van Emde Boas Trees

Outline for Today

- ***Data Structures on Integers***
  - How can we speed up operations that work on integer data?
- ***Tiered Bitvectors***
  - A simple data structure for ordered dictionaries.
- ***van Emde Boas Trees***
  - An extremely fast data structure for ordered dictionaries.

Integer Data Structures

Working with Integers

- Integers are interesting objects to work with:
  - They can be treated as strings of bits, so we can use techniques from string processing.
  - They fit into machine words, so we can process the bits in parallel with individual word operations.
- Today, we'll explore ***van Emde Boas trees***, which rely on this second property.
- On Thursday, we'll see ***y-Fast tries***, which will pull together just about everything from the quarter.

Our Machine Model

- We will assume that we are working with a ***transdichotomous machine model***.
- Memory is split apart into integer words composed of w bits each.
- The CPU can perform basic arithmetic operations (addition, subtraction, multiplication, division, shifts, AND, OR, etc.) on machine words in time O(1) each.
- When working on a problem where each instance has size n, we assume $w = \Omega(\log n)$.

Ordered Dictionaries

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.

Ordered Dictionaries

- Balanced BSTs support all ordered dictionary operations in time O(log n) each.
- Hash tables support insertion, lookups, and deletion in expected time O(1), but require time O(n) for min, max, successor, and predecessor.

Ordered Integer Dictionaries

- Suppose that our universe consists of natural numbers upper-bounded by some number *U*.
  - Specifically, $\mathscr{U} = [U] = \{0, 1, 2, \ldots, U - 1\}$.
  - In our analysis, we'll assume that *U* fits into O(1) machine words.
- ***Question:*** Can we design a data structure that supports the ordered dictionary operations on $\mathscr{U}$ faster than a balanced BST?
- The answer is *yes*, and we'll see van Emde Boas trees and *y*-fast tries as two possible solutions.

A Preliminary Approach: ***Bitvectors***

Bitvectors

- A ***bitvector*** is an array of bits of length U.
- Represents a set of elements with insertions, deletions, and lookups each taking time O(1):
  - To insert x, set the bit for x to 1.
  - To delete x, set the bit for x to 0.
  - To lookup x, check whether the bit for x is 1.
- Space usage is $\Theta(U)$.

1101110010111011110001001101010111100110111101111

Bitvectors

- A ***bitvector*** is an array of bits of length U.
- Represents a set of elements with insertions, deletions, and lookups each taking time O(1):
  - To insert x, set the bit for x to 1.
  - To delete x, set the bit for x to 0.
  - To lookup x, check whether the bit for x is 1.
- Space usage is $\Theta(U)$.

11011<u>1</u>0010111011110001001101010111100110111101111

Bitvectors

- A ***bitvector*** is an array of bits of length U.
- Represents a set of elements with insertions, deletions, and lookups each taking time O(1):
  - To insert x, set the bit for x to 1.
  - To delete x, set the bit for x to 0.
  - To lookup x, check whether the bit for x is 1.
- Space usage is $\Theta(U)$.

11011<u>0</u>0010111011110001001101010111100110111101111

Bitvectors

- A ***bitvector*** is an array of bits of length U.
- Represents a set of elements with insertions, deletions, and lookups each taking time O(1):
  - To insert x, set the bit for x to 1.
  - To delete x, set the bit for x to 0.
  - To lookup x, check whether the bit for x is 1.
- Space usage is $\Theta(U)$.

1101100010111011110001001101010111100110111101111

Bitvectors

- A ***bitvector*** is an array of bits of length U.
- Represents a set of elements with insertions, deletions, and lookups each taking time O(1):
  - To insert x, set the bit for x to 1.
  - To delete x, set the bit for x to 0.
  - To lookup x, check whether the bit for x is 1.
- Space usage is $\Theta(U)$.

1101100010111011110<u>0</u>0100110101011100110111101111

Bitvectors

- A ***bitvector*** is an array of bits of length U.
- Represents a set of elements with insertions, deletions, and lookups each taking time O(1):
  - To insert x, set the bit for x to 1.
  - To delete x, set the bit for x to 0.
  - To lookup x, check whether the bit for x is 1.
- Space usage is $\Theta(U)$.

```
11011000101110111101010011010101111001101111 01111
```

Bitvectors

- A ***bitvector*** is an array of bits of length U.
- Represents a set of elements with insertions, deletions, and lookups each taking time O(1):
  - To insert x, set the bit for x to 1.
  - To delete x, set the bit for x to 0.
  - To lookup x, check whether the bit for x is 1.
- Space usage is $\Theta(U)$.

1101100010111011110101001101010111100110111101111

Bitvectors

- The min, max, predecessor, and successor operations on bitvectors can be extremely slow.
- Runtime will be $\Theta(U)$ in the worst case.

```
000000000000000000000000000000010000000000000000000000000000000
```

Tiered Bitvectors

- Adapting an approach similar to our hybrid RMQs, we can put a summary structure on top of our bitvector.
- Break the universe $\mathscr{U}$ into $\Theta(U / B)$ blocks of size B.
- Create an auxiliary bitvector of size $\Theta(U / B)$ that stores which blocks are nonempty.

```
0010001000000000000000000000000000000100111011100000000000000000
```

Tiered Bitvectors

- Adapting an approach similar to our hybrid RMQs, we can put a summary structure on top of our bitvector.
- Break the universe $\mathscr{U}$ into $\Theta(U / B)$ blocks of size B.
- Create an auxiliary bitvector of size $\Theta(U / B)$ that stores which blocks are nonempty.

```
00100010 00000000 00000000 00000000 00000100 11110111 00000000 00000000
```

Tiered Bitvectors

- Adapting an approach similar to our hybrid RMQs, we can put a summary structure on top of our bitvector.
- Break the universe $\mathscr{U}$ into $\Theta(U / B)$ blocks of size B.
- Create an auxiliary bitvector of size $\Theta(U / B)$ that stores which blocks are nonempty.

00100010	00000000	00000000	00000000	00000100	11110111	00000000	00000000

Tiered Bitvectors

- Adapting an approach similar to our hybrid RMQs, we can put a summary structure on top of our bitvector.
- Break the universe $\mathscr{U}$ into $\Theta(U / B)$ blocks of size B.
- Create an auxiliary bitvector of size $\Theta(U / B)$ that stores which blocks are nonempty.

1	0	0	0	1	1	0	0
00100010	00000000	00000000	00000000	00000100	11110111	00000000	00000000

Tiered Bitvectors

- Using the same techniques we used for RMQ, we can speed up ordered dictionary operations so that they run in time $O(U / B + B)$.
- As before, this is minimized when $B = \Theta(U^{1/2})$.
- ***Claim:*** Ordered dictionary runtimes are now all $O(U^{1/2})$ (and possibly faster).

1	0	0	0	1	1	0	0
00100010	00000000	00000000	00000000	00000100	11110111	00000000	00000000

Tiered Bitvectors

- We can view our tiered bitvector structure in a different light that will help lead to future improvements.
- Instead of thinking of this as two bitvectors (a main and a summary), think of it as $\Theta(U^{1/2})$ smaller main bitvectors and a summary bitvector.

1	0	0	0	1	1	0	0
00100010	00000000	00000000	00000000	00000100	11110111	00000000	00000000

Tiered Bitvectors

- To perform ***lookup***(x) in this structure, check the $\lfloor x / U^{1/2} \rfloor$th bitvector to see if $x \bmod U^{1/2}$ is present.
- In other words, our top-level ***lookup***(x) call turns into a recursive ***lookup***($\lfloor x / U^{1/2} \rfloor$) call in a smaller bitvector.

1	0	0	0	1	1	0	0
00100010	00000000	00000000	00000000	00000100	11110111	00000000	00000000

Tiered Bitvectors

- To perform ***lookup***(x) in this structure, check the $\lfloor x / U^{1/2} \rfloor$th bitvector to see if $x \bmod U^{1/2}$ is present.
- In other words, our top-level ***lookup***(x) call turns into a recursive ***lookup***($\lfloor x / U^{1/2} \rfloor$) call in a smaller bitvector.

1	0	0	0	1	1	0	0
00100010	00000000	00000000	00000000	00000100	11110111	00000000	00000000

$x = 42$

Tiered Bitvectors

- To perform ***lookup***(x) in this structure, check the $\lfloor x \,/\, U^{1/2} \rfloor$th bitvector to see if x mod $U^{1/2}$ is present.
- In other words, our top-level ***lookup***(x) call turns into a recursive ***lookup***$(\lfloor x \,/\, U^{1/2} \rfloor)$ call in a smaller bitvector.

1	0	0	0	1	1	0	0
00100010	00000000	00000000	00000000	00000100	11110111	00000000	00000000

$\lfloor 42 \,/\, 8 \rfloor = 5$ $\quad$ $42 \bmod 8 = 2$

Tiered Bitvectors

- To perform ***lookup***(x) in this structure, check the $\lfloor x / U^{1/2} \rfloor$th bitvector to see if $x \bmod U^{1/2}$ is present.
- In other words, our top-level ***lookup***(x) call turns into a recursive ***lookup***($\lfloor x / U^{1/2} \rfloor$) call in a smaller bitvector.

1	0	0	0	1	1	0	0
00100010	00000000	00000000	00000000	00000100	11110111	00000000	00000000

$\lfloor 42 / 8 \rfloor = 5$ $42 \bmod 8 = 2$

Tiered Bitvectors

- To perform ***insert***(x) in this structure, insert x mod $U^{1/2}$ into the $\lfloor x / U^{1/2} \rfloor$th bitvector, then insert $\lfloor x / U^{1/2} \rfloor$ into the summary bitvector.
- Turns one ***insert*** call into two recursive ***insert*** calls.

1	0	0	0	1	1	0	0
00100010	00000000	00000000	00000000	00000100	11110111	00000000	00000000

Tiered Bitvectors

- To perform ***insert***(x) in this structure, insert x mod $U^{1/2}$ into the $\lfloor x / U^{1/2} \rfloor$th bitvector, then insert $\lfloor x / U^{1/2} \rfloor$ into the summary bitvector.
- Turns one ***insert*** call into two recursive ***insert*** calls.

1	0	0	0	1	1	0	0
00100010	00000000	00000000	00000000	00000100	11110111	00000000	00000000

$x = 25$

Tiered Bitvectors

- To perform ***insert***(x) in this structure, insert x mod $U^{1/2}$ into the $\lfloor x / U^{1/2} \rfloor$th bitvector, then insert $\lfloor x / U^{1/2} \rfloor$ into the summary bitvector.
- Turns one ***insert*** call into two recursive ***insert*** calls.

1	0	0	0	1	1	0	0
00100010	00000000	00000000	00000000	00000100	11110111	00000000	00000000

$\lfloor 25 / 8 \rfloor = 3$ $25 \bmod 8 = 1$

Tiered Bitvectors

- To perform ***insert***(x) in this structure, insert $x \bmod U^{1/2}$ into the $\lfloor x / U^{1/2} \rfloor$th bitvector, then insert $\lfloor x / U^{1/2} \rfloor$ into the summary bitvector.
- Turns one ***insert*** call into two recursive ***insert*** calls.

1	0	0	0	1	1	0	0
00100010	00000000	00000000	00000000	00000100	11110111	00000000	00000000

$\lfloor 25 / 8 \rfloor = 3 \qquad 25 \bmod 8 = 1$

Tiered Bitvectors

- To perform ***insert***(x) in this structure, insert x mod $U^{1/2}$ into the $\lfloor x \;/\; U^{1/2} \rfloor$th bitvector, then insert $\lfloor x \;/\; U^{1/2} \rfloor$ into the summary bitvector.
- Turns one ***insert*** call into two recursive ***insert*** calls.

1	0	0	1	1	1	0	0
00100010	00000000	00000000	01000000	00000100	11110111	00000000	00000000

$\lfloor 25 \;/\; 8 \rfloor = 3 \qquad 25 \text{ mod } 8 = 1$

Tiered Bitvectors

- To perform ***max***(), call ***max*** on the summary structure.
- If it returns value *v*, return ***max*** of the *v*th bitvector.
- Turns one ***max*** call into two recursive ***max***s.

1	0	0	0	1	1	0	0
00100010	00000000	00000000	00000000	00000100	11110111	00000000	00000000

Tiered Bitvectors

- To perform ***max***(), call ***max*** on the summary structure.
- If it returns value *v*, return ***max*** of the *v*th bitvector.
- Turns one ***max*** call into two recursive ***max***s.

