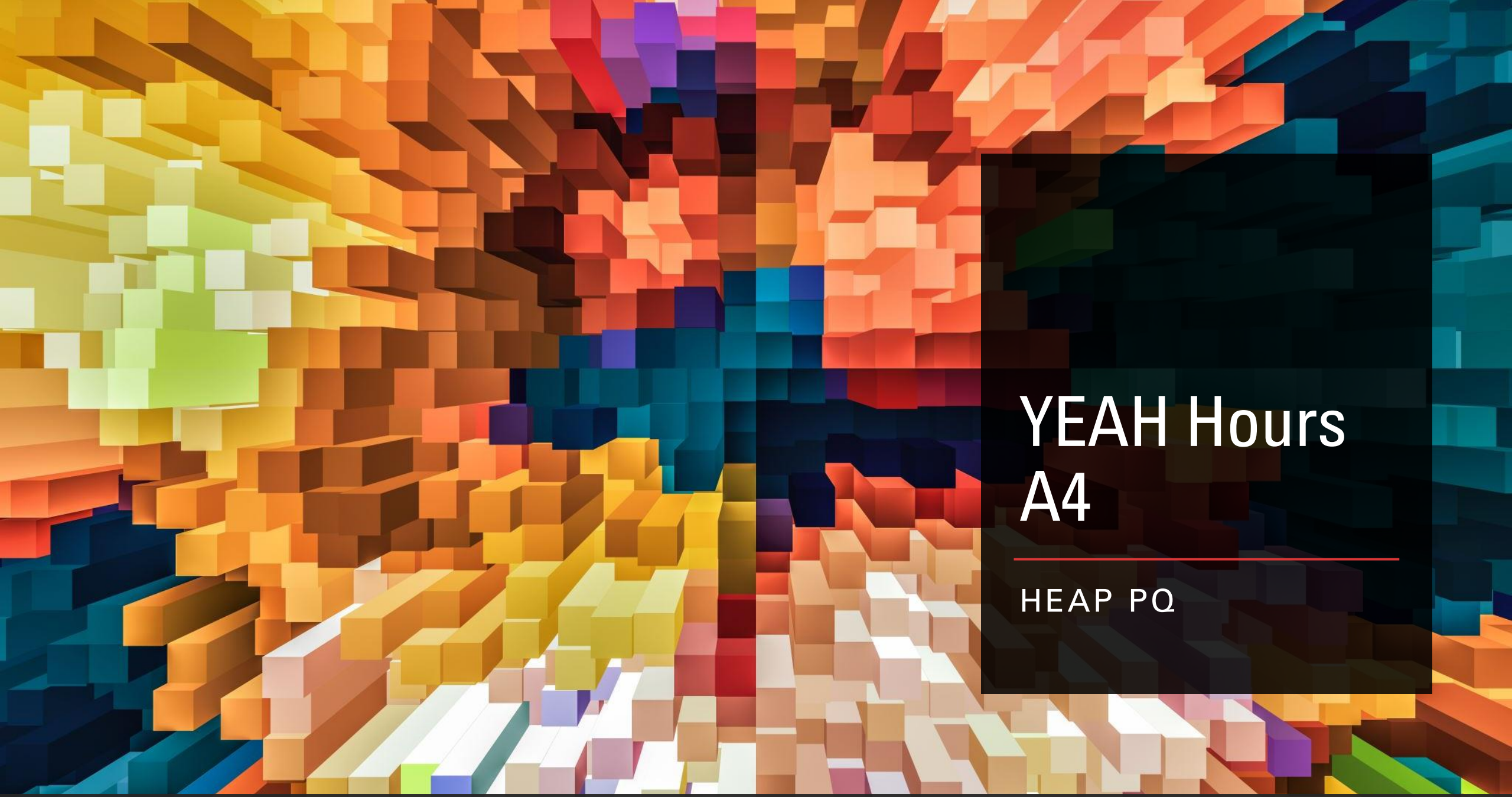


# YEAH Hours A4

## HEAP PQ


# Let's take a second…

- Congrats, you're past the halfway point in the quarter!
  - Take a second to pat yourself on the back. This is hard stuff, and you're doing great ☺

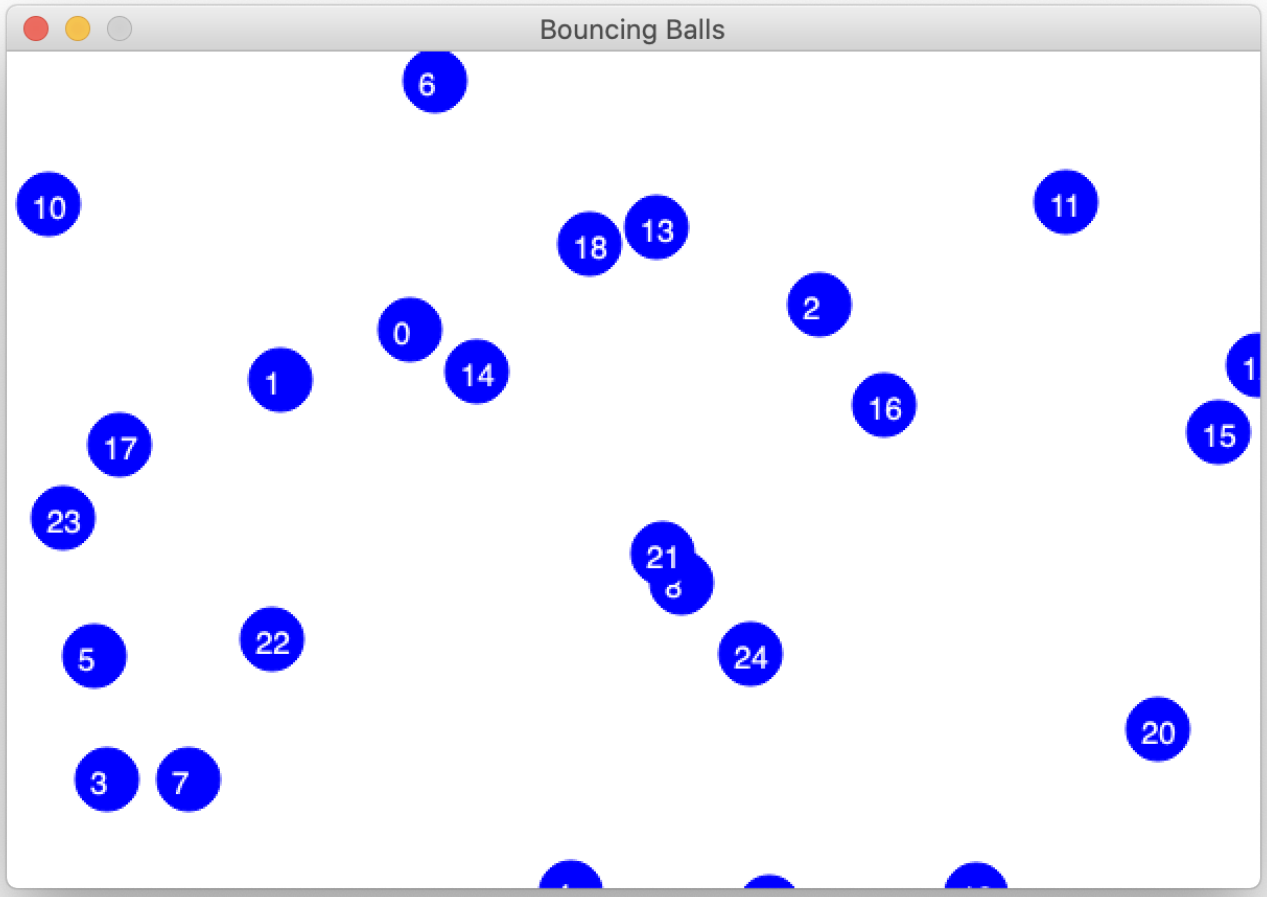
Stack Efron, CS106B alum,
congratulating on a job well done so far!

# The Breakdown:

1. Warmups – Two exercises in which you learn more helpful tips about using the debugger. We **highly** recommend paying close attention to these in the handout, because debugging the PQ assignment is historically quite difficult – these were designed to help!
2. Part 1: PQ Sorted Array – Implement enqueue() in a self-sorting priority queue!
3. Part 2: Streaming Top-K – Using a **priorityqueue**, what kinds of powerful things can you do?
4. Part 3: Heap PQ – Implement a **priorityqueue** using a binary min-heap!

# Warmup Debrief

- In this week's warmups, you'll **first** examine a **bouncing balls** program to learn about debugging objects. You'll also learn how to set **conditional breakpoints**, breakpoints that only trigger when the program is at a pre-defined state.



# Warmup Debrief

- In the second half of the warmups, you'll work with debugging arrays in c++.
- As you can see, you'll be using a new data structure called a **DataPoint**. It's pretty simple – you can see its contents to the right!
- The above code is buggy, and you'll need to step through it in the debugger in order to expose its **memory errors!**

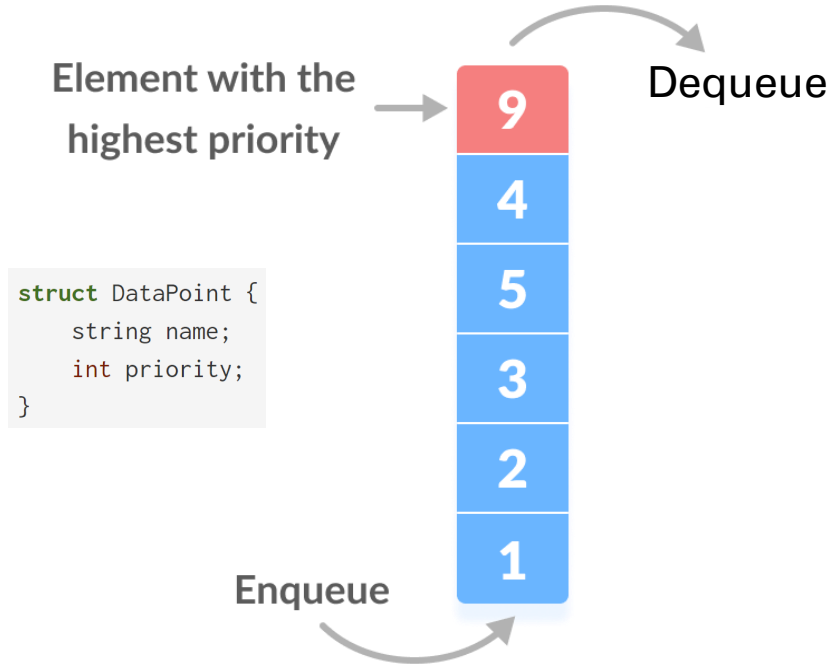
```
PROVIDED_TEST("Test valid use of array, allocate/access/deallocate") {
    DataPoint* shoppingList = new DataPoint[6]; // allocate

    for (int i = 0; i < 3; i++) {
        shoppingList[i].name = "eggs"; // set struct field by field
        shoppingList[i].priority = 10 + i;
    }
    shoppingList[0].priority += 5;
    delete[] shoppingList; // deallocate
}
```

```
struct DataPoint {
    string name;
    int priority;
}
```

# What's a priority queue?

- A priority queue, or a **pq** as lazy computer scientists like to say, is a **queue-like** data structure (think enqueue() and dequeue()), but it has a cool extra feature!
  - All elements in a **pq** are assigned a **priority** upon enqueue(), and that **priority** determines the order that they will be dequeue()'d in!
  - For this assignment, your **pq** will store **DataPoint**'s, which have an internal priority value!
  - A **pq** can either prioritize **high priorities** or **low priorities**, meaning that the element dequeue()'d will always be the one with the **highest** or **lowest** priority.
    - We'll be very clear about which magnitude we care about each time ☺.




```
struct DataPoint {
    string name;
    int priority;
}
```

A "max" priotiy queue. Notice how the structure doesn't have to be sorted, so long as the "highest priority element" is always next to be dequeue()'d

# Part 1: PQ Sorted Array

- For this first part, we're giving you **almost fully implemented priority queue .h and .cpp files!**
- The data structure that stores the pq is **an array of elements**, much like you've seen in lecture and section examples!
  - In this particular array, **all elements are sorted from high to low priorities (front to back), and the smallest value priority element will be dequeue()'d first!**
  - That makes this pq a **min** priority queue!
- How does this queue work?

