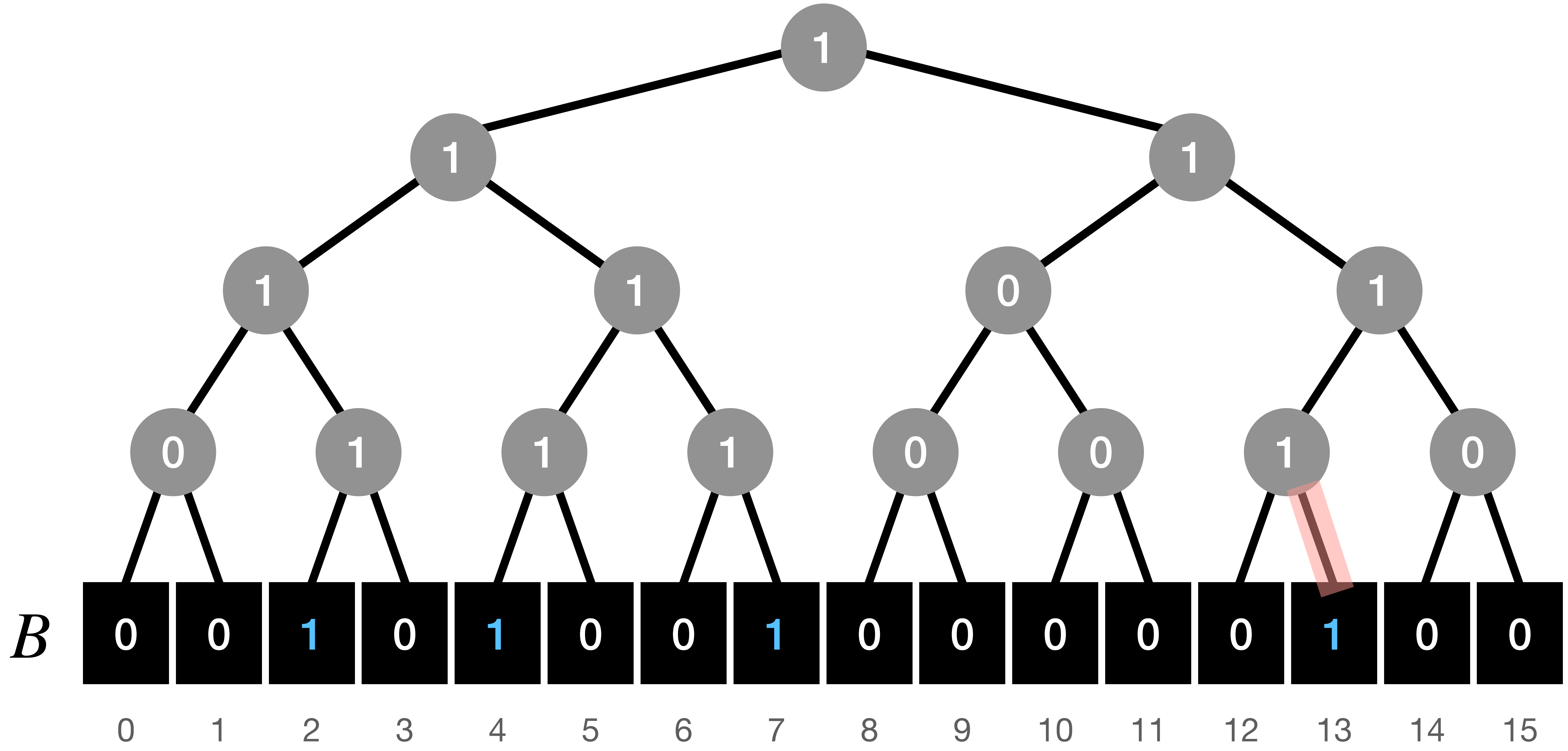
- **Intuition.** Use the tree to **avoid scanning long runs on zeros** upon Predecessor/Successor.