1	0	0	0	1	1	0	0
00100010	00000000	00000000	00000000	00000100	11110111	00000000	00000000

Tiered Bitvectors

- To perform ***max***(), call ***max*** on the summary structure.
- If it returns value *v*, return ***max*** of the *v*th bitvector.
- Turns one ***max*** call into two recursive ***max***s.

1	0	0	0	1	1	0	0
00100010	00000000	00000000	00000000	00000100	11110111	00000000	00000000

Tiered Bitvectors

- To perform ***successor***(x), do the following:
  - Find ***max*** in the $\lfloor x \mathbin{/} U^{1/2} \rfloor$th bitvector.
  - If it exists and is greater than x, find ***successor***(x mod $U^{1/2}$) in that bitvector.
  - Otherwise, find ***successor***($\lfloor x \mathbin{/} U^{1/2} \rfloor$) in the summary structure; let it be j if it exists.
  - Return ***min*** of the jth bitvector of it exists or ∞ otherwise.
- Turns ***successor*** into a ***max***, a ***min***, and a ***successor***.

1	0	0	0	1	1	0	0
00100010	00000000	00000000	00000000	00000100	11110111	00000000	00000000

Tiered Bitvectors

- To perform an ***is-empty*** query, return the result of that query on the summary structure.
- Turns one ***is-empty*** query into a single smaller ***is-empty*** query.

1	0	0	0	1	1	0	0
00100010	00000000	00000000	00000000	00000100	11110111	00000000	00000000

Tiered Bitvectors

- To perform ***delete***(x) in this structure, delete x mod $U^{1/2}$ from the $\lfloor x / U^{1/2} \rfloor$th bitvector.
- Then, check ***is-empty*** on that bitvector, and if so, ***delete***($\lfloor x / U^{1/2} \rfloor$) from the summary bitvector.
- Turns one ***delete*** call into up to two recursive ***delete***s and one ***is-empty***.

1	0	0	0	1	1	0	0
00100010	00000000	00000000	00000000	00000100	11110111	00000000	00000000

Tiered Bitvectors

- To perform ***delete***(x) in this structure, delete x mod $U^{1/2}$ from the $\lfloor x / U^{1/2} \rfloor$th bitvector.
- Then, check ***is-empty*** on that bitvector, and if so, ***delete***($\lfloor x / U^{1/2} \rfloor$) from the summary bitvector.
- Turns one ***delete*** call into up to two recursive ***delete***s and one ***is-empty***.

1	0	0	0	1	1	0	0
00100010	00000000	00000000	00000000	00000**1**00	11110111	00000000	00000000

Tiered Bitvectors

- To perform ***delete***(x) in this structure, delete x mod $U^{1/2}$ from the $\lfloor x / U^{1/2} \rfloor$th bitvector.
- Then, check ***is-empty*** on that bitvector, and if so, ***delete***($\lfloor x / U^{1/2} \rfloor$) from the summary bitvector.
- Turns one ***delete*** call into up to two recursive ***delete***s and one ***is-empty***.

1	0	0	0	0	1	0	0
00100010	00000000	00000000	00000000	00000000	11110111	00000000	00000000

The Story So Far

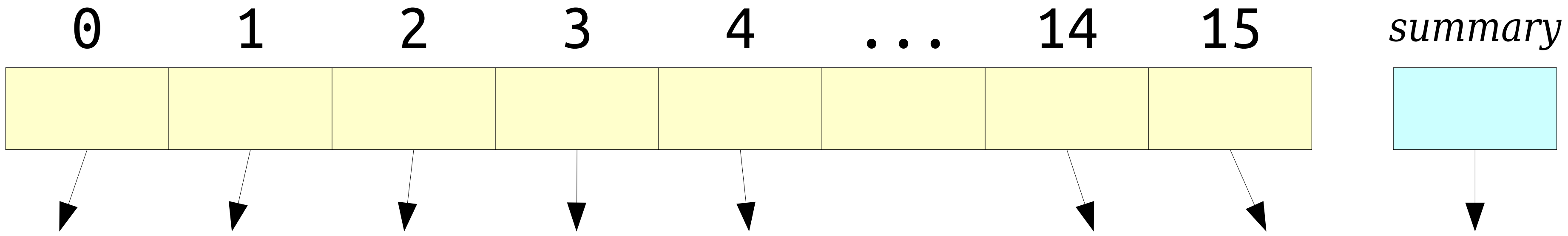
- Each operation turns into recursive operations on a smaller bitvector:
  - ***insert***: 2x ***insert***
  - ***lookup***: 1x ***lookup***
  - ***is-empty***: 1x ***is-empty***
  - ***min***: 2x ***min***
  - ***successor***: 1x ***successor***, 1x ***max***, 1x ***min***
  - ***delete***: 2x ***delete***, 1x ***is-empty***

A Recursive Approach

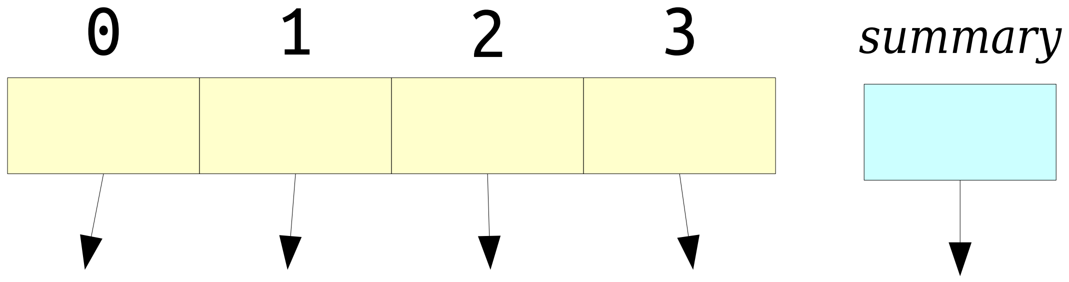
- Adding one tier to the bitvector sped things up appreciably.
- ***Idea:*** What if we apply this same approach to each of the smaller bitvectors?
- Builds a recursive data structure:
  - If $U \leq 2$, just use a normal bitvector.
  - Otherwise, to build a data structure for a universe of size U, split the input apart into $\Theta(U^{1/2})$ blocks of size $\Theta(U^{1/2})$ and add a summary data structure on top.
  - Answer queries using the same approach we outlined earlier.

Our Data Structure

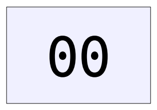
- Let $\mathscr{U} = [256]$.
- The top-level structure looks like this:



- Each structure one level below (and the summary) looks like this:



- Each structure one level below that looks like this:



So… how efficient is it?

Analyzing the Operations

- Let's analyze the ***is-empty*** and ***lookup*** operations in this structure.
- Each makes a recursive call to a problem of size $\Theta(U^{1/2})$ and does O(1) work.
- Recurrence relation:

$$\begin{aligned} T(2) &= \Theta(1) \\ T(U) &\le T(U^{1/2}) + \Theta(1) \end{aligned}$$

- How do we solve this recurrence?

A Useful Substitution

- The Master Theorem is great for working with recurrences of the form

$$T(n) \leq aT(n / b) + \mathrm{O}(n^d)$$

- This recurrence doesn't have this form because the “shrinking” step is a square root rather than a division.
- To address this, we'll transform the recurrence so that it fits into the above form.
- If we write $U = 2^k$, then $U^{1/2} = 2^{k/2}$.
- Turn the recurrence from a recurrence in U to a recurrence in $k = \log U$.

The Substitution

- Define $S(k) = T(2^k)$.
- Since

$$T(2) \leq \Theta(1)$$
$$T(U) \leq T(U^{1/2}) + \Theta(1)$$

- We have

$$S(1) \leq \Theta(1)$$
$$S(k) \leq S(k / 2) + \Theta(1)$$

- This means that $S(k) = \mathrm{O}(\log k)$.
- So $T(U) = T(2^{\lg U}) = S(\lg U) =$ **$\mathbf{O}(\log \log U)$**.

Analyzing the Operations

- The ***insert*** and ***min*** operations each make two recursive calls on subproblems of size $\Theta(U^{1/2})$ and do $\Theta(1)$ work.
- Gives this recurrence:

$$T(2) \leq \Theta(1)$$
$$T(U) \leq 2T(U^{1/2}) + \Theta(1)$$

- Substituting $S(k) = T(2^k)$ yields

$$S(1) \leq \Theta(1)$$
$$S(k) \leq 2S(k / 2) + \Theta(1)$$

- So $S(k) = O(k)$.
- Therefore, $T(U) = S(2^{\lg U}) =$ **$O(\log U)$**.

Analyzing the Operations

- Each ***delete*** call makes two recursive ***delete*** calls and one call to ***is-empty***.
- As we saw, ***is-empty*** takes time O(log log U)
- Recurrence relation is

$$T(2) \leq \Theta(1)$$
$$T(U) \leq 2T(U^{1/2}) + O(\log \log U)$$

- Letting $S(k) = T(2^k)$ gives

$$S(1) \leq \Theta(1)$$
$$S(k) \leq 2S(k / 2) + O(\log k)$$

- Via the Master Theorem, $S(k) = O(k)$.
- Thus $T(U) =$ **O(log U)**.

Analyzing the Operations

- Each ***successor*** call makes one recursive ***successor*** call and one call to ***max*** and ***min***.
- As we saw, ***max*** and ***min*** takes time $O(\log U)$
- Recurrence relation is

$$T(2) \le \Theta(1)$$
$$T(U) \le T(U^{1/2}) + O(\log U)$$

- Letting $S(k) = T(2^k)$ gives

$$S(1) \le \Theta(1)$$
$$S(k) \le T(k / 2) + O(k)$$

- Via the Master Theorem, $S(k) = O(k)$.
- Thus $T(U) =$ **$O(\log U)$**.

Where We Stand

- Right now, we have a data structure where lookups are exponentially faster than a balanced BST if $n = \Omega(\log U)$.
- Other operations have runtime proportional to $\log U$, which is (usually) greater than $\log n$.
- ***Can we speed things up?***

Time-Out for Announcements!

Problem Set Five

- As a reminder, Problem Set Five is due this Thursday at the start of class.
- Have questions? As always, stop by our office hours or ask questions on Piazza.

Midterm Logistics

- The CS166 midterm exam is next **Tuesday, May 24** from **7PM – 10PM** in **320-105**.
- Exam is cumulative and covers everything we've talked about this quarter, with a focus on the topics from PS1 – PS5. Topics from this week and next Tuesday are fair game but, understandably, won't be tested in nearly as much depth.
- Exam is closed-book, closed-computer, and limited-note: you can bring a double-sided, 8.5” × 11” sheet of notes with you when you take the exam.
- We've released a set of practice problems to help you prepare for the exam. They're up on the course website and we'll distribute solutions on Thursday.
- Can't make the exam time? Let us know ASAP so that we can set up an alternate time.

WISE Inspirations Network at Stanford (WINS)

featuring **Maria Klawe:**

Twists and Turns as an Academic Leader

Join WINS as we feature **Maria Klawe** and her discussion of the adventures, mishaps and insights of a female scientist and engineer who wanted to make the culture of science and engineering welcoming to everyone with passion and ability, from poets and football players to women and under-represented minorities.

• Wednesday, May 25, 2016 •
• 4:30 PM - 6:30 PM •
• Mackenzie Room, Huang 300 •
RSVP HERE by May 23, 2016
Program will begin at 4:30
Networking Reception 6:00 – 6:30 PM

MARIA KLAWE became Harvey Mudd College's fifth president in 2006. A renowned mathematician, computer scientist and scholar, Klawe is the first woman to lead the College since its founding in 1955. Prior to joining Harvey Mudd, she served as dean of engineering and professor of computer science at Princeton University. Klawe has made significant research contributions to and held numerous leadership roles in the areas of mathematics and computer science. Maria received her PhD (1977) and BSc (1973) in mathematics from the University of Alberta.

Be Inspired • Network • Connect

wise.stanford.edu
vpge.stanford.edu/wise

Register to Vote!

- I have a giant stack of voter registration forms up at the front of class today.
- If you're not registered to vote and would like to register, feel free to pick one up.

Back to CS166!