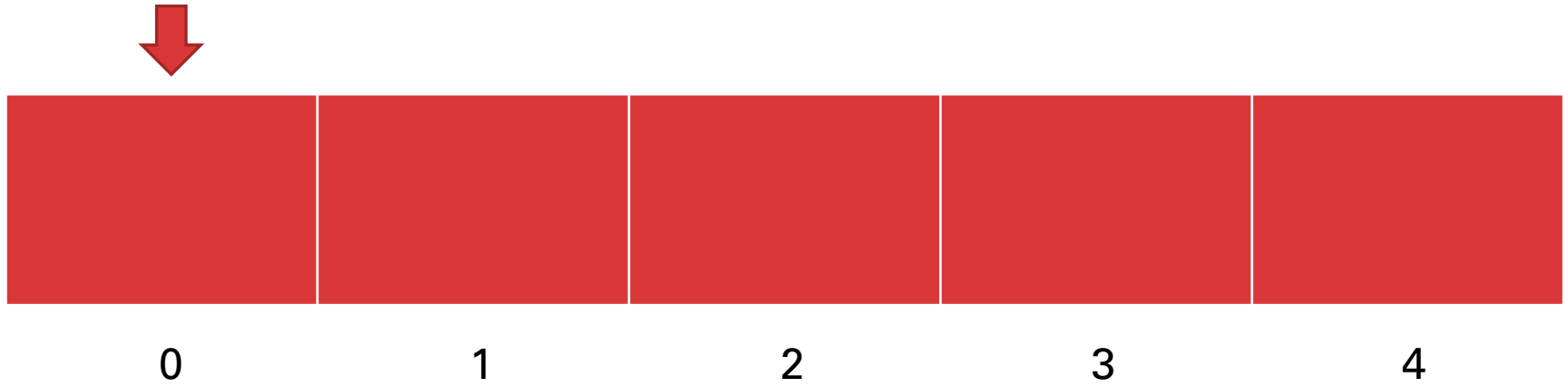
# Let's see an example…

**PQ.enqueue(10);**

Disclaimer: to simplify things, I'll represent **DataPoints** simply as their **priorities**