- Predecessor(x): from $B[x]$, navigate up in the tree until we enter a node v from the **right** that has a 1 in its **left** child. Then return the **max** of the subtree rooted in v . left. (If v is the root and the bit in v . left is 0, return $\perp$.)

Imposing a binary tree

- **Idea.** Impose a complete binary tree on top of B. The leaves of the tree correspond to the bits of B; an internal node stores the logical OR between the bits of its children.
- **Intuition.** Use the tree to **avoid scanning long runs on zeros** upon Predecessor/Successor.

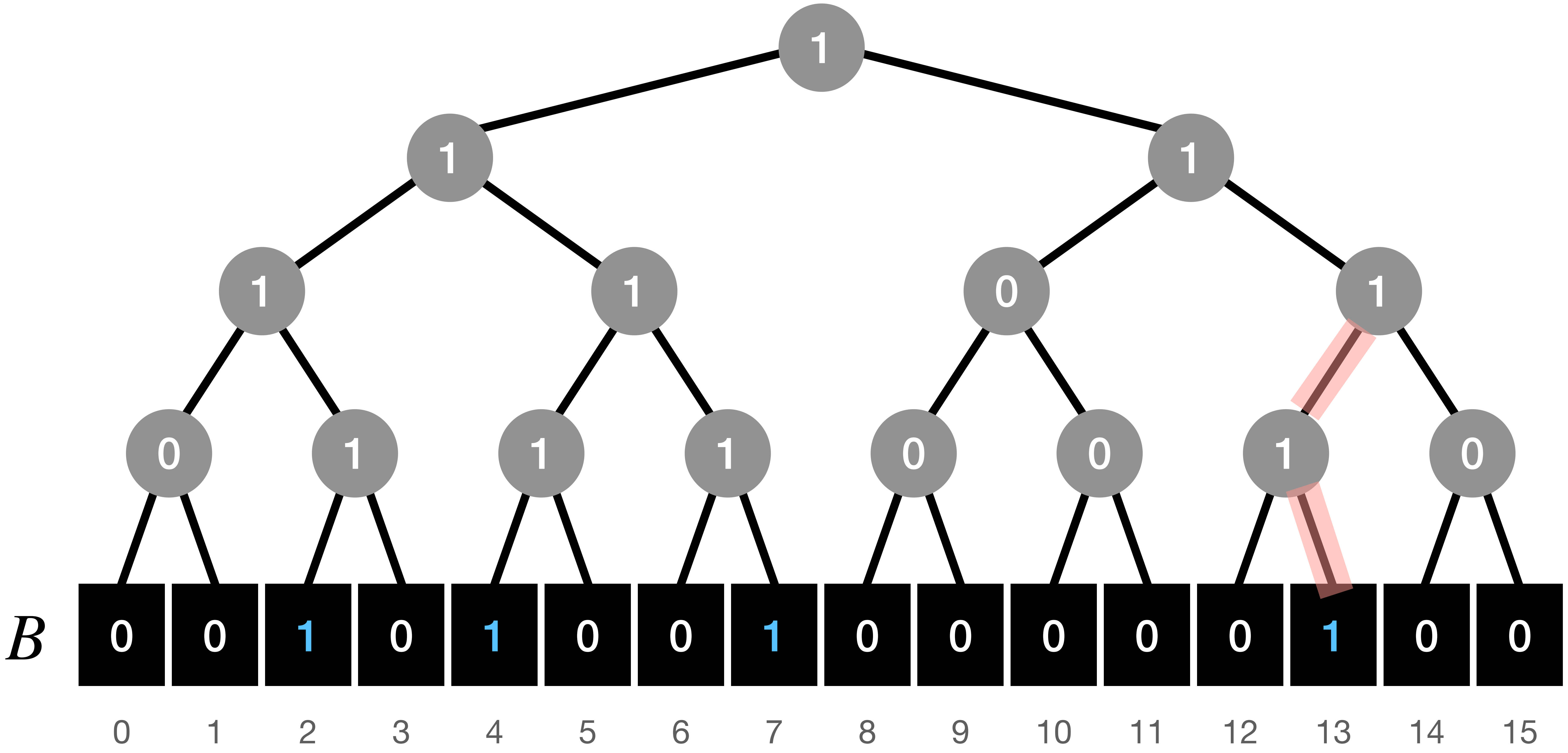


- Predecessor(x): from $B[x]$, navigate up in the tree until we enter a node v from the **right** that has a 1 in its **left** child. Then return the **max** of the subtree rooted in v . left. (If v is the root and the bit in v . left is 0, return ⊥.)

Imposing a binary tree

- **Idea.** Impose a complete binary tree on top of B. The leaves of the tree correspond to the bits of B; an internal node stores the logical OR between the bits of its children.

- **Intuition.** Use the tree to **avoid scanning long runs on zeros** upon Predecessor/Successor.