Identifying Inefficiencies

- Fundamentally, the recurrences we need to solve to determine the costs of these operations either have the form

$$T(U) = T(U^{1/2}) + f(U)$$

  or

$$T(U) = 2T(U^{1/2}) + f(U).$$

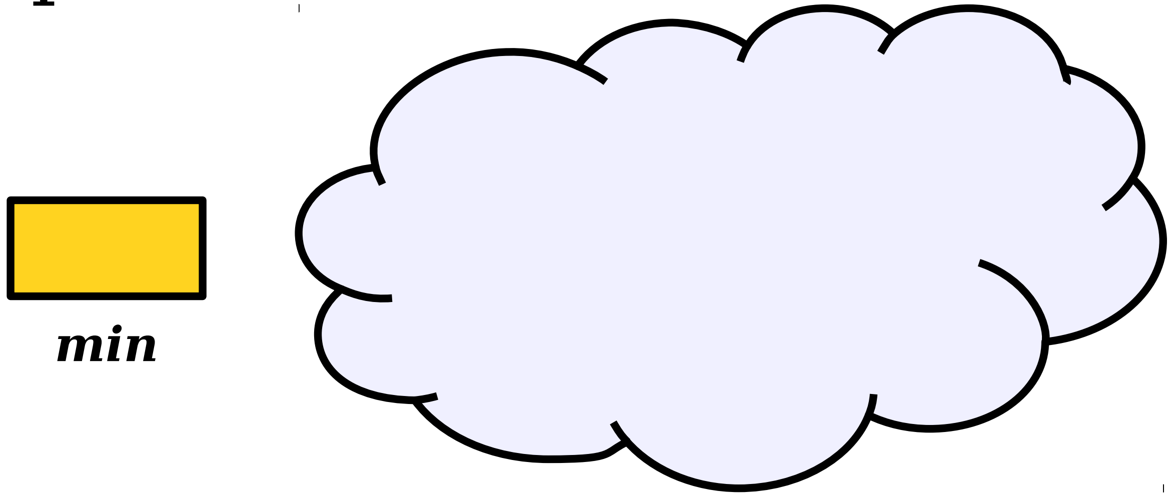
- This first recurrence (often) solves to $O(\log \log U)$, which is $o(\log n)$ as long as $n = \omega(\log U)$.
- The second recurrence often solves to $O(\log U)$, which is never $o(\log n)$.
- ***Theoryland Goal:*** Find a way to convert recurrences of the second type into recurrences of the first type.

Identifying Inefficiencies

- A few operations seem like easy candidates for speedups:
  - ***is-empty*** certainly seems like it shouldn't take time O(log log *U*).
  - ***max*** and ***min*** can probably don't actually need time O(log *U*).
- We'll show how to speed up these three operations.
- By doing so, we'll significantly improve the runtimes of the other operations.

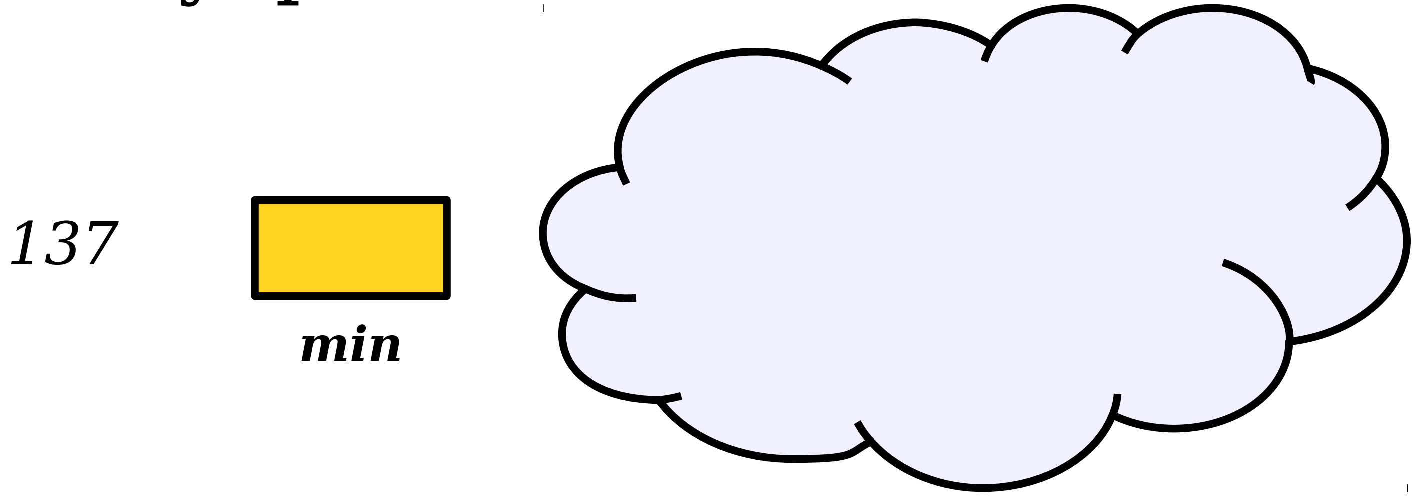
Improving Min and Max

- Suppose you have a priority queue where finding the min takes time ω(1).
- How could you modify it so that finding the min can be done in time O(1)?
- ***Answer:*** Store the minimum outside of the priority queue.



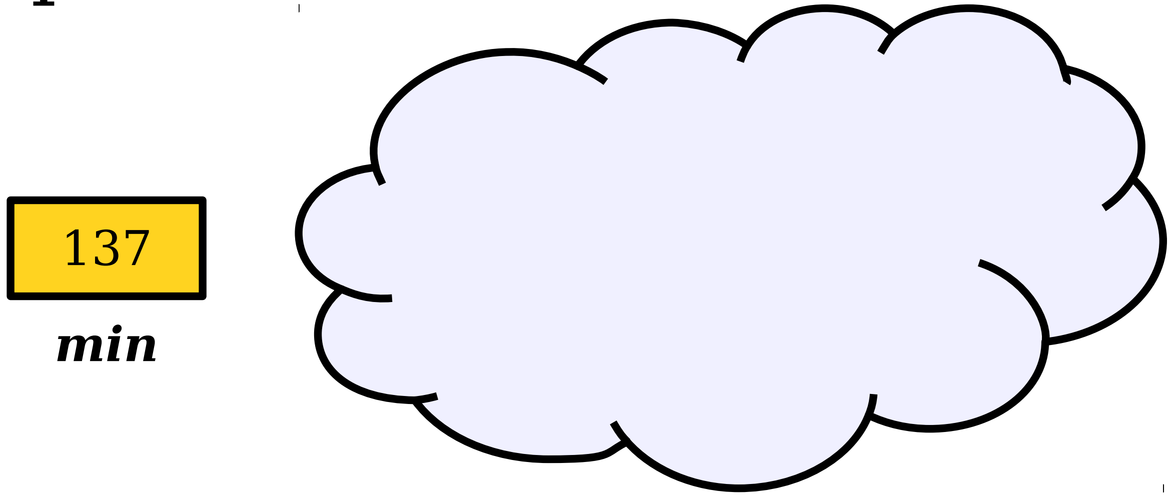
Improving Min and Max

- Suppose you have a priority queue where finding the min takes time ω(1).
- How could you modify it so that finding the min can be done in time O(1)?
- ***Answer:*** Store the minimum outside of the priority queue.



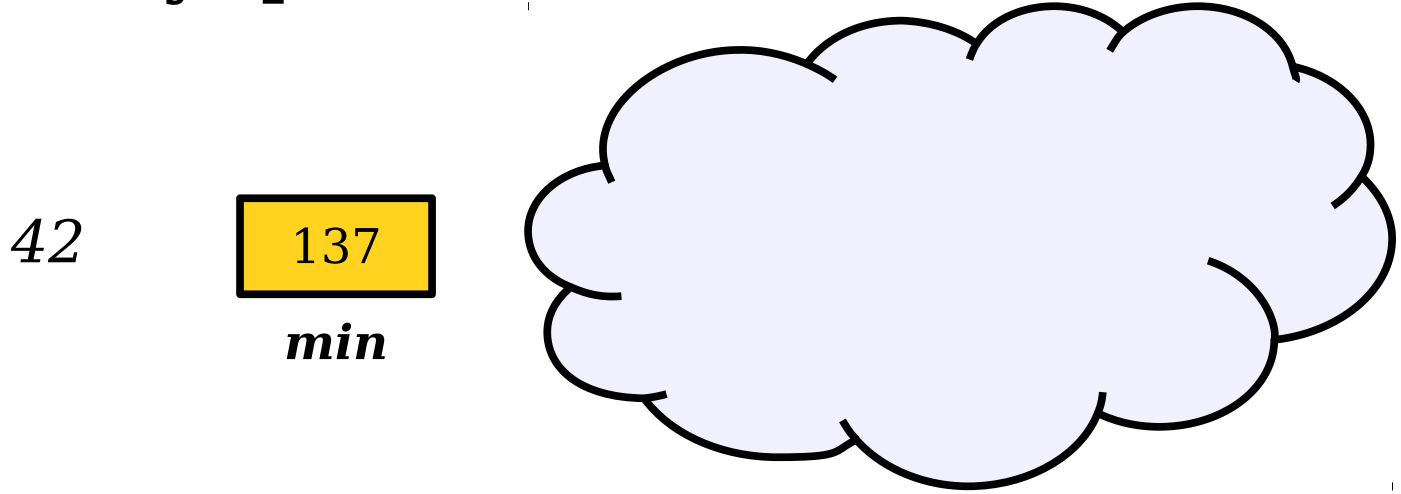
Improving Min and Max

- Suppose you have a priority queue where finding the min takes time ω(1).
- How could you modify it so that finding the min can be done in time O(1)?
- ***Answer:*** Store the minimum outside of the priority queue.



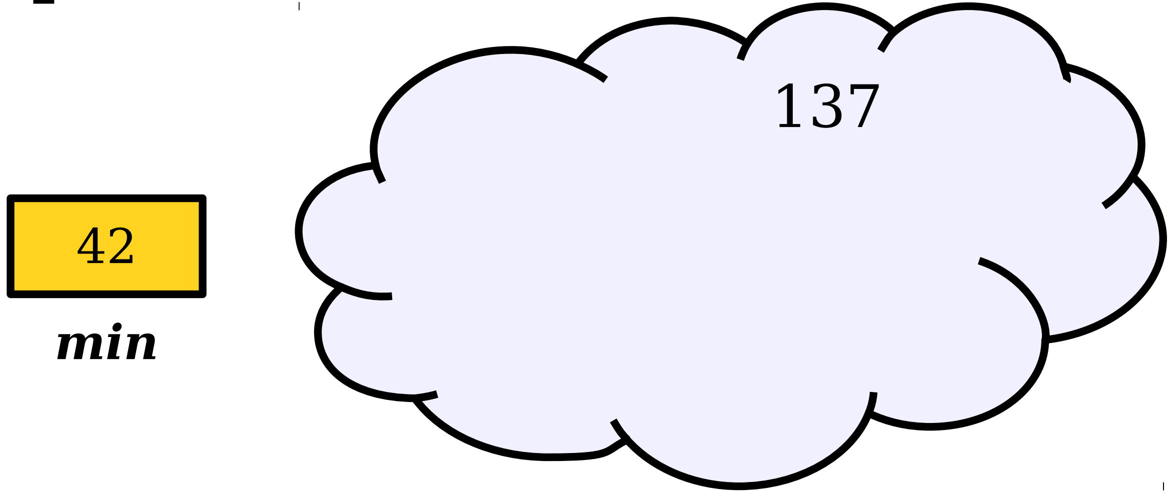
Improving Min and Max

- Suppose you have a priority queue where finding the min takes time ω(1).
- How could you modify it so that finding the min can be done in time O(1)?
- ***Answer:*** Store the minimum outside of the priority queue.