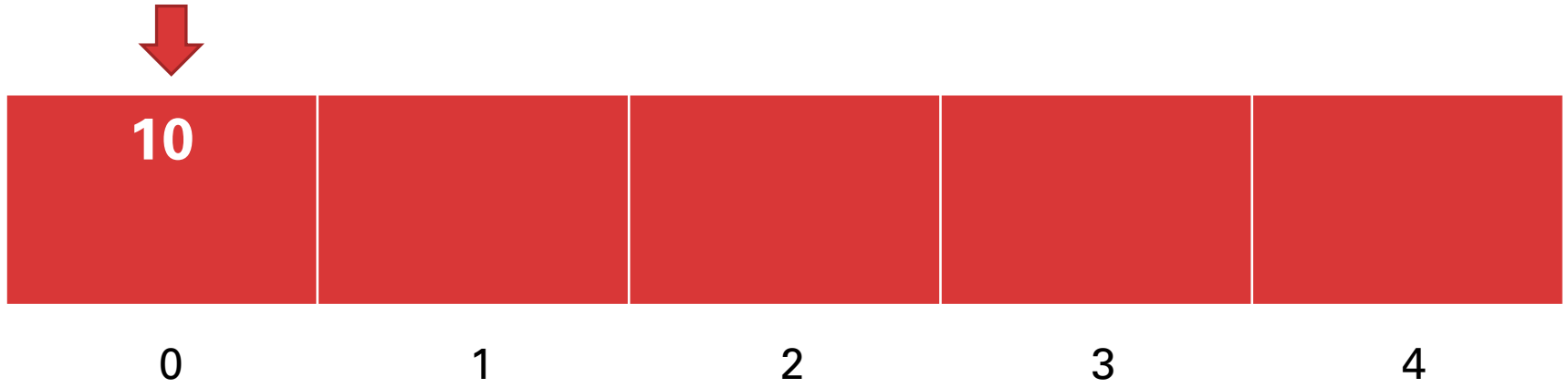
# Let’s see an example…

**PQ.enqueue(10);**



# Let’s see an example…

**PQ.enqueue(10);**

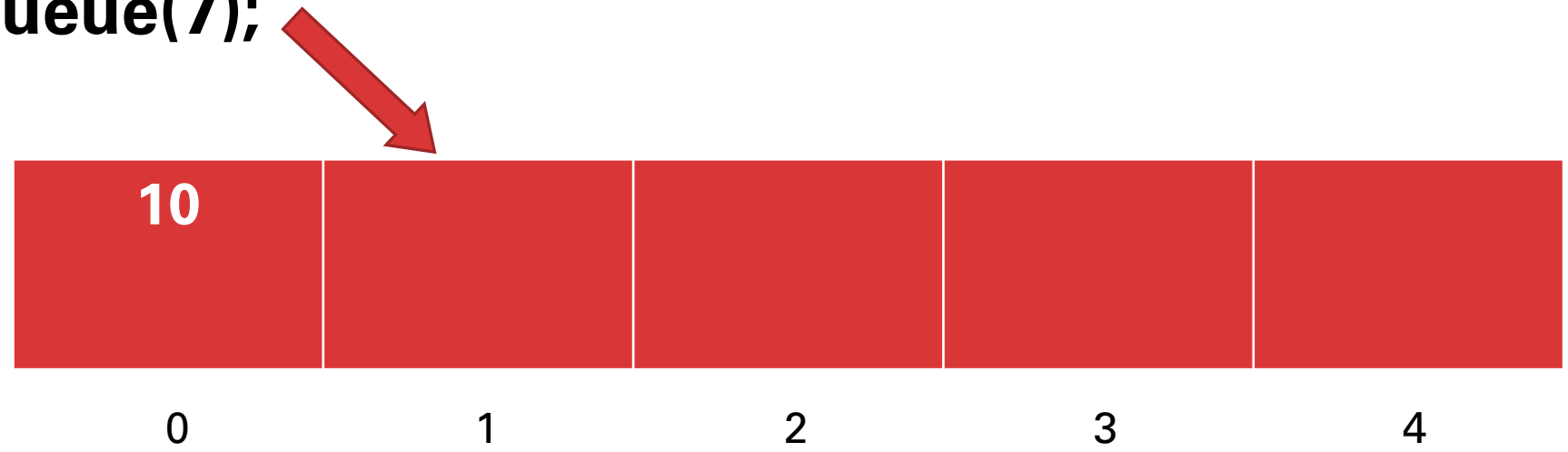


# Let’s see an example…

**PQ.enqueue(7);**

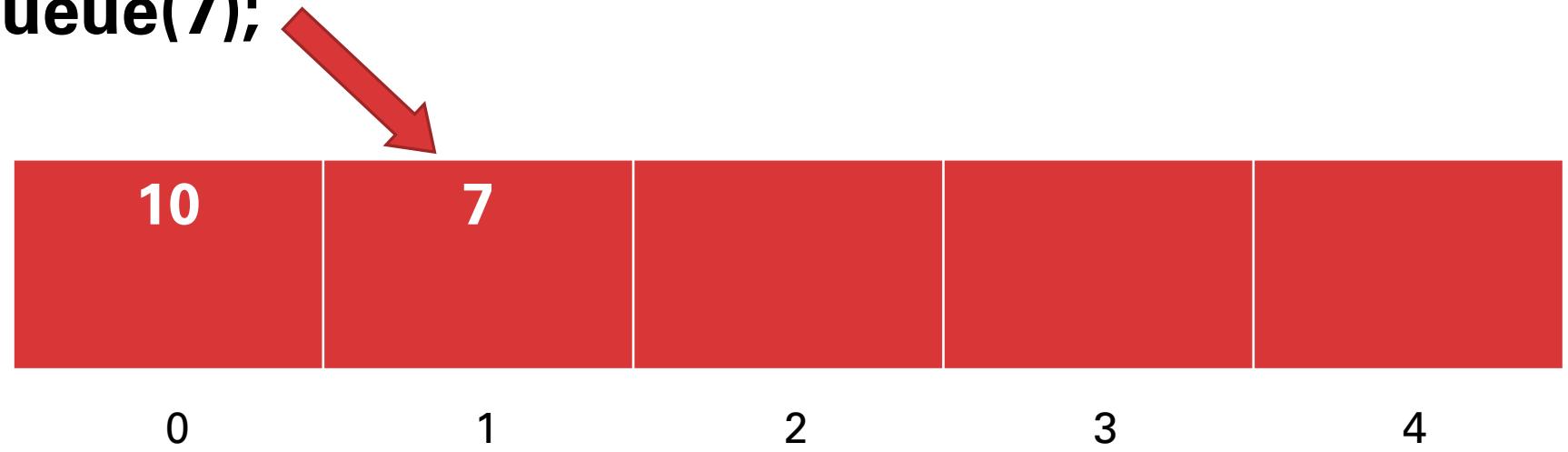
# Let’s see an example…

**PQ.enqueue(7);**



# Let’s see an example…

**PQ.enqueue(7);**

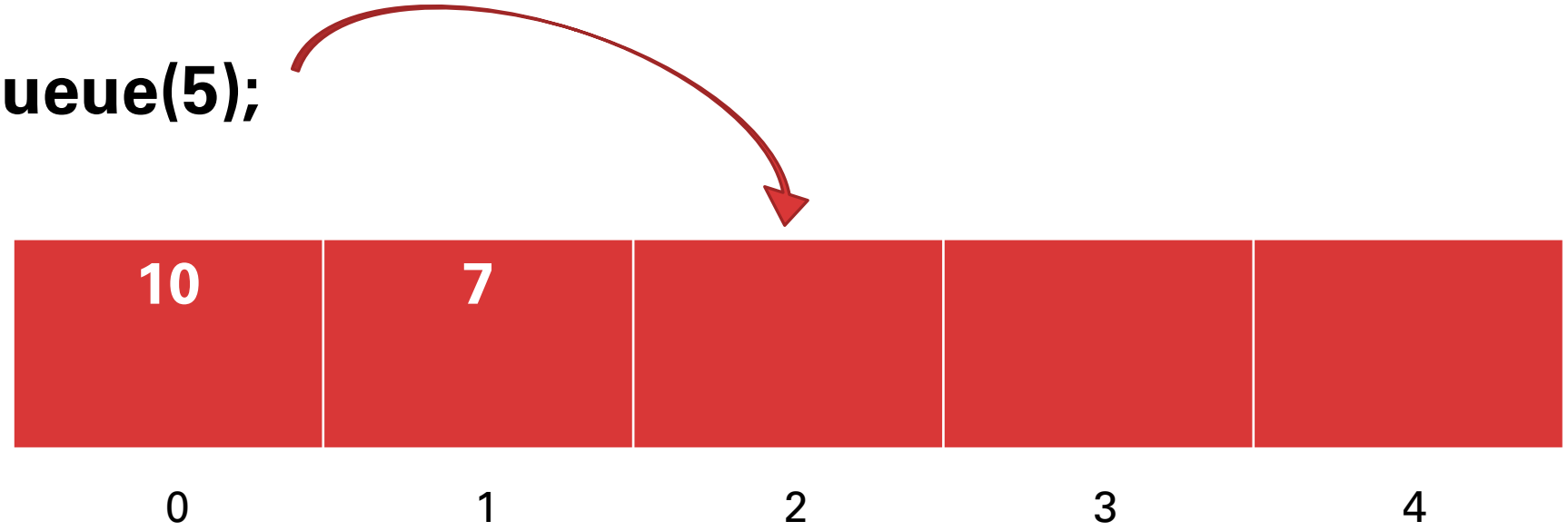


# Let’s see an example…

**PQ.enqueue(5);**

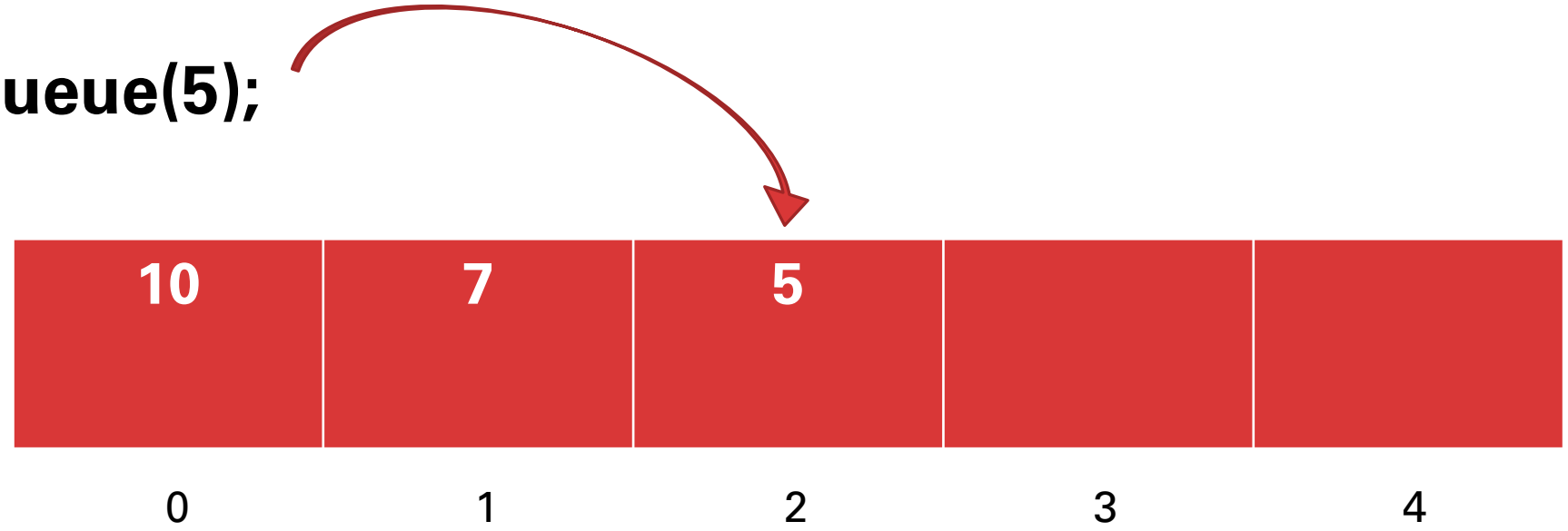
# Let’s see an example…

**PQ.enqueue(5);**



# Let’s see an example…

**PQ.enqueue(5);**

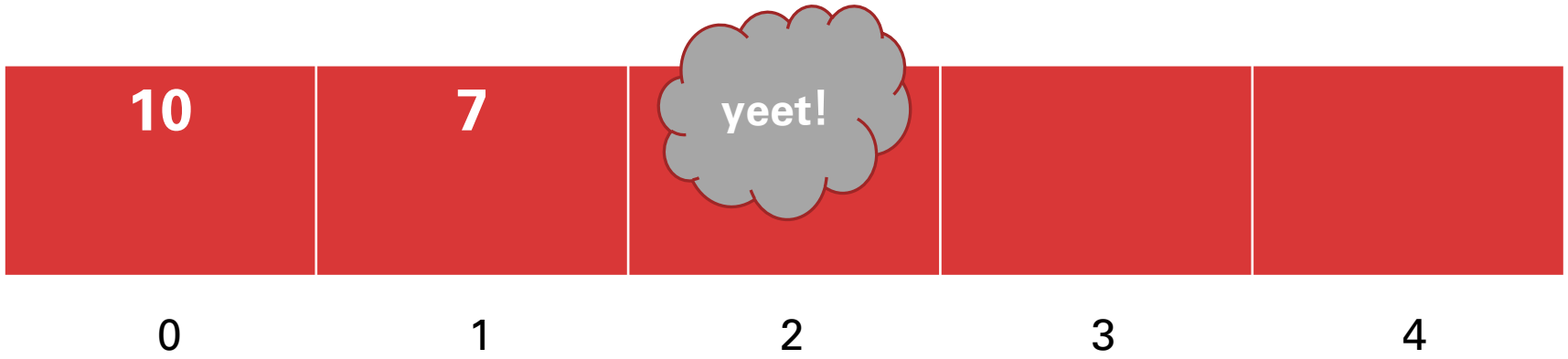


# Let’s see an example…

**PQ.dequeue();**

# Let’s see an example…

**PQ.dequeue();**



# Let’s see an example…

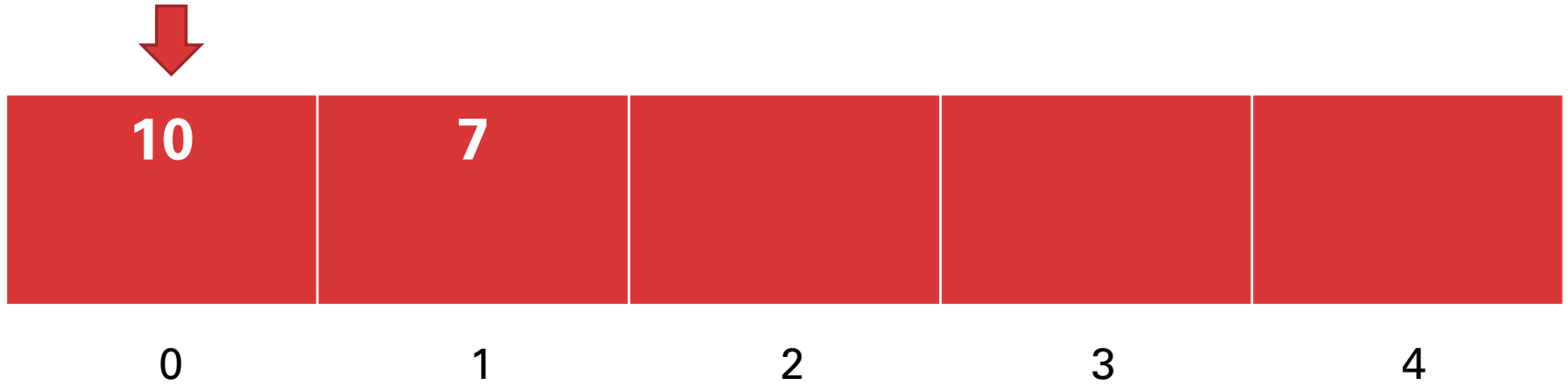
**PQ.dequeue();**

# Let’s see an example…

**PQ.enqueue(20);**

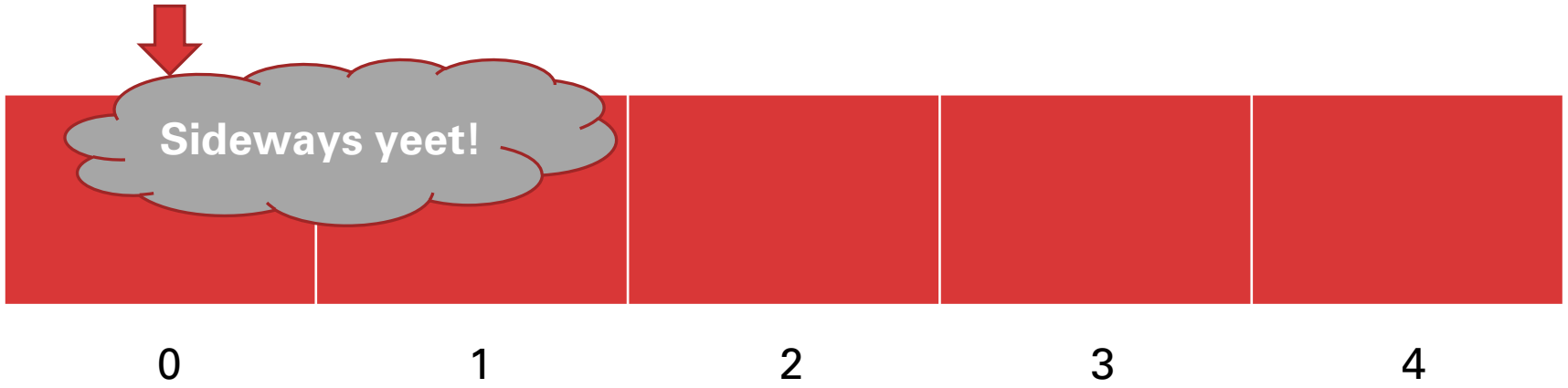
# Let’s see an example…

**PQ.enqueue(20);**



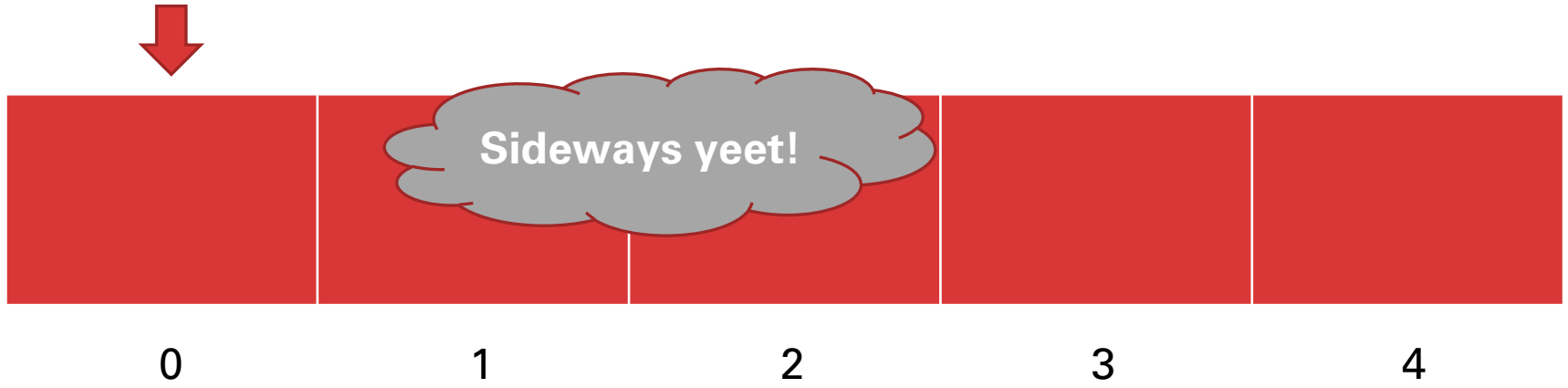
# Let’s see an example…

**PQ.enqueue(20);**



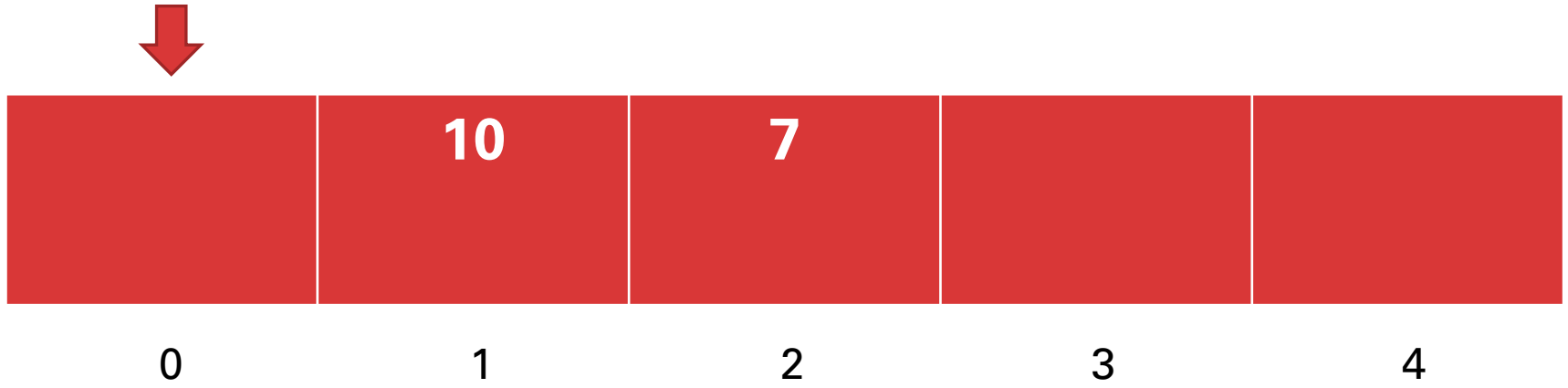
# Let’s see an example…

**PQ.enqueue(20);**



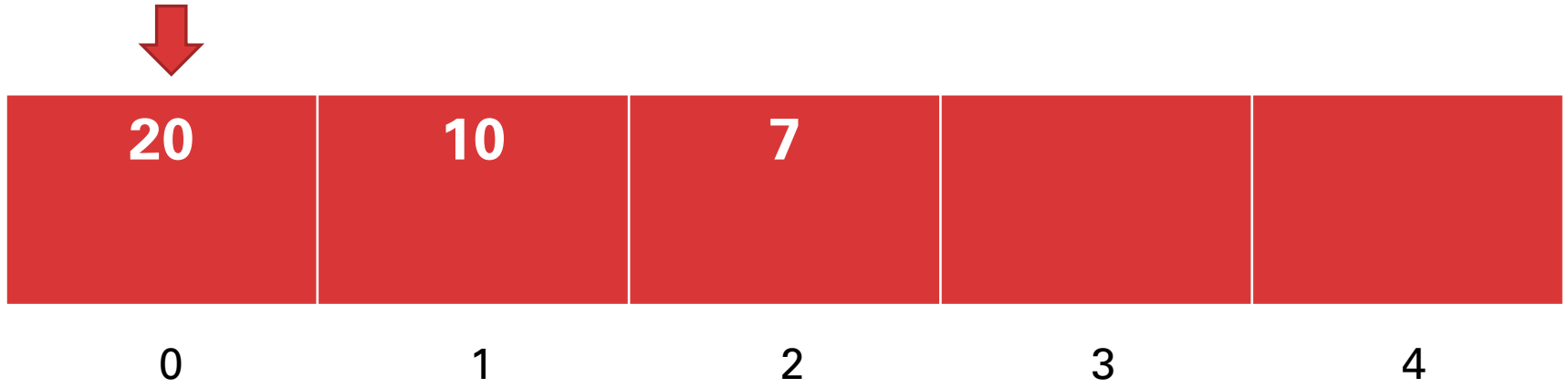
# Let’s see an example…

**PQ.enqueue(20);**



# Let’s see an example…

**PQ.enqueue(20);**



# Part 1: PQ Sorted Array

- In this part of the assignment, you'll be asked to implement a **single method** in the `pqsortedarray.cpp` file: the **enqueue**(DataPoint element) method!
- The rest of the `pqsortedarray.cpp` `pqsortedarray.h` are completed for you!
- You are responsible for **inserting** the provided element in the correct place in the array to preserve the sorted order.
  - If you are not appending to the end of the array, you will have to **shift** the contents of the array over in order to accommodate this new element.
  - **If you attempt the enqueue() an element when the array is full, you are responsible for resizing the array. We recommend doubling the current capacity.**

# Part 1: PQ Sorted Array

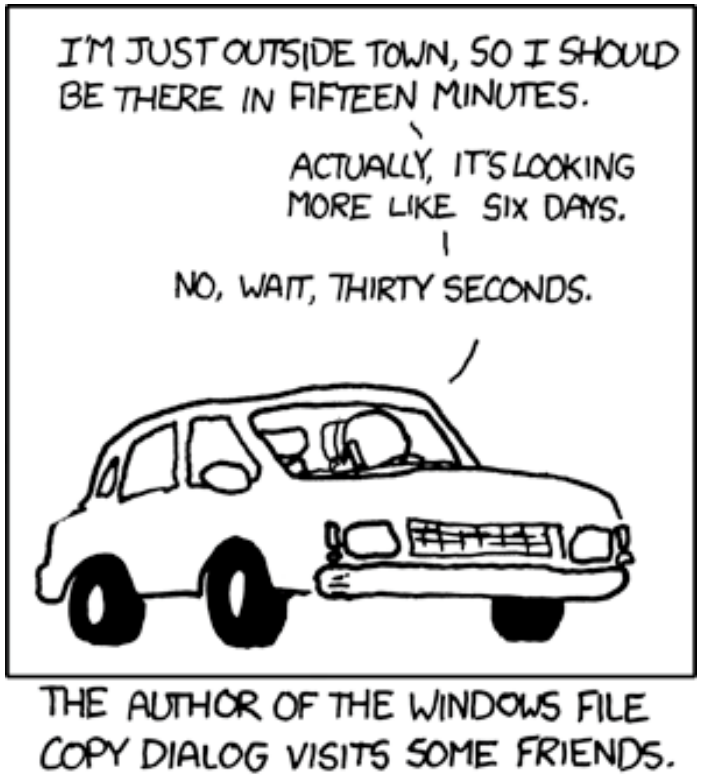
- Because verifying the internal state of your array can be tricky, we've also written you a function header called **validateInternalState()** that you can call after your shiny new **enqueue()** method.
  - This function can traverse your array to verify that it's in sorted order after you enqueue a **DataPoint**.
  - We **strongly** recommend that you implement this helper method – it'll make your debugging life a lot easier!

# Part 1: PQ Sorted Array

Helpful hints:

- You might want to make the resize() method a private helper method – it makes for a cleaner implementation.
- Apart from enqueue() and validateInternalState(), **you may not** modify any other functions. Adding helpers is okay, though!
- Not sure how to resize an array? Take a look at Section 5's RBQueue example!