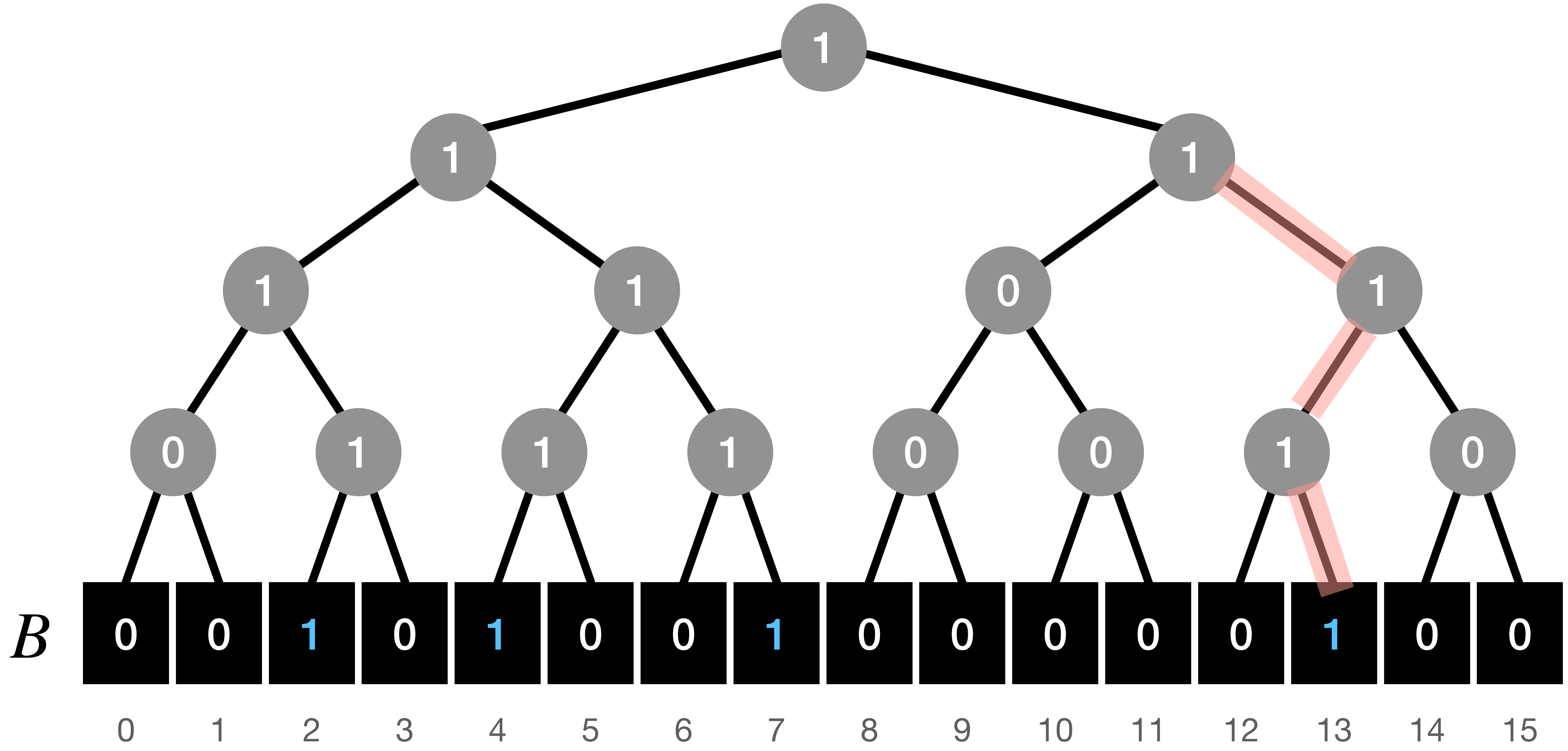


- Predecessor(x): from $B[x]$, navigate up in the tree until we enter a node v from the **right** that has a 1 in its **left** child. Then return the **max** of the subtree rooted in v . left. (If v is the root and the bit in v . left is 0, return ⊥.)

Imposing a binary tree

- **Idea.** Impose a complete binary tree on top of B. The leaves of the tree correspond to the bits of B; an internal node stores the logical OR between the bits of its children.

- **Intuition.** Use the tree to **avoid scanning long runs on zeros** upon Predecessor/Successor.