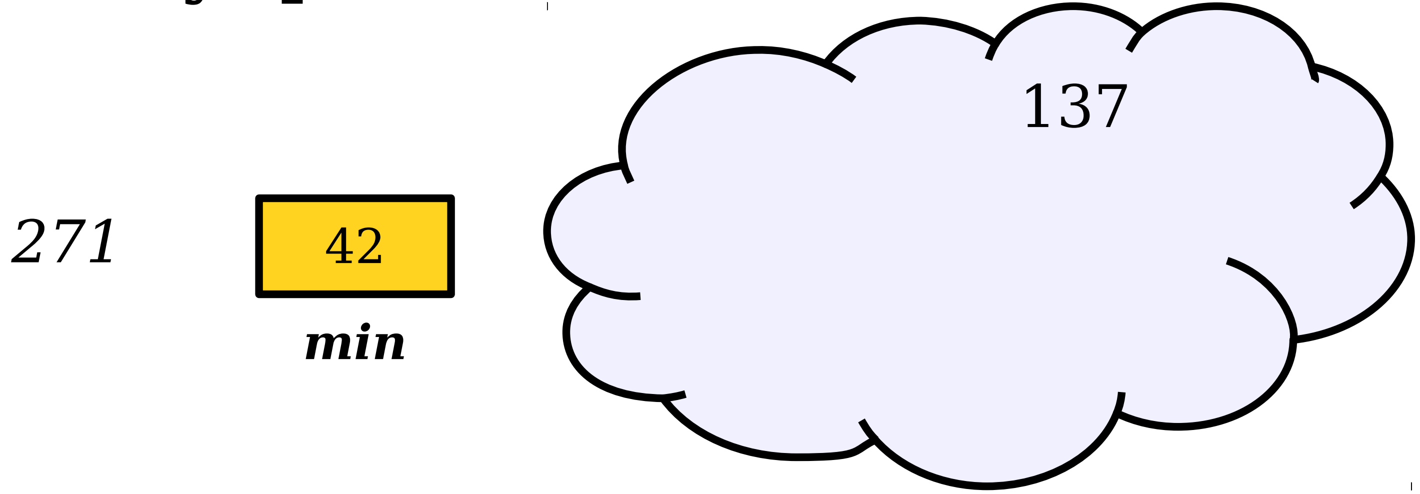
Improving Min and Max

- Suppose you have a priority queue where finding the min takes time ω(1).
- How could you modify it so that finding the min can be done in time O(1)?
- ***Answer:*** Store the minimum outside of the priority queue.



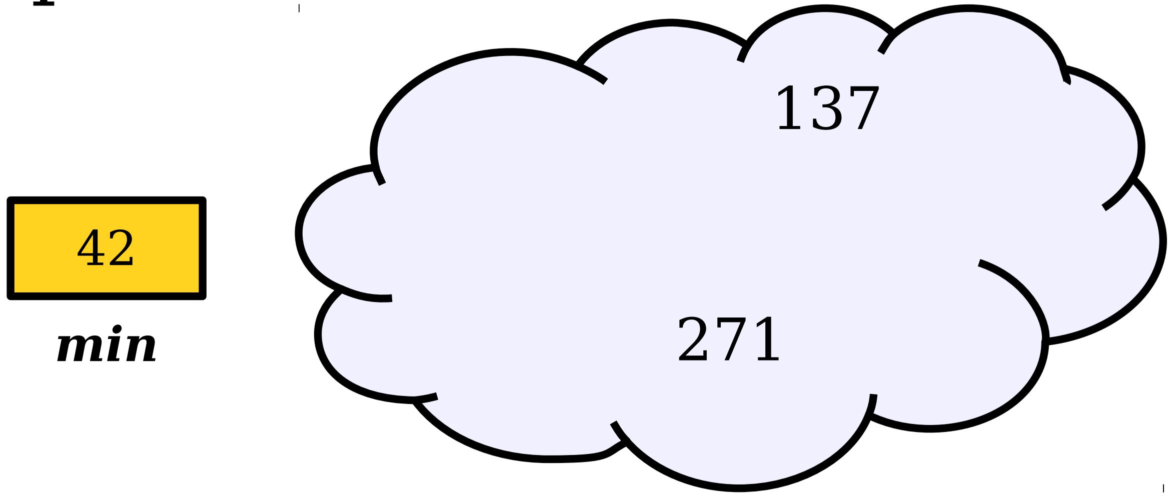
Improving Min and Max

- Suppose you have a priority queue where finding the min takes time ω(1).
- How could you modify it so that finding the min can be done in time O(1)?
- ***Answer:*** Store the minimum outside of the priority queue.



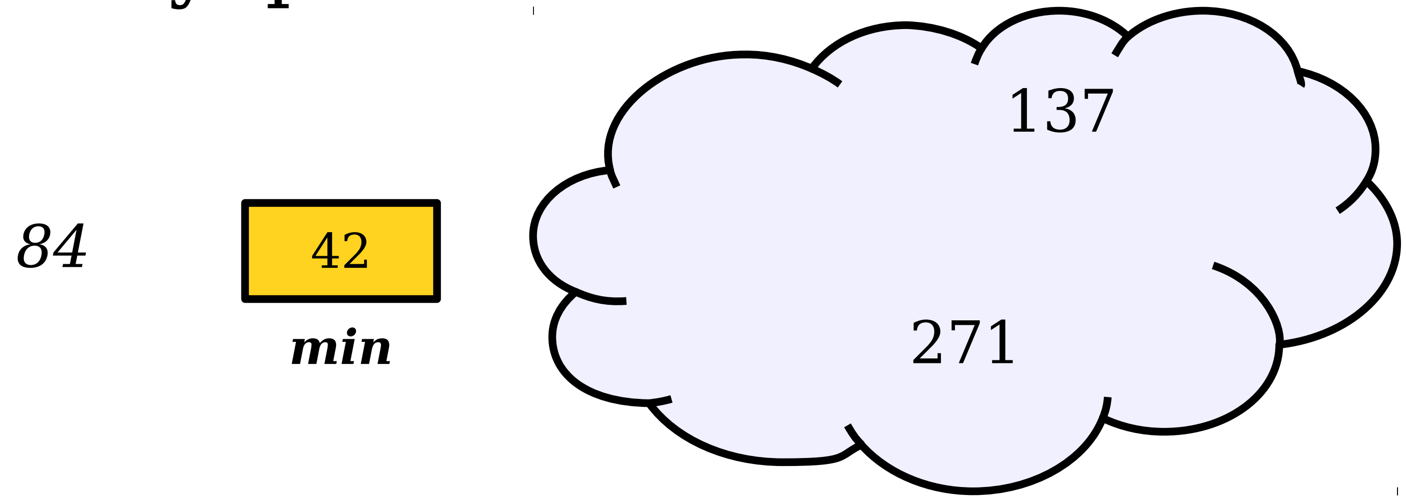
Improving Min and Max

- Suppose you have a priority queue where finding the min takes time ω(1).
- How could you modify it so that finding the min can be done in time O(1)?
- ***Answer:*** Store the minimum outside of the priority queue.



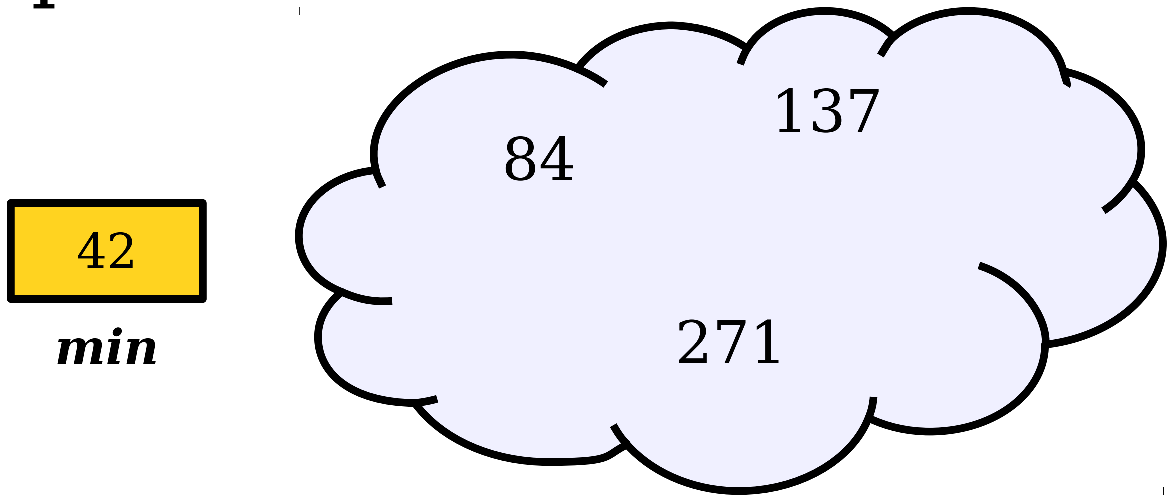
Improving Min and Max

- Suppose you have a priority queue where finding the min takes time ω(1).
- How could you modify it so that finding the min can be done in time O(1)?
- ***Answer:*** Store the minimum outside of the priority queue.



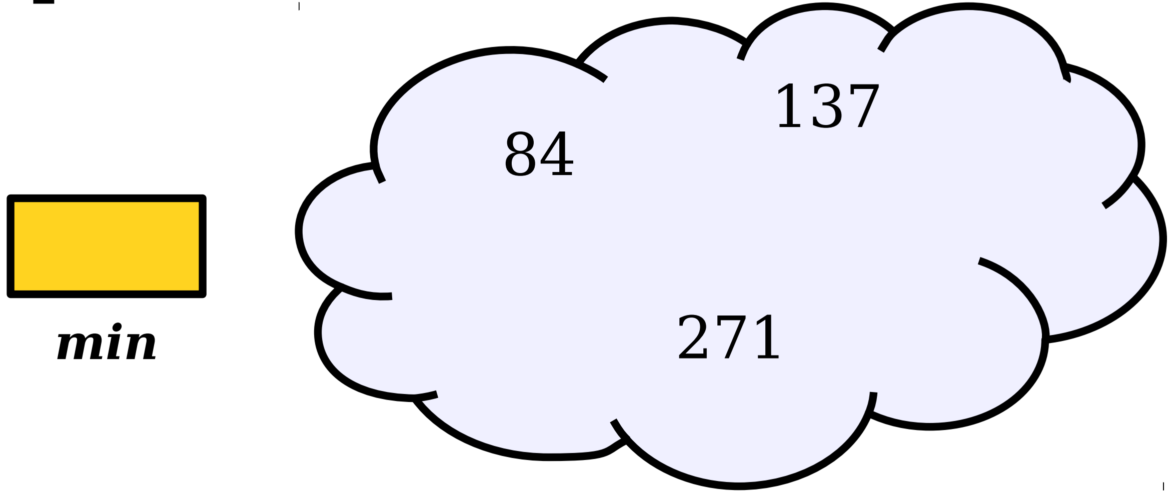
Improving Min and Max

- Suppose you have a priority queue where finding the min takes time ω(1).
- How could you modify it so that finding the min can be done in time O(1)?
- ***Answer:*** Store the minimum outside of the priority queue.



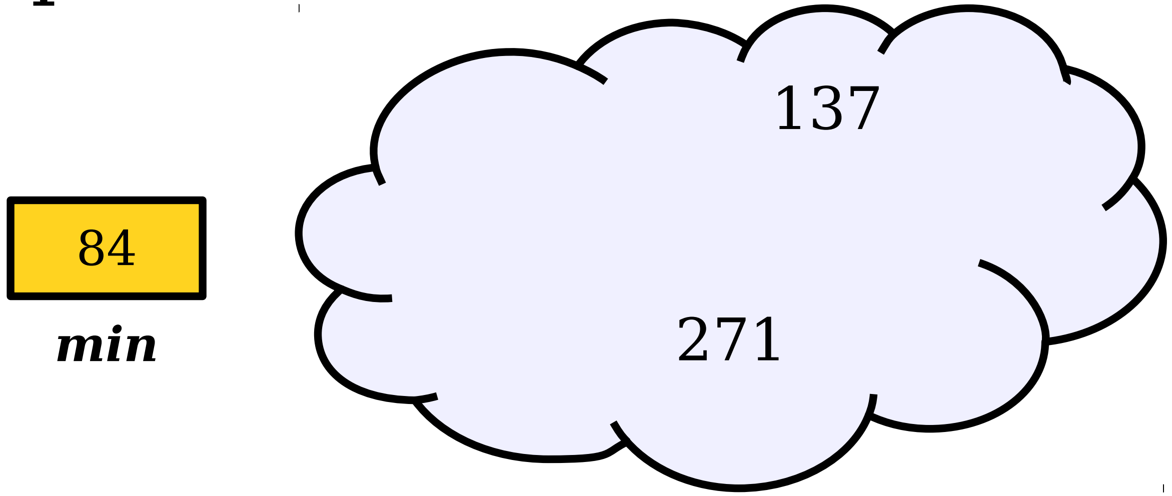
Improving Min and Max

- Suppose you have a priority queue where finding the min takes time ω(1).
- How could you modify it so that finding the min can be done in time O(1)?
- ***Answer:*** Store the minimum outside of the priority queue.