- Debugging this one can be tricky, because there can be subtle edge cases. To expose these bugs, **stepping through with the debugger** and **using validateInternalState()** will be helpful!

# Questions about Part 1?




This xkcd isn't actually relevant to the material, but as a proud Windows user, this hits a little too close to home.

# Part 2: Client Tasks

- In this part of the assignment, you will be a **client**, or a user, of the pq class.
- With a pq, you can do some really powerful things! The code to the right sorts a vector using just **enqueue!** and **dequeue()!** Take a second to see why this works.
- You'll make a big-O guess about this sorting algorithm and then time it to verify your prediction!
- *Follow up question: Would this still work if your priority queue was <u>not</u> backed by a <u>sorted</u> array?*

```
void pqSort(Vector<DataPoint>& v) {
    PQSortedArray pq;

    /* Add all the elements to the priority queue. */
    for (int i = 0; i < v.size(); i++) {
        pq.enqueue(v[i]);
    }

    /* Extract all the elements from the priority queue. Due
     * to the priority queue property, we know that we will get
     * these elements in sorted order, in order of increasing priority
     * value.
     */
    for (int i = 0; i < v.size(); i++) {
        v[i] = pq.dequeue();
    }
}
```

# Part 2: Client Tasks

- For the next step, you'll be implementing the function **Vector**<DataPoint> **topK**(istream& stream, int k);
- An **istream** is a special abstraction that acts like a massive data structure. Streams allow you to move around massive amounts of memory because they don't need to hold the data in your computer's memory all at once – as you read data from the stream, the stream can read more data from its source – a file on disk for example!
  - You won't need to worry about the inner-workings of streams in this class, but it's important to know that **streams can store huge amounts of data.**
- In the above function, your job is harness the power of the PQ in order to return a **Vector**<DataPoint> of the *k* highest-priority points in the stream.
- You must do so in **O(k)** space, meaning you can only store *k* elements in your priority queue at any given time.

# Part 2: Client Tasks

- You will need to return the *k* largest elements in a Vector<DataPoint> sorted in **largest to smallest** order.
  - Note that it's very easy to get this backwards! **pq.dequeue() returns the SMALLEST element in the queue, which should go at the END of the vector.**
  - The vector .reverse() method might be helpful here ☺

# Part 2: Client Tasks

Tips / Tricks

- Here's how you can loop through every DataPoint in the stream ->

```
DataPoint point;
while (stream >> point)  {
    /* do something with point */
}
```

- Because you can only store *k* elements at a time, how can you use the priority queue to your advantage?
  - When your pq has *k* elements in it, what's special about the element returned by pq.peek()?