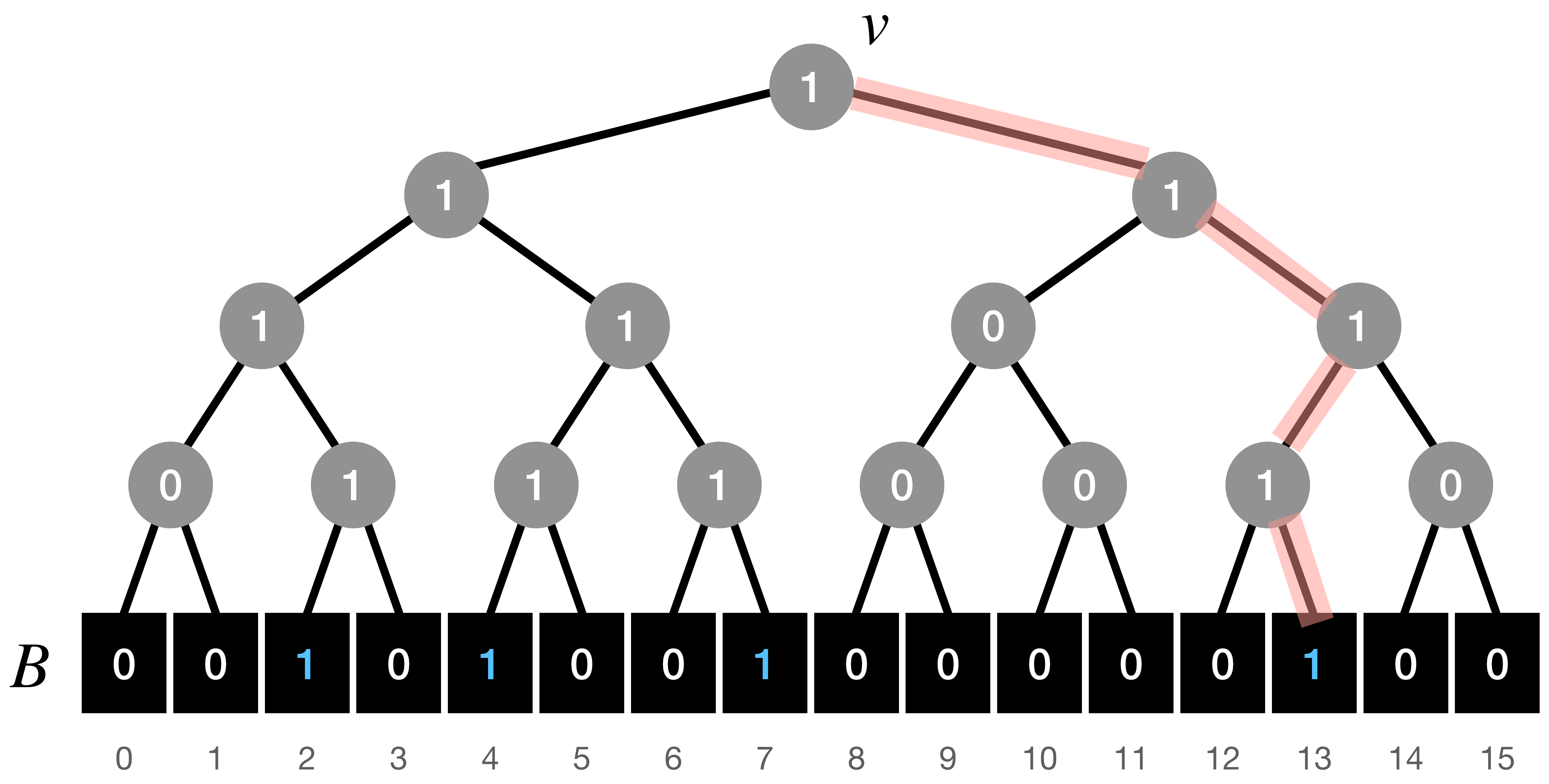


- Predecessor(x): from $B[x]$, navigate up in the tree until we enter a node v from the **right** that has a 1 in its **left** child. Then return the **max** of the subtree rooted in v.left. (If v is the root and the bit in v.left is 0, return $\perp$.)

Imposing a binary tree

- **Idea.** Impose a complete binary tree on top of B. The leaves of the tree correspond to the bits of B; an internal node stores the logical OR between the bits of its children.

- **Intuition.** Use the tree to **avoid scanning long runs on zeros** upon Predecessor/Successor.