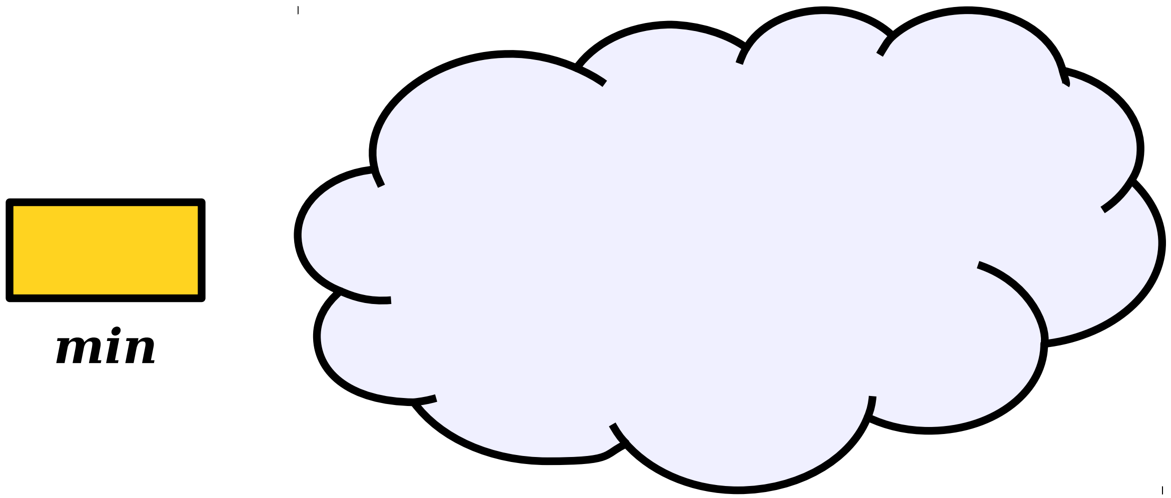
Improving Min and Max

- Suppose you have a priority queue where finding the min takes time ω(1).
- How could you modify it so that finding the min can be done in time O(1)?
- ***Answer:*** Store the minimum outside of the priority queue.



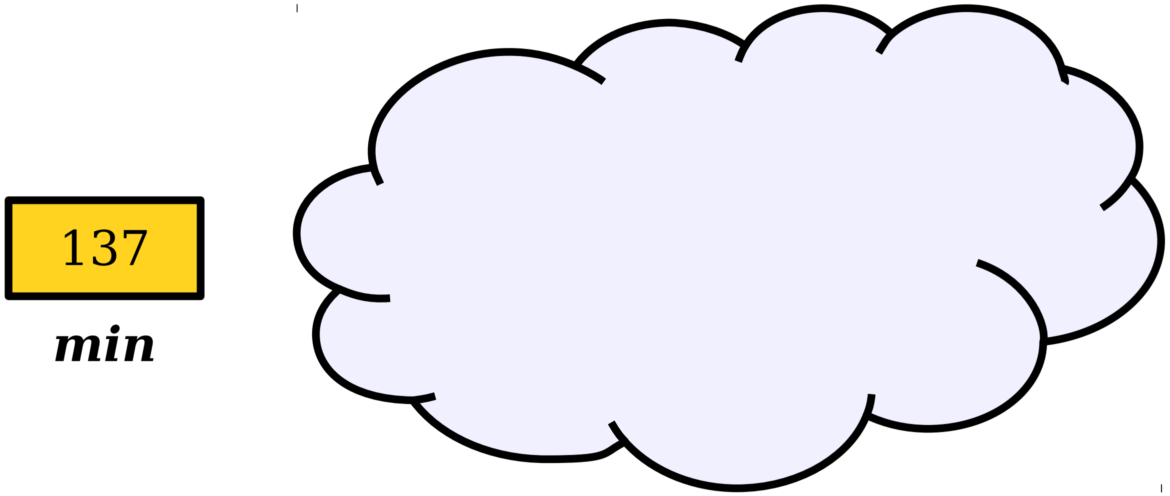
Improving Min and Max

- ***Observation:*** In this setup, the cost of inserting into an empty priority queue is O(1).
- We just store the value in the *min* field without having to do any priority queue operations.



Improving Min and Max

- ***Observation:*** In this setup, the cost of inserting into an empty priority queue is O(1).
- We just store the value in the *min* field without having to do any priority queue operations.

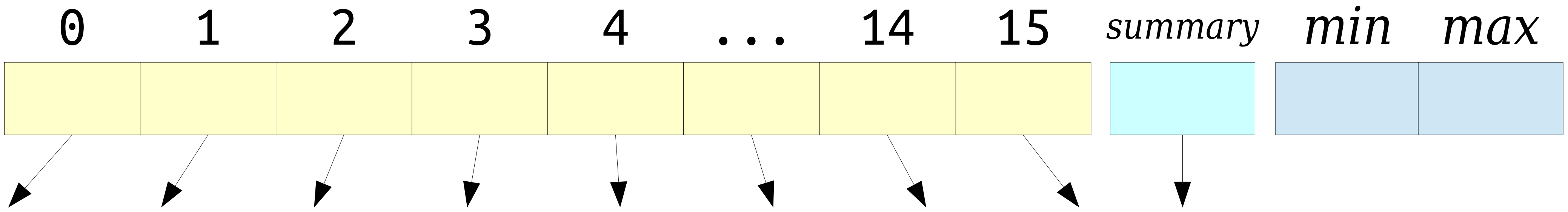


van Emde Boas Trees

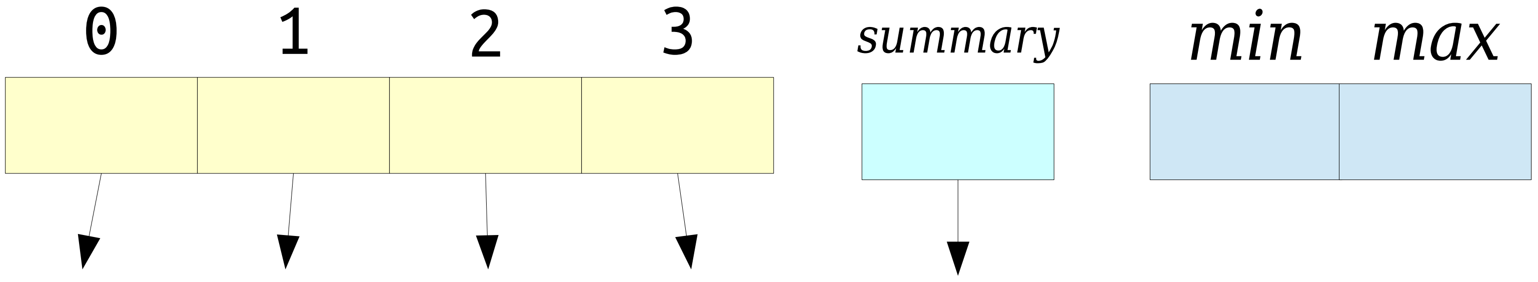
- A ***van Emde Boas tree*** is a slight modification to our previous structure.
- As before, each level recursively splits the universe into $\Theta(U^{1/2})$ blocks of size $\Theta(U^{1/2})$.
- As before, each level stores pointers to children for each subuniverse and stores a pointer to a summary structure of size $\Theta(U^{1/2})$.
- Each recursive copy of data structure stores its minimum and maximum values separately from the rest of the structure.
    - ***This is not the same as caching the min and max***. The minimum and maximum values at each level of the recursion are stored separately and not recursively added into the rest of the structure. ***This is important for the later analysis!***

van Emde Boas Trees

- Let $\mathscr{U} = [256]$.
- The top-level structure looks like this:

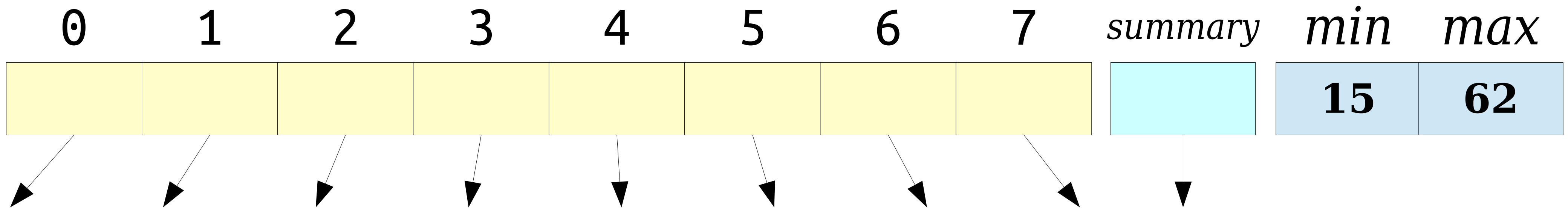


- Each structure one level below (and the summary) looks like this:



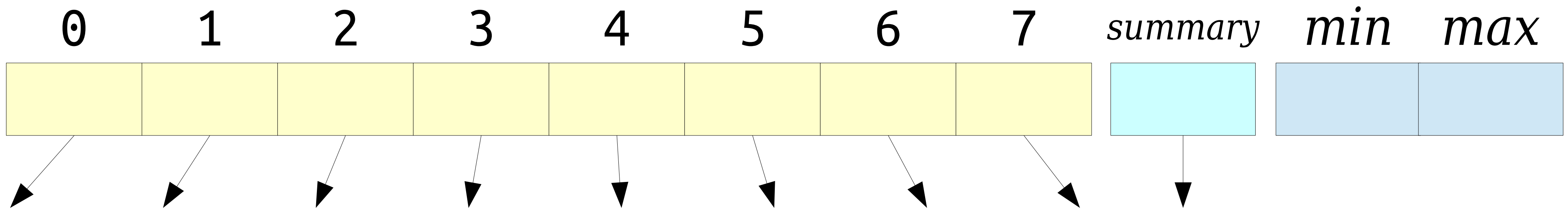
vEB Tree Lookups

- Lookups in a vEB tree work as before, but with one extra step: check whether the value being searched for is the *min* or *max* value.



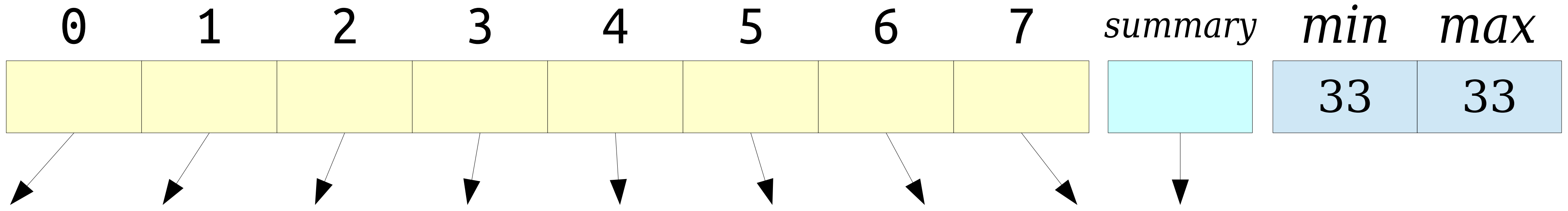
vEB Tree Insertions

- Insertions in a vEB tree work as before, but with extra logic to handle *min* and *max*.
  - May need to handle the case where the tree is empty.



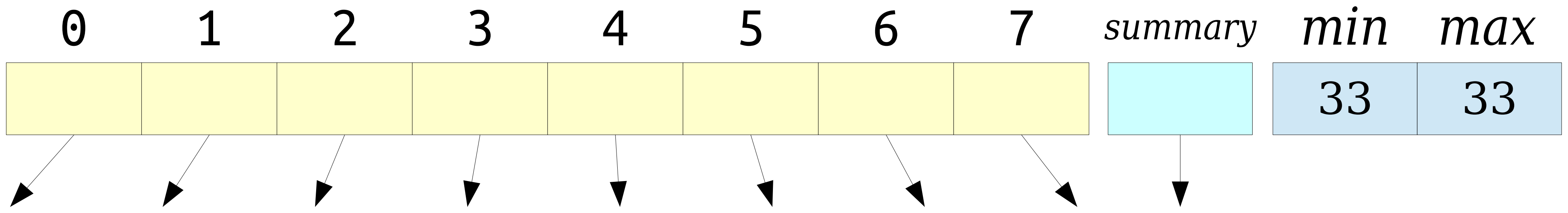
vEB Tree Insertions

- Insertions in a vEB tree work as before, but with extra logic to handle *min* and *max*.
  - May need to handle the case where the tree is empty.