# Questions about Top-K?

streaming
Netflix

streaming
Top-K

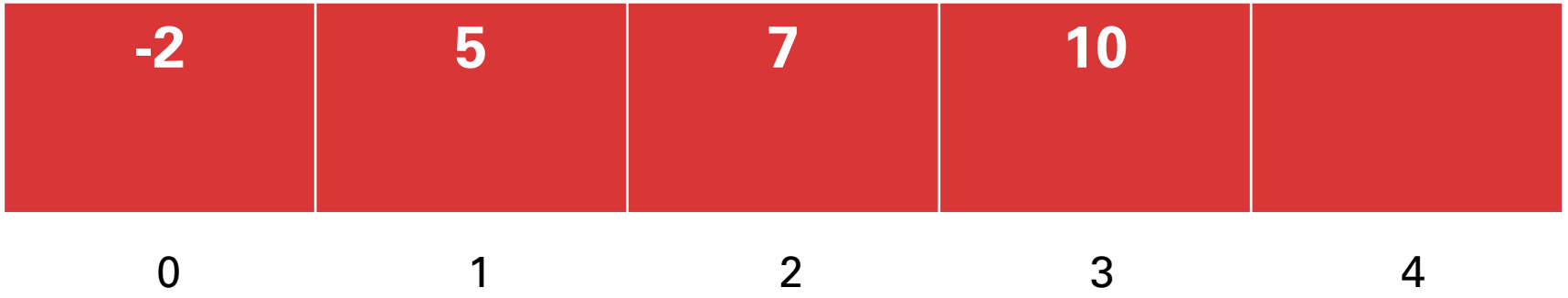
# Part 3: Heap PQ

- In this final part, you'll be implementing a full priority queue using a binary min heap!
  - In this case, we mean that the "most prioritized" element is the element with the smallest value.
    - Just like on the sorted array part!
  - In order to keep that property in your queue, you will be using a min heap like you've seen in lecture!
- The lecture from 7/21 is an excellent source for all you'll need to know about how to implement one of these heaps!
- Moreover, the non heap-related code you have may end up looking quite a bit like the code already written for you in PQSortedArray!
- Let's go over a few key points

# Part 3: Heap PQ

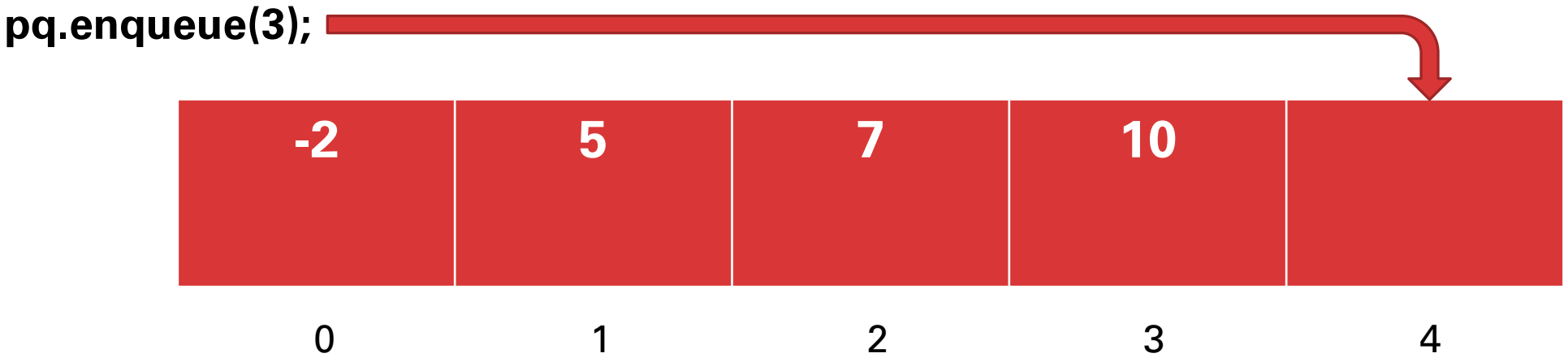
- Let's talk about enqueue()!
  - To enqueue an element, first add it to the end of your pqueue!

**pq.enqueue(3);**



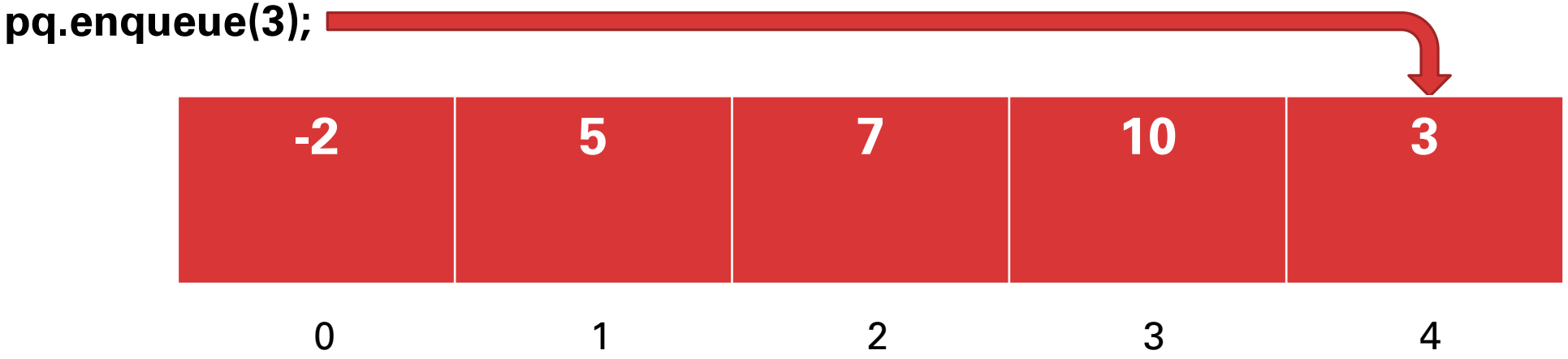
# Part 3: Heap PQ

- Let's talk about enqueue()!
  - To enqueue an element, first add it to the end of your pqueue!



# Part 3: Heap PQ

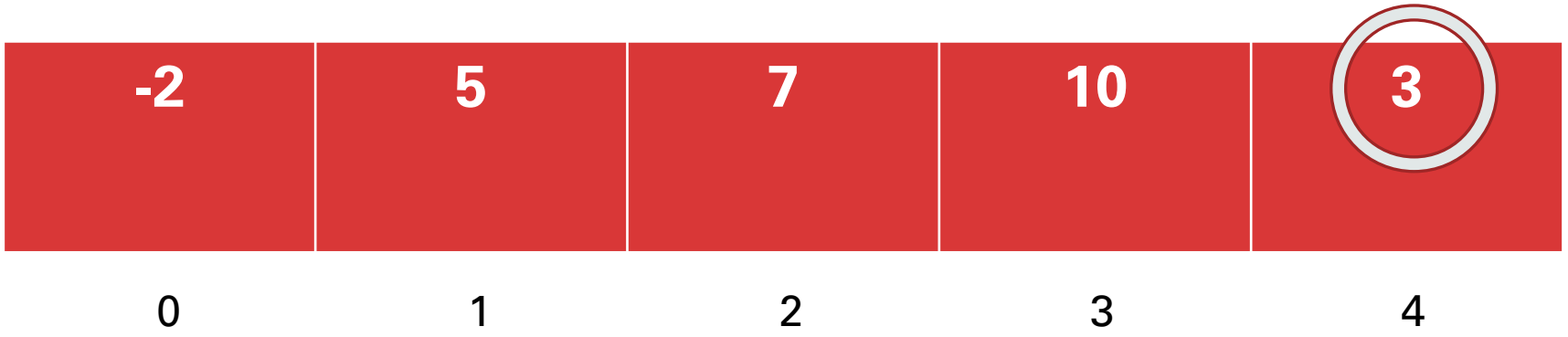
- Let's talk about enqueue()!
  - To enqueue an element, first add it to the end of your pqueue!



# Part 3: Heap PQ

- Let's talk about enqueue()!
  - To enqueue an element, first add it to the end of your pqueue!
  - Next, bubble the element up (if a parent exists!) Compare it with its parent at index (i-1)/2. Swap if your element **is less than** its parent!

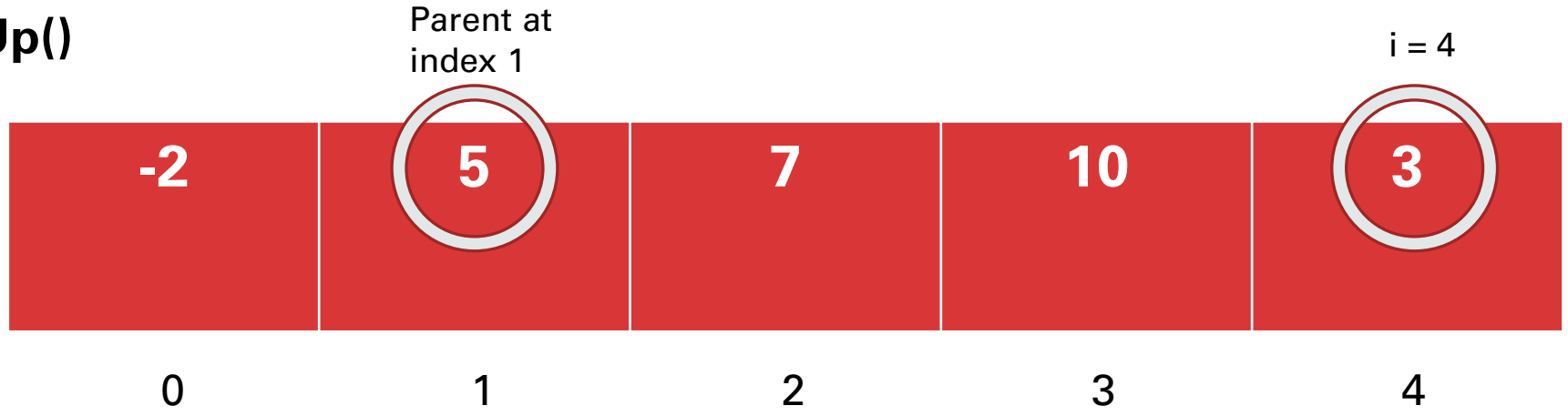
**bubbleUp()**



# Part 3: Heap PQ

- Let's talk about enqueue()!
  - To enqueue an element, first add it to the end of your pqueue!
  - Next, bubble the element up (if a parent exists!) Compare it with its parent at index (i-1)/2. Swap if your element **is less than** its parent!