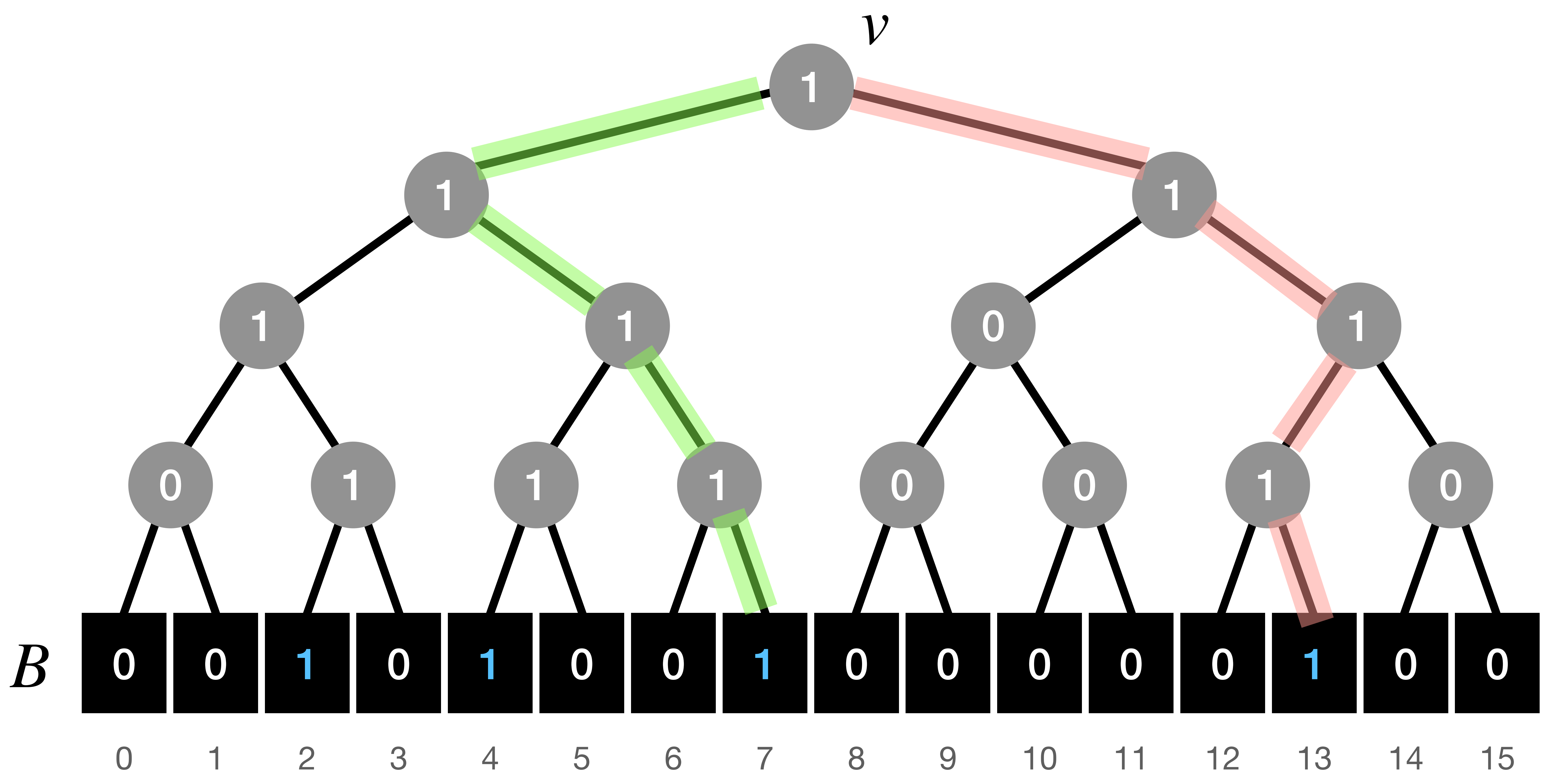


- Predecessor(x): from $B[x]$, navigate up in the tree until we enter a node v from the **right** that has a 1 in its **left** child. Then return the **max** of the subtree rooted in v . left. (If v is the root and the bit in v . left is 0, return $\perp$.)

Imposing a binary tree

- **Idea.** Impose a complete binary tree on top of B. The leaves of the tree correspond to the bits of B; an internal node stores the logical OR between the bits of its children.