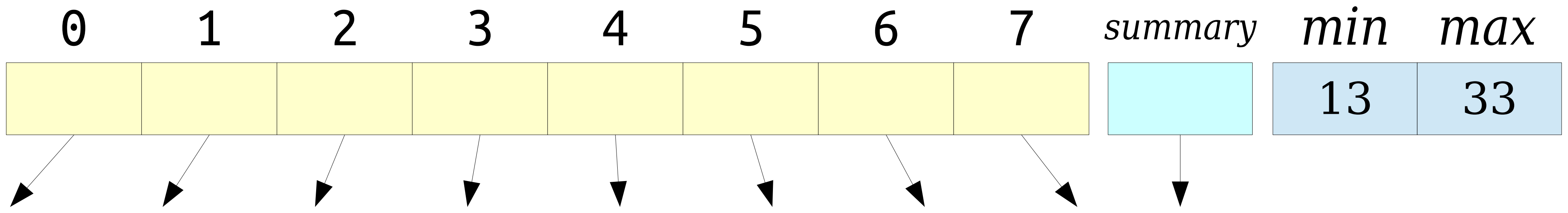
vEB Tree Insertions

- Insertions in a vEB tree work as before, but with extra logic to handle *min* and *max*.
  - May need to handle the case where the tree is empty.
  - May need to handle the case where the tree has just one element.



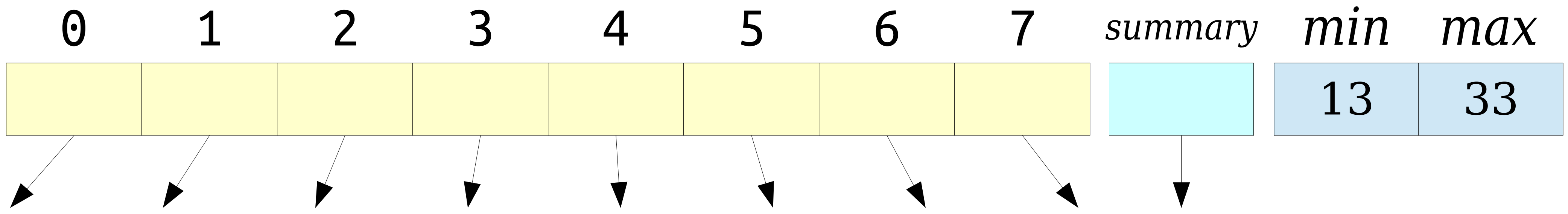
vEB Tree Insertions

- Insertions in a vEB tree work as before, but with extra logic to handle *min* and *max*.
    - May need to handle the case where the tree is empty.
    - May need to handle the case where the tree has just one element.



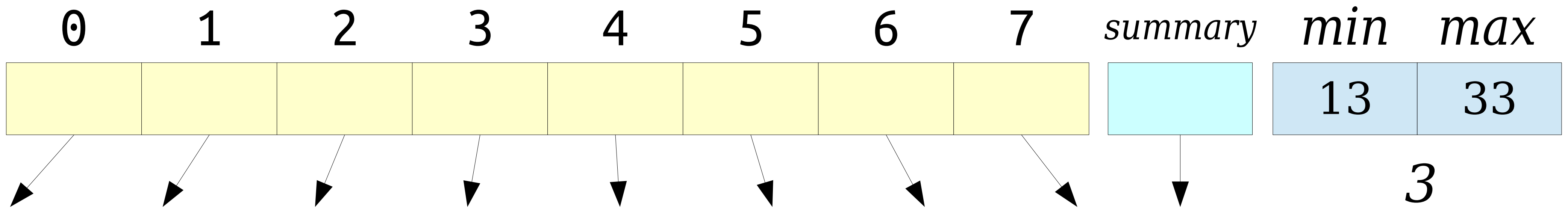
vEB Tree Insertions

- Insertions in a vEB tree work as before, but with extra logic to handle *min* and *max*.
    - May need to handle the case where the tree is empty.
    - May need to handle the case where the tree has just one element.
    - May need to displace *min* or *max* into the tree.



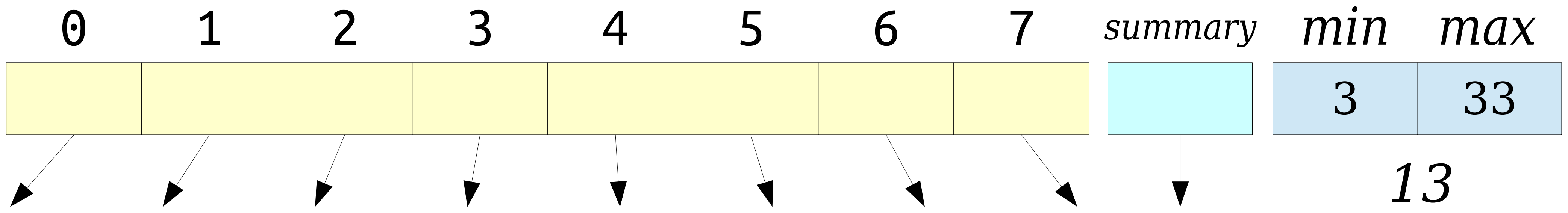
vEB Tree Insertions

- Insertions in a vEB tree work as before, but with extra logic to handle *min* and *max*.
    - May need to handle the case where the tree is empty.
    - May need to handle the case where the tree has just one element.
    - May need to displace *min* or *max* into the tree.



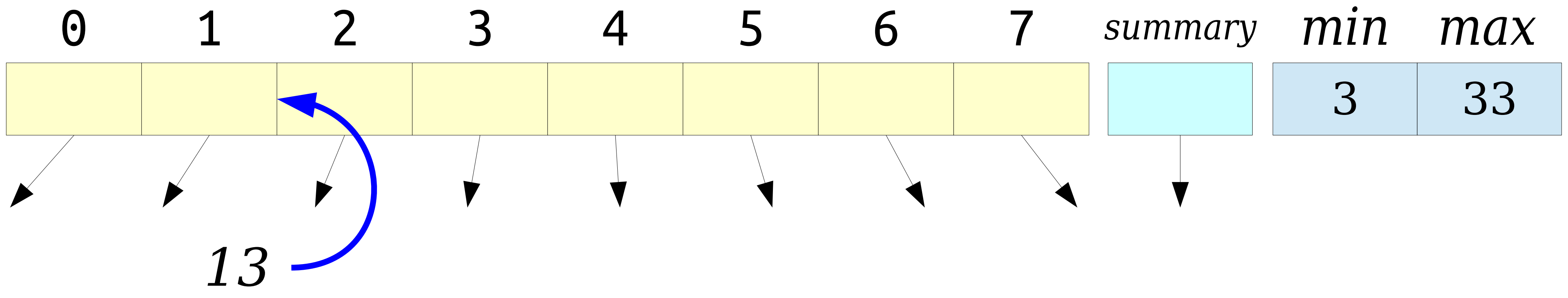
vEB Tree Insertions

- Insertions in a vEB tree work as before, but with extra logic to handle *min* and *max*.
  - May need to handle the case where the tree is empty.
  - May need to handle the case where the tree has just one element.
  - May need to displace *min* or *max* into the tree.



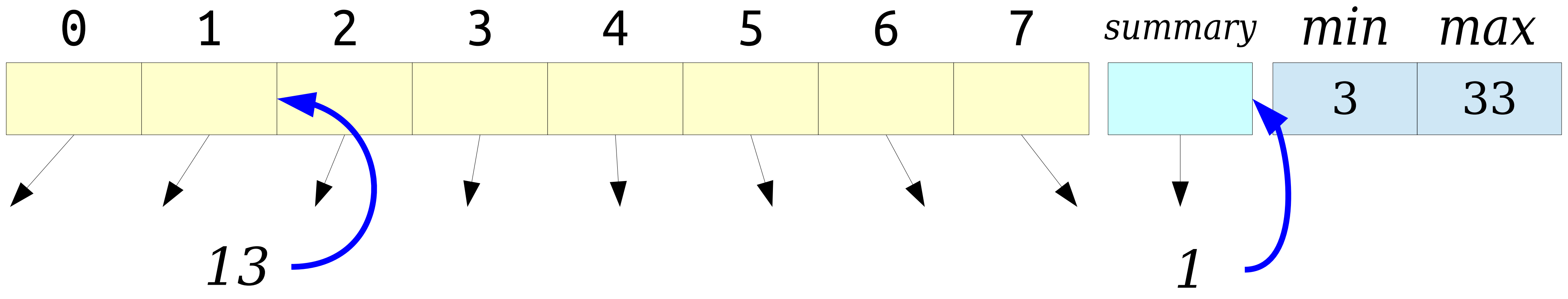
vEB Tree Insertions

- Insertions in a vEB tree work as before, but with extra logic to handle *min* and *max*.
  - May need to handle the case where the tree is empty.
  - May need to handle the case where the tree has just one element.
  - May need to displace *min* or *max* into the tree.



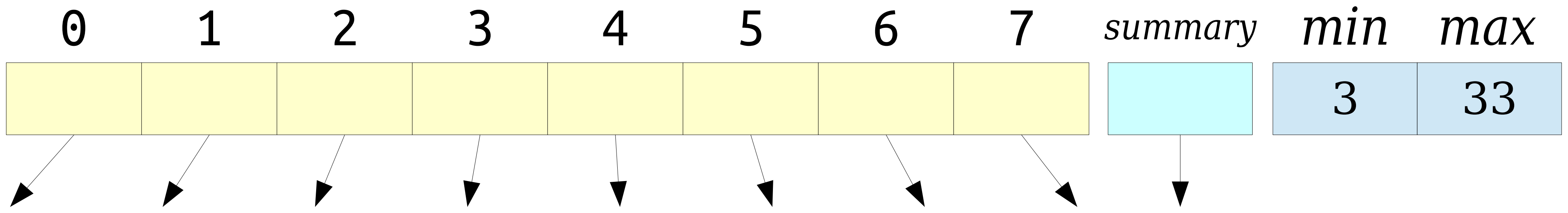
vEB Tree Insertions

- Insertions in a vEB tree work as before, but with extra logic to handle *min* and *max*.
  - May need to handle the case where the tree is empty.
  - May need to handle the case where the tree has just one element.
  - May need to displace *min* or *max* into the tree.



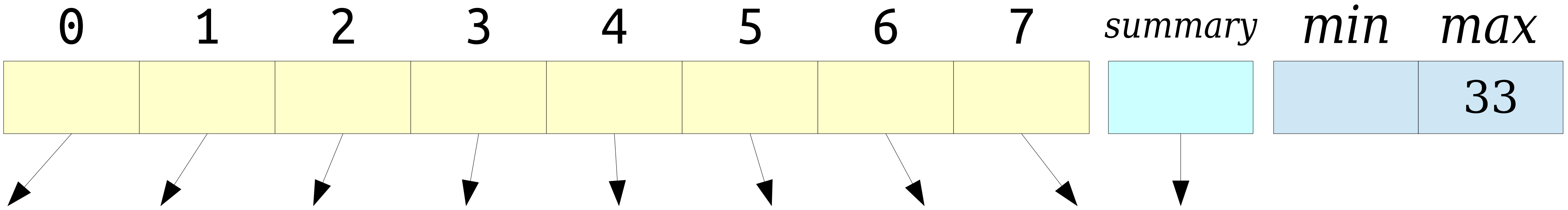
vEB Tree Deletions

- Deletions in a vEB tree work as before, but with extra logic to handle *min* and *max*.
    - May need to pull an element to fill in a missing *min* or *max*.



vEB Tree Deletions

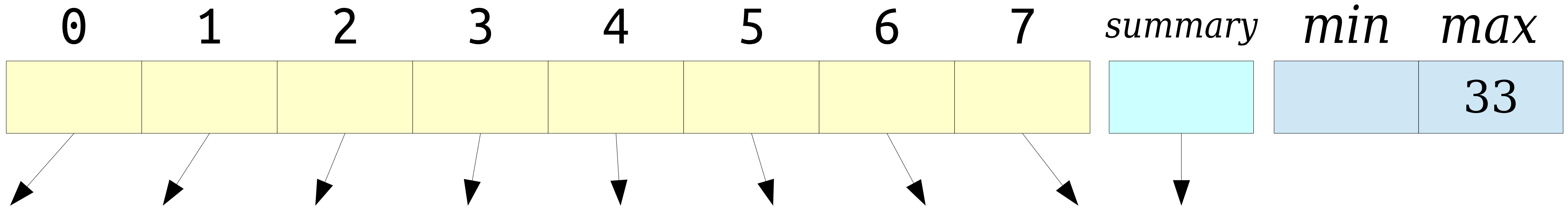
- Deletions in a vEB tree work as before, but with extra logic to handle *min* and *max*.
  - May need to pull an element to fill in a missing *min* or *max*.



vEB Tree Deletions

- Deletions in a vEB tree work as before, but with extra logic to handle *min* and *max*.
    - May need to pull an element to fill in a missing *min* or *max*.