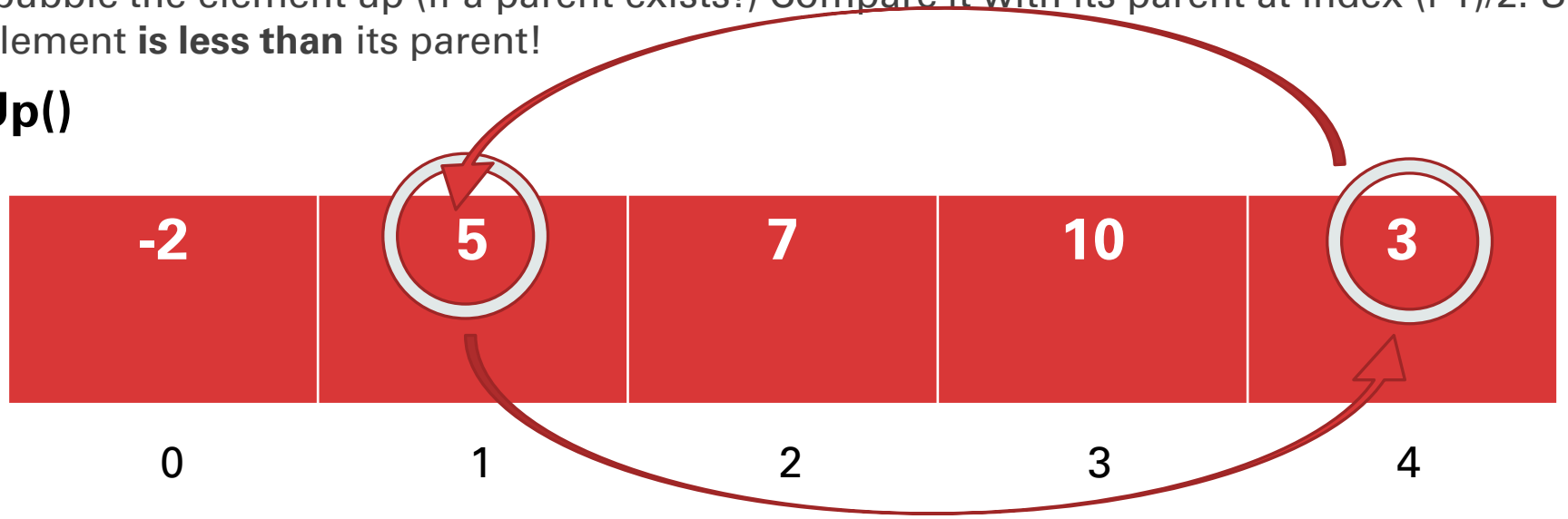
**bubbleUp()**



# Part 3: Heap PQ

- Let's talk about enqueue()!
  - To enqueue an element, first add it to the end of your pqueue!
  - Next, bubble the element up (if a parent exists!) Compare it with its parent at index (i-1)/2. Swap if your element **is less than** its parent!

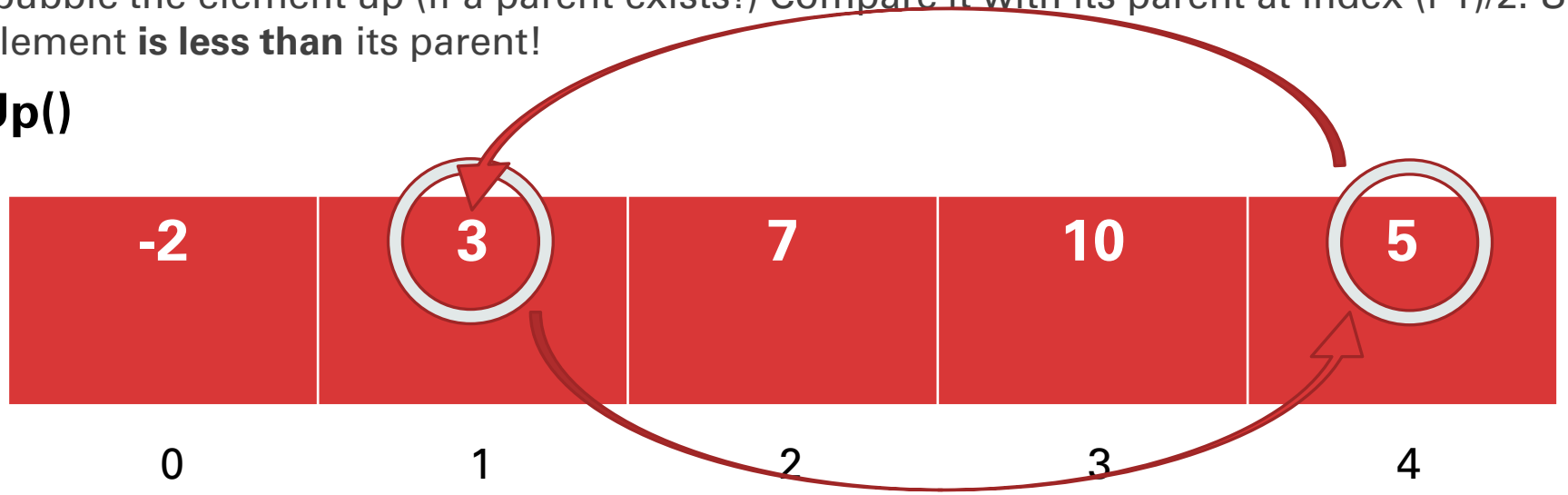
**bubbleUp()**



# Part 3: Heap PQ

- Let's talk about enqueue()!
  - To enqueue an element, first add it to the end of your pqueue!
  - Next, bubble the element up (if a parent exists!) Compare it with its parent at index (i-1)/2. Swap if your element **is less than** its parent!

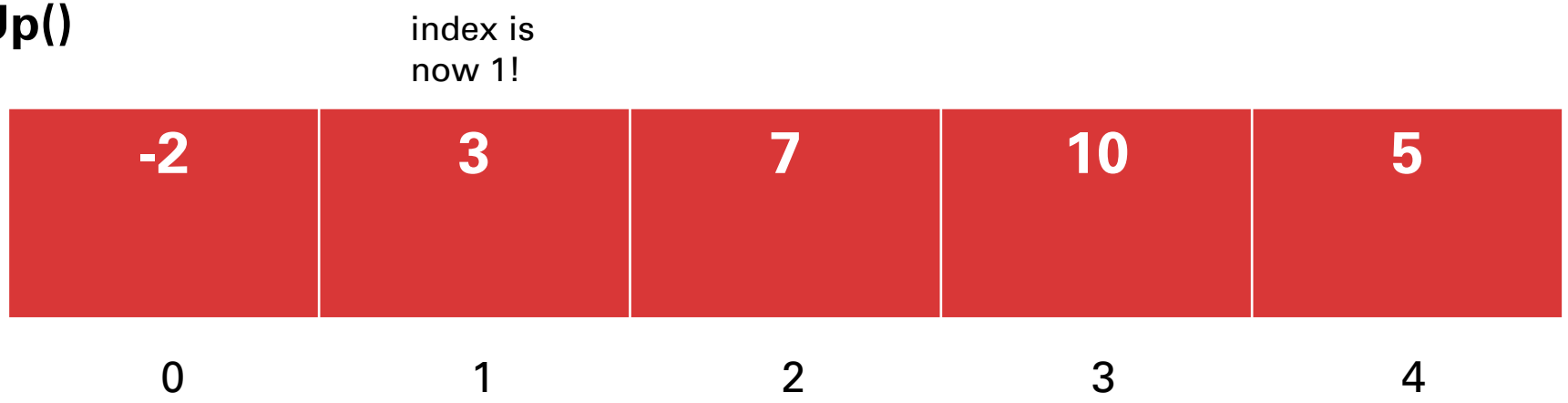
**bubbleUp()**



# Part 3: Heap PQ

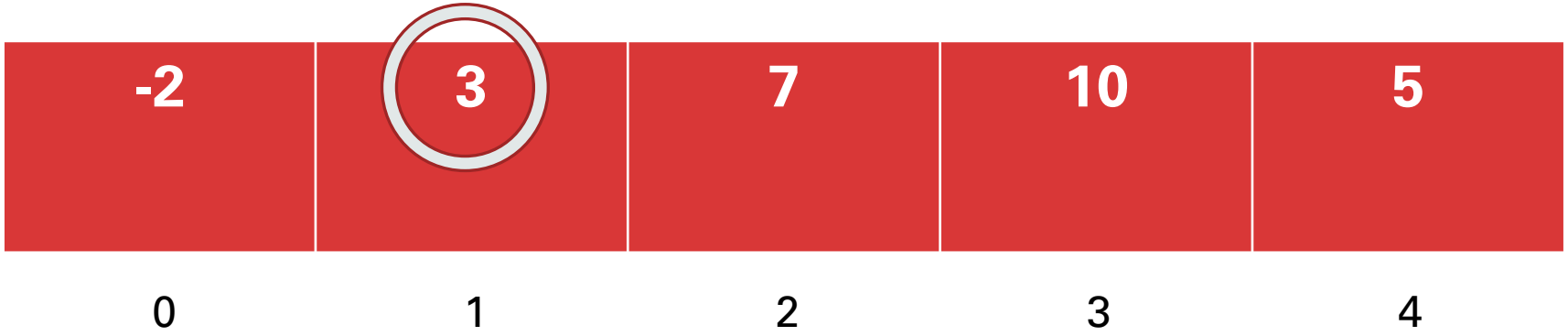
- Let's talk about enqueue()!
  - To enqueue an element, first add it to the end of your pqueue!
  - Next, bubble the element up (if a parent exists!) Compare it with its parent at index (i-1)/2. Swap if your element **is less than** its parent! Remember to update your current index to reflect the swap ☺

**bubbleUp()**



# Part 3: Heap PQ

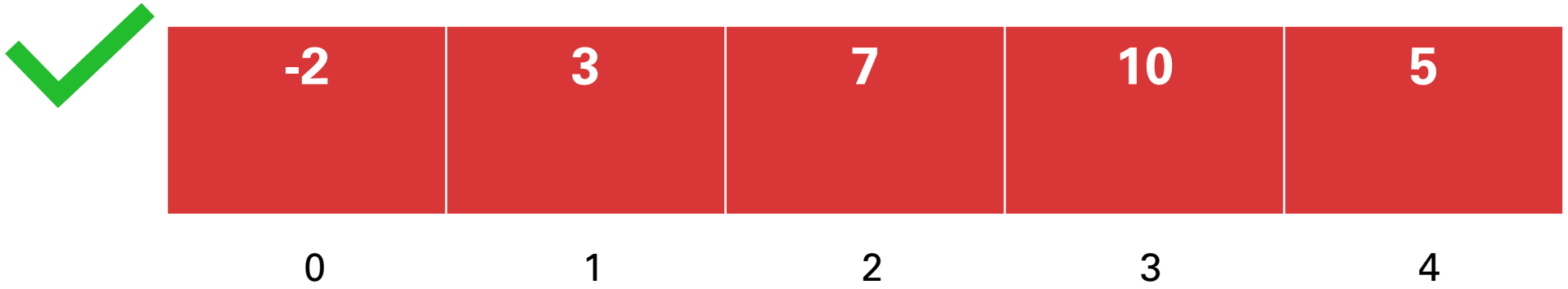
- Let's talk about enqueue()!
  - To enqueue an element, first add it to the end of your pqueue!
  - Next, bubble the element up (if a parent exists!) Compare it with its parent at index (i-1)/2. Swap if your element **is less than** its parent! Remember to update your current index to reflect the swap ☺
  - Repeat this process until either your parent is smaller than you, or you're at the top of the heap!