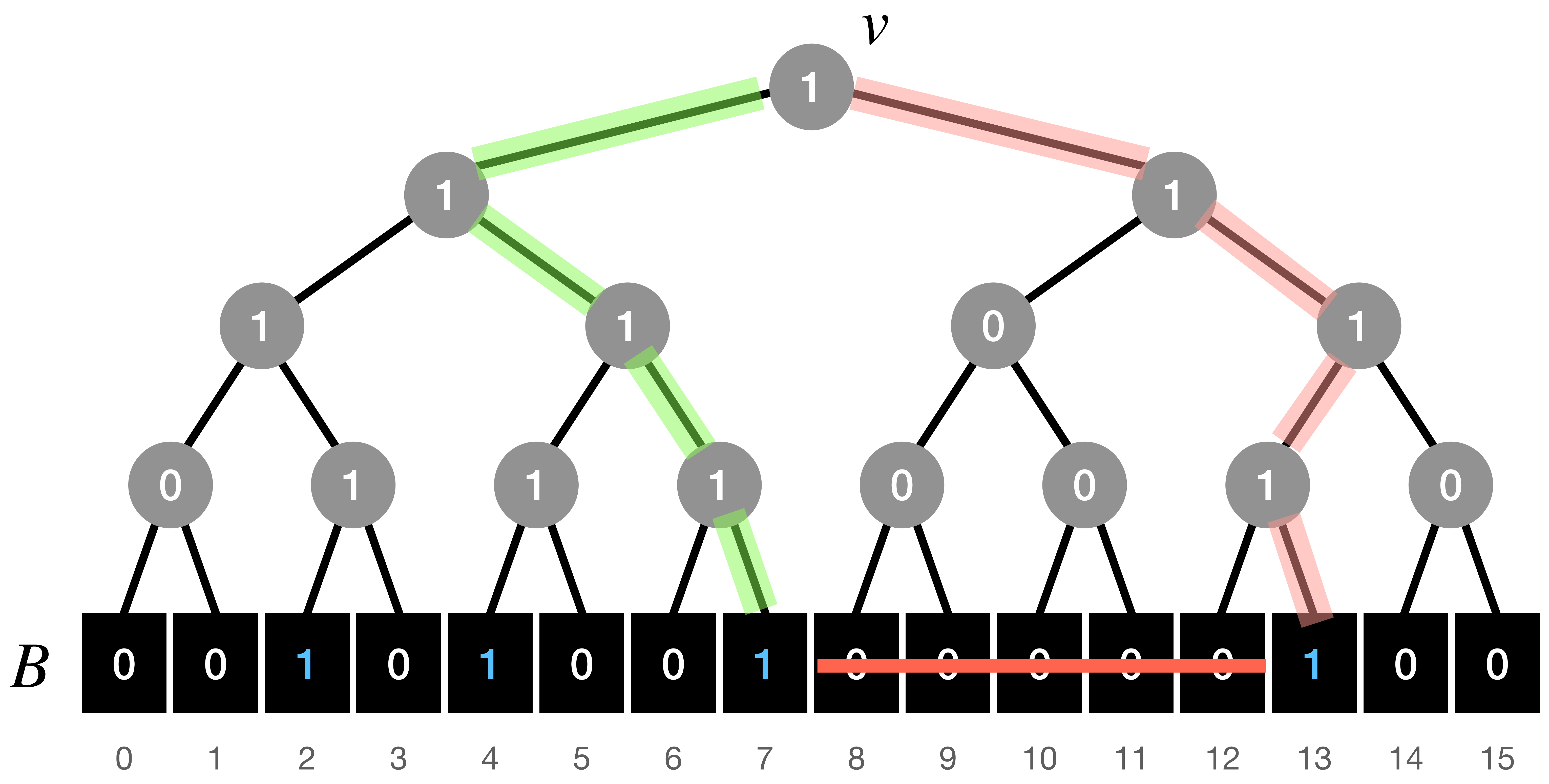
- **Intuition.** Use the tree to **avoid scanning long runs on zeros** upon Predecessor/Successor.



- Predecessor(x): from $B[x]$, navigate up in the tree until we enter a node v from the **right** that has a 1 in its **left** child. Then return the **max** of the subtree rooted in v.left. (If v is the root and the bit in v.left is 0, return ⊥.)

Imposing a binary tree

- **Idea.** Impose a complete binary tree on top of B. The leaves of the tree correspond to the bits of B; an internal node stores the logical OR between the bits of its children.
- **Intuition.** Use the tree to **avoid scanning long runs on zeros** upon Predecessor/Successor.