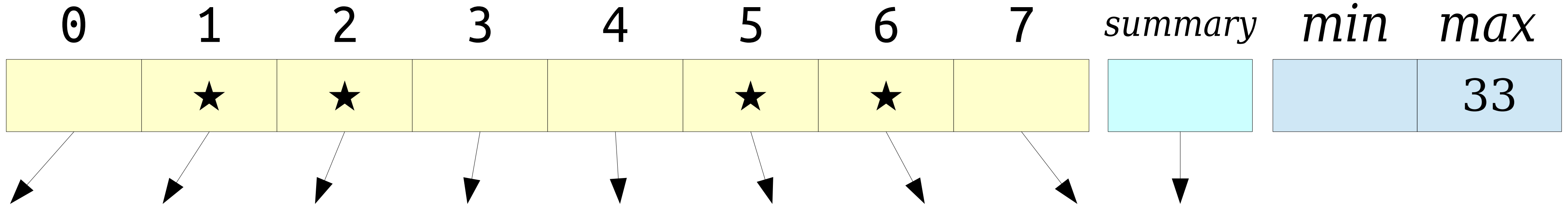
We need to find the minimum element in these buckets.



vEB Tree Deletions

- Deletions in a vEB tree work as before, but with extra logic to handle *min* and *max*.
  - May need to pull an element to fill in a missing *min* or *max*.

We need to find the minimum element in these buckets.

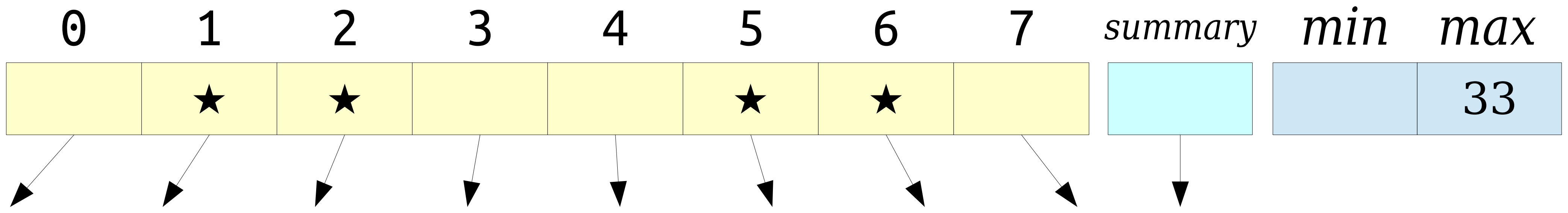


vEB Tree Deletions

- Deletions in a vEB tree work as before, but with extra logic to handle *min* and *max*.
  - May need to pull an element to fill in a missing *min* or *max*.

We need to find the minimum element in these buckets.

Ask the summary for the first nonempty block…

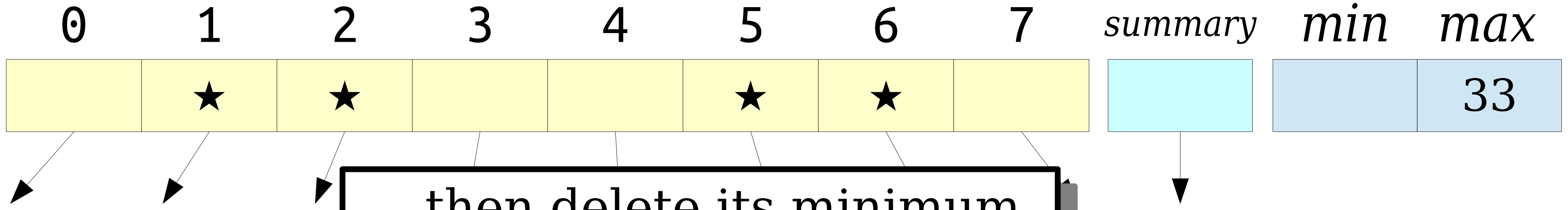


vEB Tree Deletions

- Deletions in a vEB tree work as before, but with extra logic to handle *min* and *max*.
  - May need to pull an element to fill in a missing *min* or *max*.

We need to find the minimum element in these buckets.

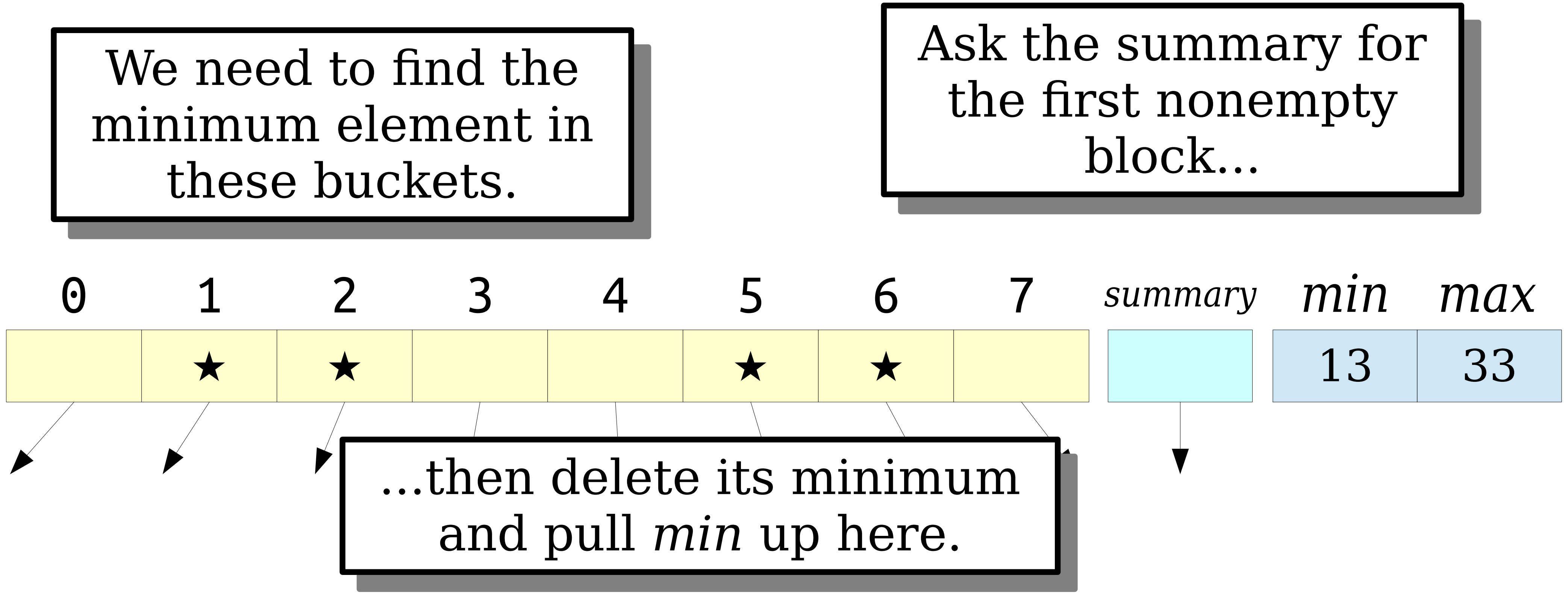
Ask the summary for the first nonempty block…



…then delete its minimum and pull *min* up here.

vEB Tree Deletions

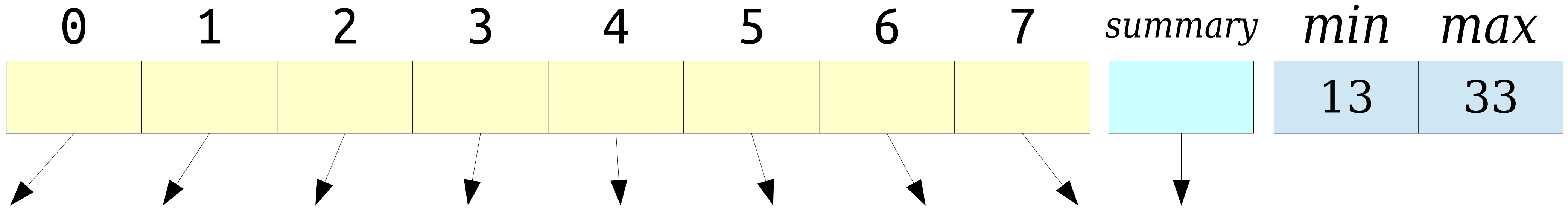
- Deletions in a vEB tree work as before, but with extra logic to handle *min* and *max*.
  - May need to pull an element to fill in a missing *min* or *max*.



We need to find the minimum element in these buckets.
Ask the summary for the first nonempty block…
0
1
2
3
4
5
6
7
summary
min
max
★
★
★
★
13
33
…then delete its minimum and pull *min* up here.

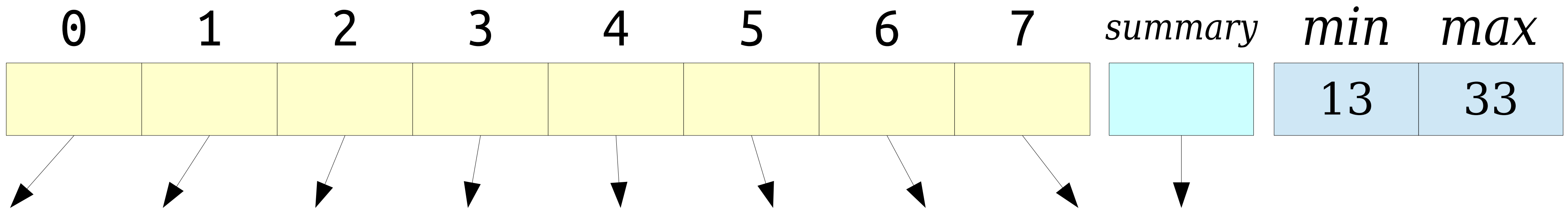
vEB Tree Deletions

- Deletions in a vEB tree work as before, but with extra logic to handle *min* and *max*.
  - May need to pull an element to fill in a missing *min* or *max*.



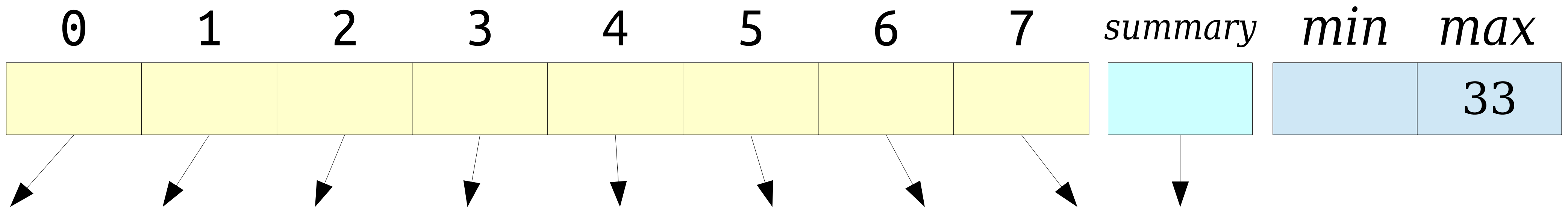
vEB Tree Deletions

- Deletions in a vEB tree work as before, but with extra logic to handle *min* and *max*.
  - May need to pull an element to fill in a missing *min* or *max*.
  - May need to clear *min* or *max*.