# Part 3: Heap PQ

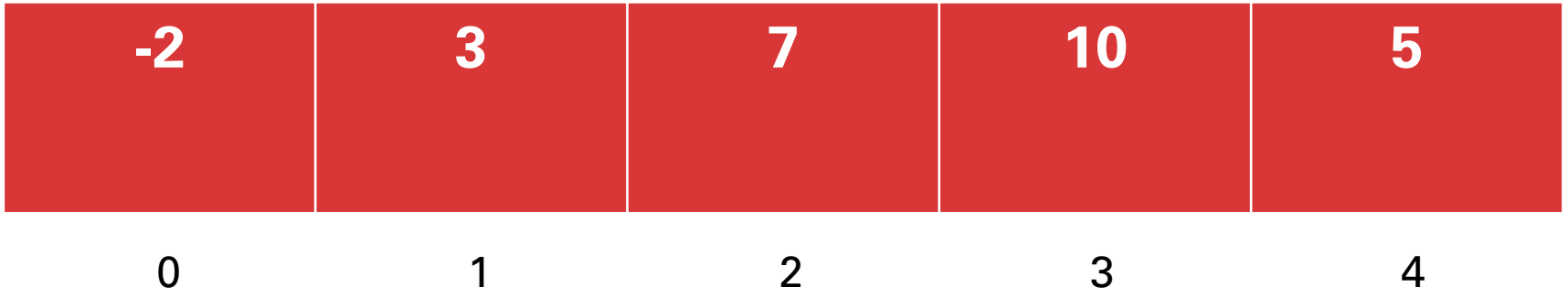
- Let's talk about enqueue()!
  - To enqueue an element, first add it to the end of your pqueue!
  - Next, bubble the element up (if a parent exists!) Compare it with its parent at index (i-1)/2. Swap if your element **is less than** its parent! Remember to update your current index to reflect the swap ☺
  - Repeat this process until either your parent is smaller than you, or you're at the top of the heap!

**Done!**



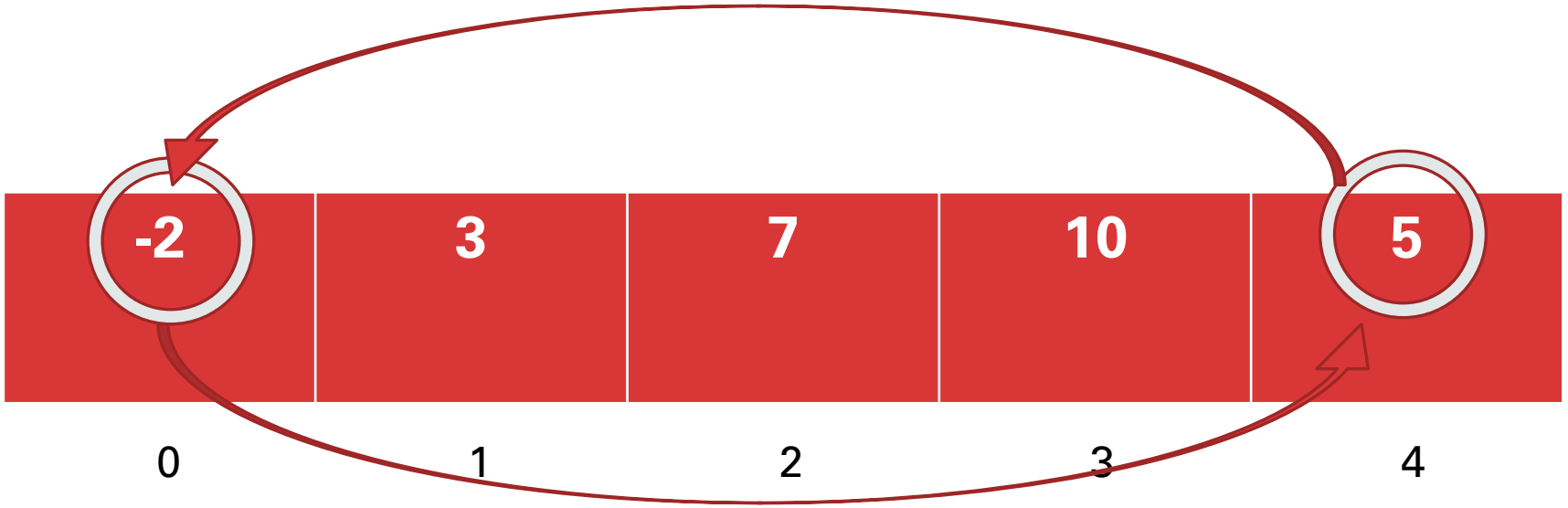
# Part 3: Heap PQ

- Let's talk about dequeue()!
  - To start, **swap** your **first** and **last** elements and reduce your size by 1 (you could also just overwrite root!)



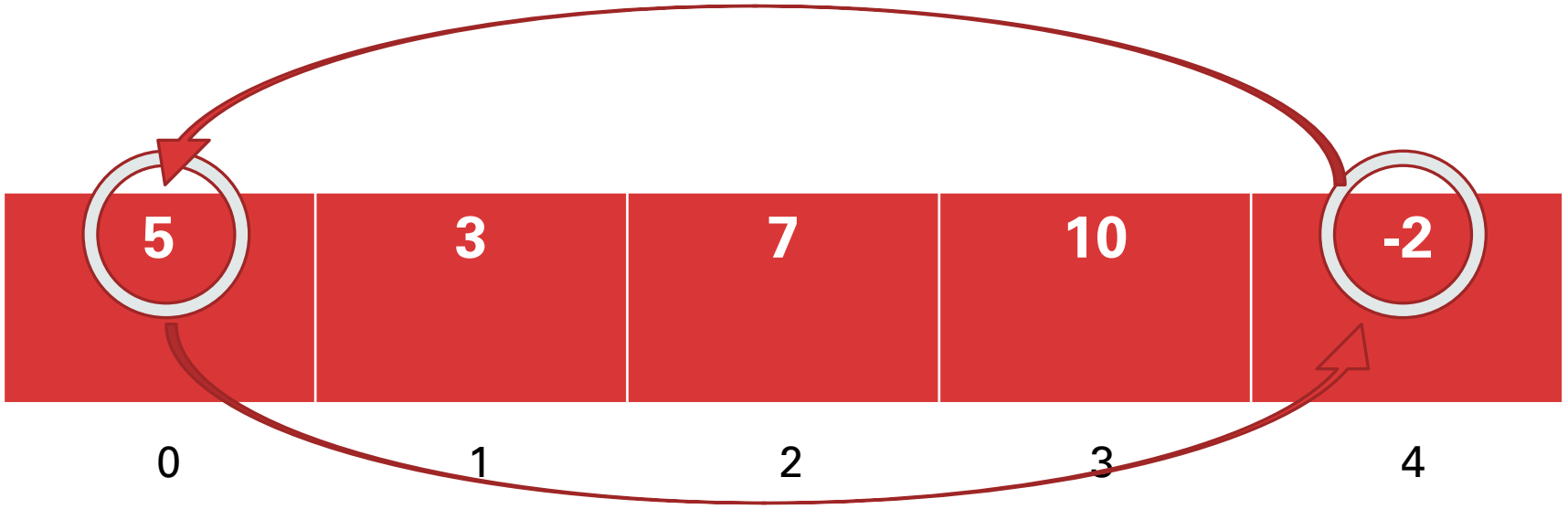
# Part 3: Heap PQ

- Let's talk about dequeue()!
  - To start, **swap** your **first** and **last** elements and reduce your size by 1 (you could also just overwrite root!)



# Part 3: Heap PQ

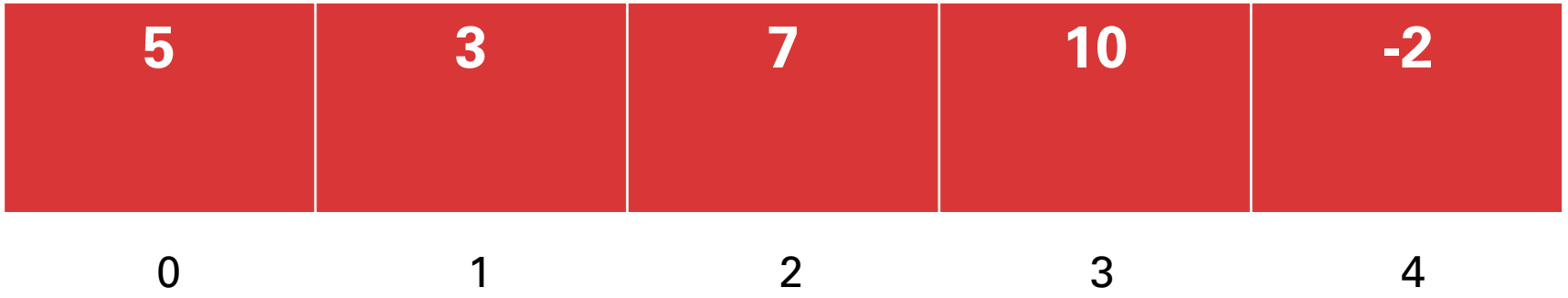
- Let's talk about dequeue()!
  - To start, **swap** your **first** and **last** elements and reduce your size by 1 (you could also just overwrite root!)



# Part 3: Heap PQ

- Let's talk about dequeue()!
  - To start, **swap** your **first** and **last** elements and reduce your size by 1 (you could also just overwrite root!)

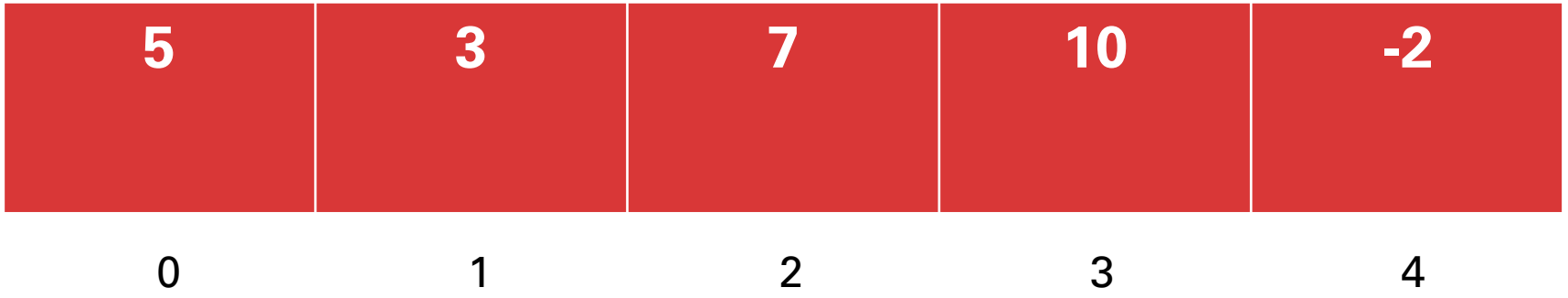
**pq.size() = 4**
**NOT**
**5**



# Part 3: Heap PQ

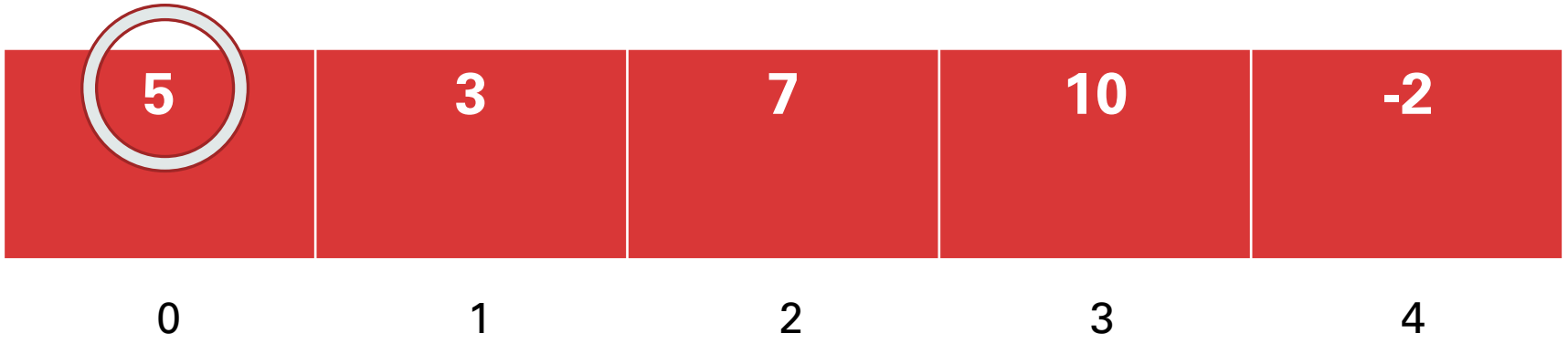
- Let's talk about dequeue()!
  - To start, **swap** your **first** and **last** elements and reduce your size by 1 (you could also just overwrite root!)
  - Next, you want to **bubble down** the root element to its correct place. Compare it with its children, who live at indices (2 * i + 1) and (2 * i + 2), and swap your element with the **smaller** of the children.