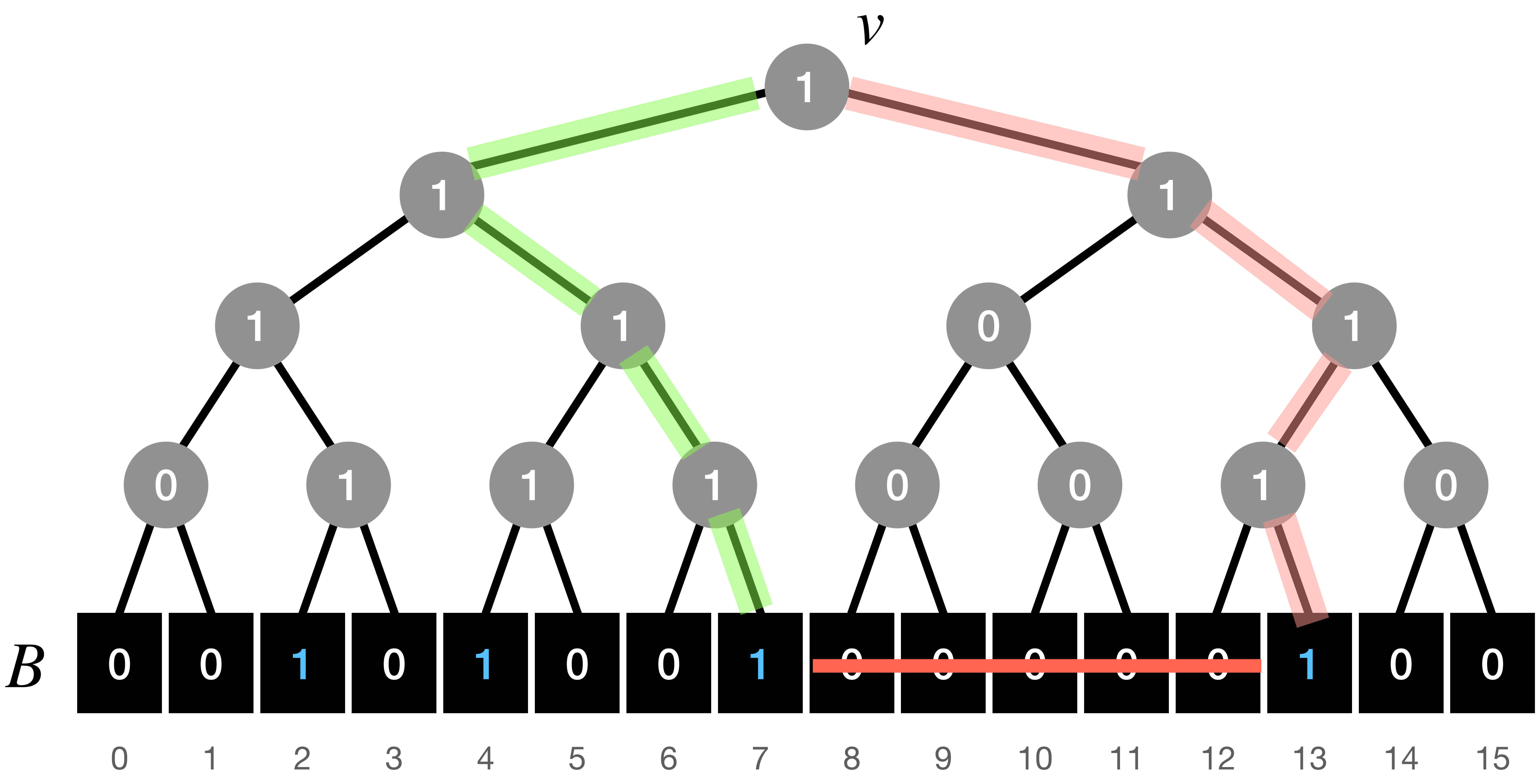


- Predecessor(x): from $B[x]$, navigate up in the tree until we enter a node v from the **right** that has a 1 in its **left** child. Then return the **max** of the subtree rooted in v . left. (If v is the root and the bit in v . left is 0, return ⊥.)

Imposing a binary tree

- **Idea.** Impose a complete binary tree on top of B. The leaves of the tree correspond to the bits of B; an internal node stores the logical OR between the bits of its children.

- **Intuition.** Use the tree to **avoid scanning long runs on zeros** upon Predecessor/Successor.



- Predecessor(x): from $B[x]$, navigate up in the tree until we enter a node v from the **right** that has a 1 in its **left** child. Then return the **max** of the subtree rooted in v.left. (If v is the root and the bit in v.left is 0, return $\perp$.)

The height of the tree is $\log_2 U$ so all ops/queries run in $O(\log U)$.