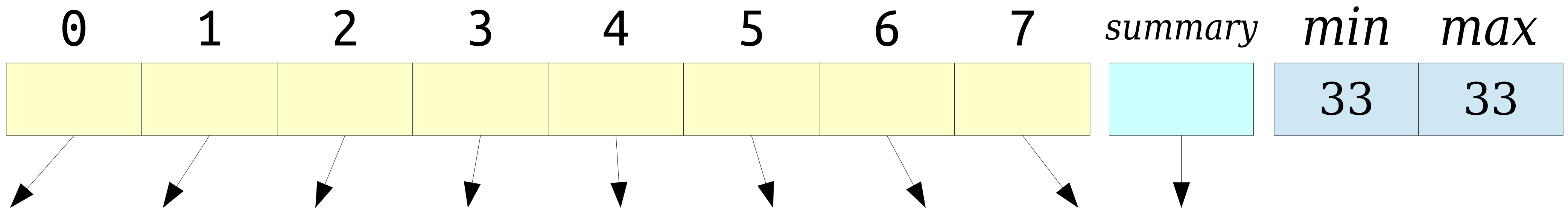
vEB Tree Deletions

- Deletions in a vEB tree work as before, but with extra logic to handle *min* and *max*.
    - May need to pull an element to fill in a missing *min* or *max*.
    - May need to clear *min* or *max*.



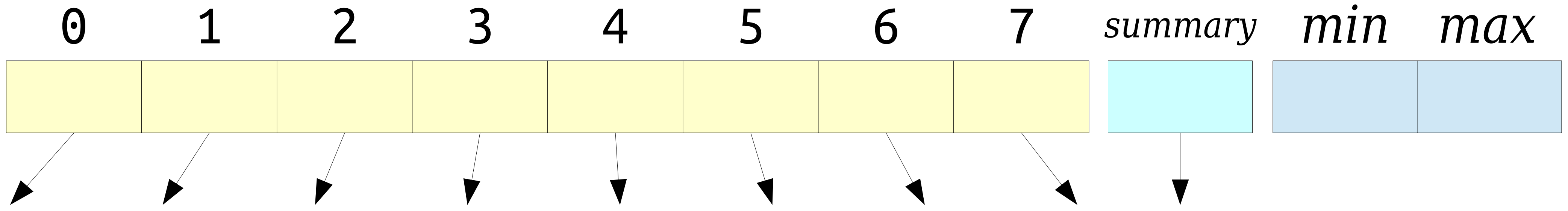
vEB Tree Deletions

- Deletions in a vEB tree work as before, but with extra logic to handle *min* and *max*.
  - May need to pull an element to fill in a missing *min* or *max*.
  - May need to clear *min* or *max*.



vEB Tree Deletions

- Deletions in a vEB tree work as before, but with extra logic to handle *min* and *max*.
  - May need to pull an element to fill in a missing *min* or *max*.
  - May need to clear *min* or *max*.



Why This Matters

- This may seem like a minor cosmetic change, but this fundamentally changes the analysis for two reasons:
  - The cost of a ***min***, ***max***, or ***is-empty*** query drops to O(1), reducing the cost of the other recurrences.
  - The cost of inserting into or deleting from an empty vEB tree is now worst-case O(1).
- Let's trace through these effects and see what happens.

Analyzing the Runtime

- This simple change profoundly affects the runtime of the operations for several reasons:
    - We can now instantly query for the min and max values in a tree.
    - The behavior of insert and delete changes slightly when working with empty or nearly empty trees.
- ***min***, ***max***, and ***is-empty*** run in time O(1).
- ***lookup*** runs in time O(log log U) as before.
- Let's revisit all the operations to see how efficiently they work.

Updating ***insert***

- The logic for ***insert***(x) works as follows:
  - If the tree is empty or has just one element, update *min* and *max* appropriately and stop.
  - Potentially displace the *min* or *max* and insert that value instead of x.
  - Insert x mod $U^{1/2}$ into the appropriate substructure.
  - Insert $\lfloor x / U^{1/2} \rfloor$ into the summary.
- Recurrence relation:

$$\begin{aligned} &T(2) = \Theta(1) \\ &T(U) = 2T(U^{1/2}) + \Theta(1). \end{aligned}$$

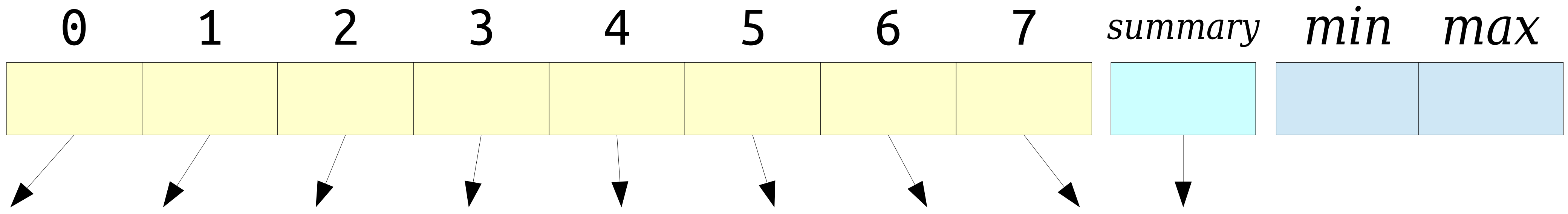
- This still solves to O(log U). Can we do better?

An Observation

- The summary structure stores the indices of the substructures that are nonempty.
- Therefore, we only need to insert $\lfloor x / U^{1/2} \rfloor$ into the summary if that block previously was empty.
- Here's our new approach:
    - If the $\lfloor x / U^{1/2} \rfloor$th substructure is not empty:
        - Call ***insert***(x mod $U^{1/2}$) into that substructure.
    - Otherwise:
        - Call ***insert***(x mod $U^{1/2}$) into that substructure.
        - Call ***insert***($\lfloor x / U^{1/2} \rfloor$) into the summary structure.

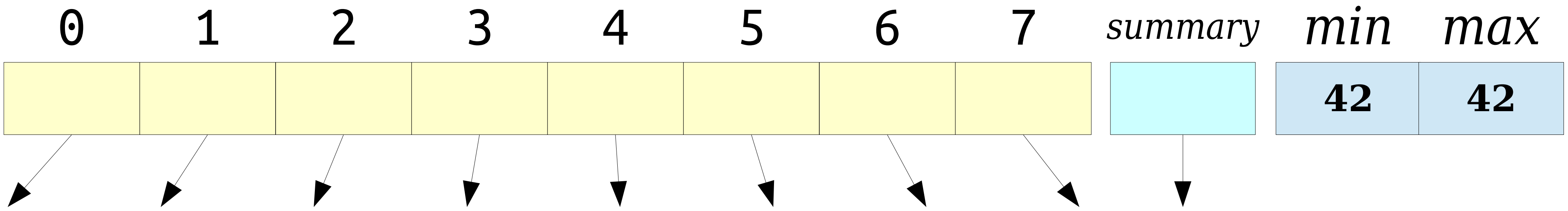
A Very Clever Insight

- ***Useful Fact:*** Inserting an element into an empty vEB tree takes time O(1).



A Very Clever Insight

- ***Useful Fact:*** Inserting an element into an empty vEB tree takes time O(1).



A Very Clever Insight

- ***Useful Fact:*** Inserting an element into an empty vEB tree takes time O(1).
- We only make at most one "real" recursive call:
  - If we don't recurse into the summary, we only made one recursive call down into a substructure.
  - If we make a recursive call into the summary, we did so because the other call was on an empty subtree, which isn't a "real" recursive call.
- New recurrence relation:

$$T(2) = \Theta(1)$$
$$T(U) \leq T(U^{1/2}) + \Theta(1)$$

- As we've seen, this solves to **O(log log *U*)**.

Analyzing ***delete***

- The logic for ***delete***(x) works as follows:
  - If the tree has just one element, update *min* and *max* appropriately and stop.
  - If *min* or *max* are being deleted, replace them with the *min* or *max* of the first or last nonempty tree, then proceed as if deleting that element instead.
  - Delete $x \bmod U^{1/2}$ from its subtree.
  - If that subtree is empty, delete $\lfloor x / U^{1/2} \rfloor$ from the summary.
- Recurrence relation:

$$T(2) = \Theta(1)$$
$$T(U) \leq 2T(U^{1/2}) + \Theta(1).$$

- Still solves to O(log U). However, is this bound tight?

A Better Analysis

- ***Observation:*** Deleting the last element out of a vEB tree takes time O(1).
  - Just need to update the *min* and *max* fields.
- Therefore, ***delete*** makes at most one “real” recursive call:
  - If it empties a subtree, the recursive call that did so ran in time O(1) and the “real” call is on the summary structure.
  - If it doesn't, then there's no second call on the summary structure.

The New Runtime

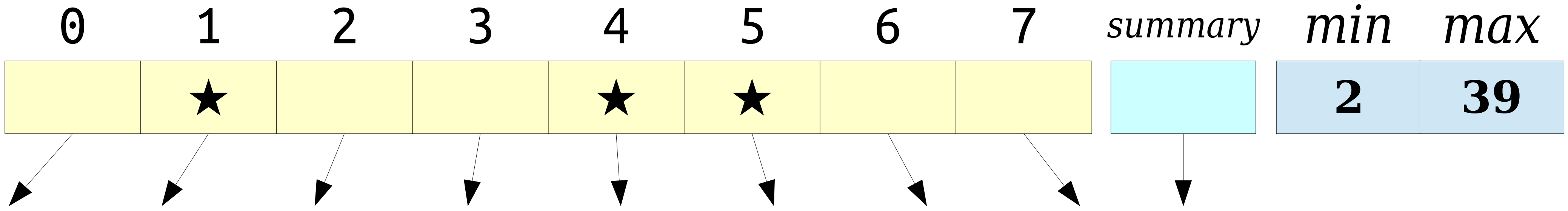
- With this factored in, the runtime of doing an ***delete*** is given by the recurrence

$$T(2) = \Theta(1)$$
$$T(U) \leq T(U^{1/2}) + \Theta(1)$$

- As we've seen, this solves to **O(log log *U*)**.

Finding a Successor

- In a vEB tree, we can find a successor as follows:
  - If the tree is empty or x > ***max***(), there is no successor.
  - Otherwise, let i be the index of the tree containing x.
  - If subtree i is nonempty and x is less than i's max, x's successor is the successor in subtree i.
  - Otherwise, find the successor j of i in the summary.
  - If j exists, return the minimum value in tree j.
  - Otherwise, return the tree max.



Finding a Successor

- In a vEB tree, we can find a successor as follows:
  - If the tree is empty or x > ***max***(), there is no successor.
  - Otherwise, let i be the index of the tree containing x.
  - If subtree i is nonempty and x is less than i's max, x's successor is the successor in subtree i.
  - Otherwise, find the successor j of i in the summary.
  - If j exists, return the minimum value in tree j.
  - Otherwise, return the tree max.
- At most one recursive call is made and each other operation needed runs in time O(1).
- Recurrence: $T(U) \leq T(U^{1/2}) + \Theta(1)$; solves to **$O(\log \log U)$**.

van Emde Boas Trees

- The van Emde Boas tree supports insertions, deletions, lookups, successor queries, and predecessor queries in time O(log log U).
- It can answer min, max, and is-empty queries in time O(1).
- If $n = \omega(\log U)$, this is ***exponentially faster*** than a balanced BST!

The Catch

- There is, unfortunately, one way in which vEB trees stumble: ***space usage***.
- We've assumed that the complete vEB tree has been constructed before we make any queries on it.
- How much space does it use?

The Recurrence

- The space usage of a van Emde Boas tree is given by the following recurrence relation:

$$S(2) = \Theta(1)$$

$$S(U) = (U^{1/2} + 1)S(U^{1/2}) + \Theta(U^{1/2})$$

- Using the substitution method, this can be shown to be $\Theta(U)$.
- Space usage is proportional to the size of the universe, not the number of elements stored!

Reducing Space Usage

- We can cut the space usage for a vEB tree down by using hash tables at each level instead of arrays.
- Using cuckoo hashing, we get the same worst-case time bounds on each operation except for ***insert***.
- This drops the space usage down to O(n).
- However, this requires a *lot* of different hash tables, and that's expensive!

Next Time

- ***x*-Fast Tries**
  - A randomized data structure matching the vEB bounds and using O(n log U) space.
- ***y*-Fast Tries**
  - A randomized data structure matching the vEB bounds in an amortized sense and using O(n) space.
- (These data structures pull together just about everything we've covered this quarter – I hope they make for great midterm review!)