**pq.size() = 4**



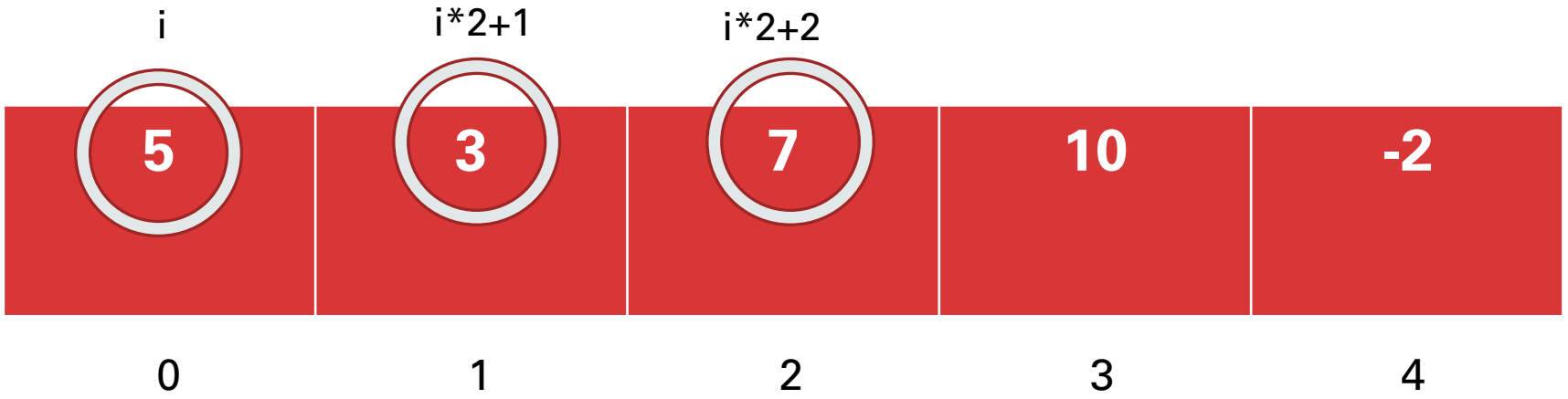
# Part 3: Heap PQ

- Let's talk about dequeue()!
  - To start, **swap** your **first** and **last** elements and reduce your size by 1 (you could also just overwrite root!)
  - Next, you want to **bubble down** the root element to its correct place. Compare it with its children, who live at indices (2 * i + 1) and (2 * i + 2), and swap your element with the **smaller** of the children.



# Part 3: Heap PQ

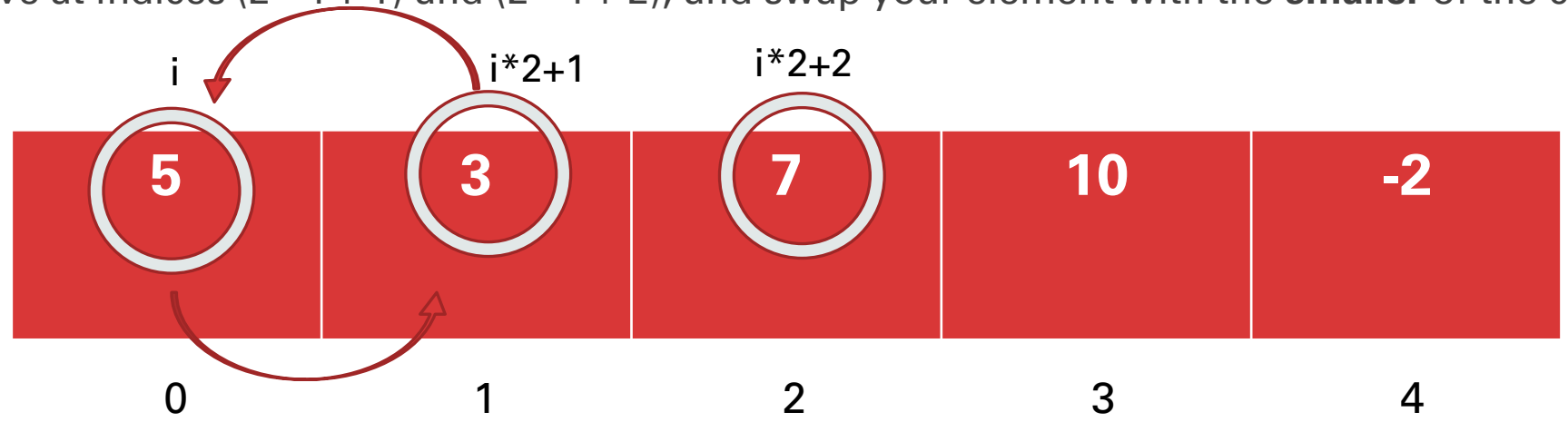
- Let's talk about dequeue()!
  - To start, **swap** your **first** and **last** elements and reduce your size by 1 (you could also just overwrite root!)
  - Next, you want to **bubble down** the root element to its correct place. Compare it with its children, who live at indices (2 * i + 1) and (2 * i + 2), and swap your element with the **smaller** of the children.




Disclaimer: I'm just using 'i' to represent the index of the element we're bubbling down; it has nothing to do with for loops ☺

# Part 3: Heap PQ

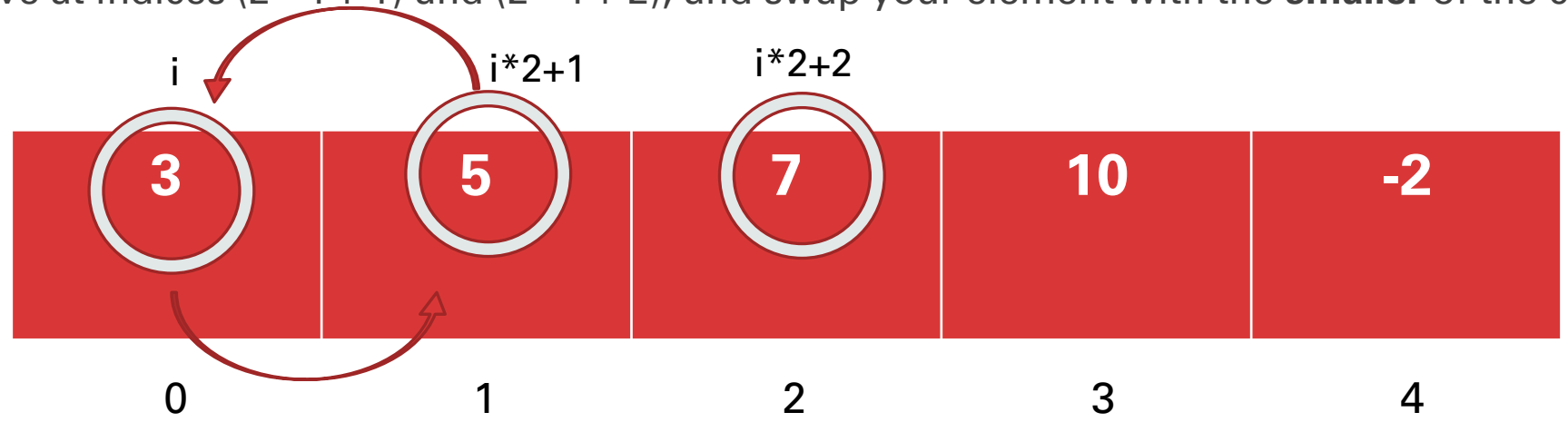
- Let's talk about dequeue()!
  - To start, **swap** your **first** and **last** elements and reduce your size by 1 (you could also just overwrite root!)
  - Next, you want to **bubble down** the root element to its correct place. Compare it with its children, who live at indices (2 * i + 1) and (2 * i + 2), and swap your element with the **smaller** of the children.




Our friend the face is back!

# Part 3: Heap PQ

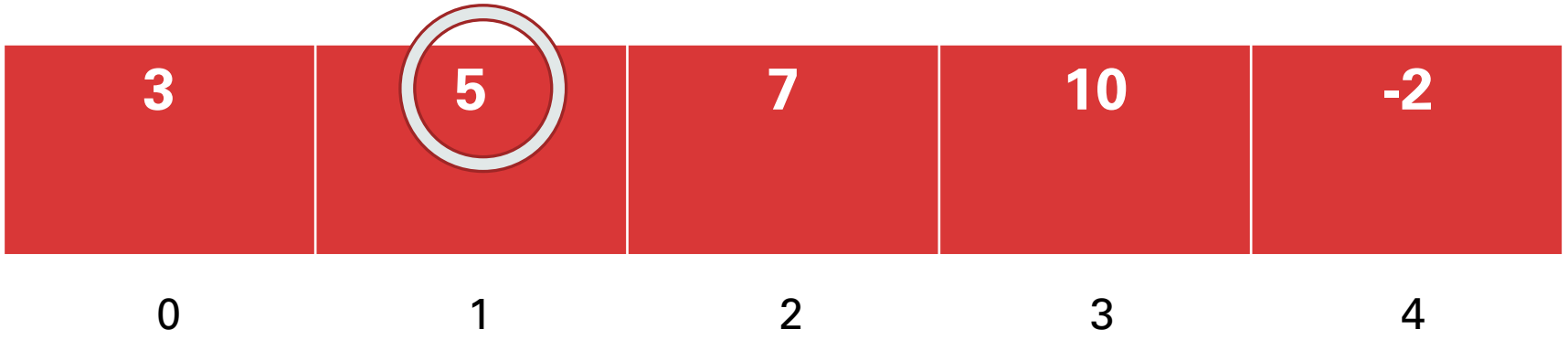
- Let's talk about dequeue()!
  - To start, **swap** your **first** and **last** elements and reduce your size by 1 (you could also just overwrite root!)
  - Next, you want to **bubble down** the root element to its correct place. Compare it with its children, who live at indices (2 * i + 1) and (2 * i + 2), and swap your element with the **smaller** of the children.




Our friend the face is back!

# Part 3: Heap PQ

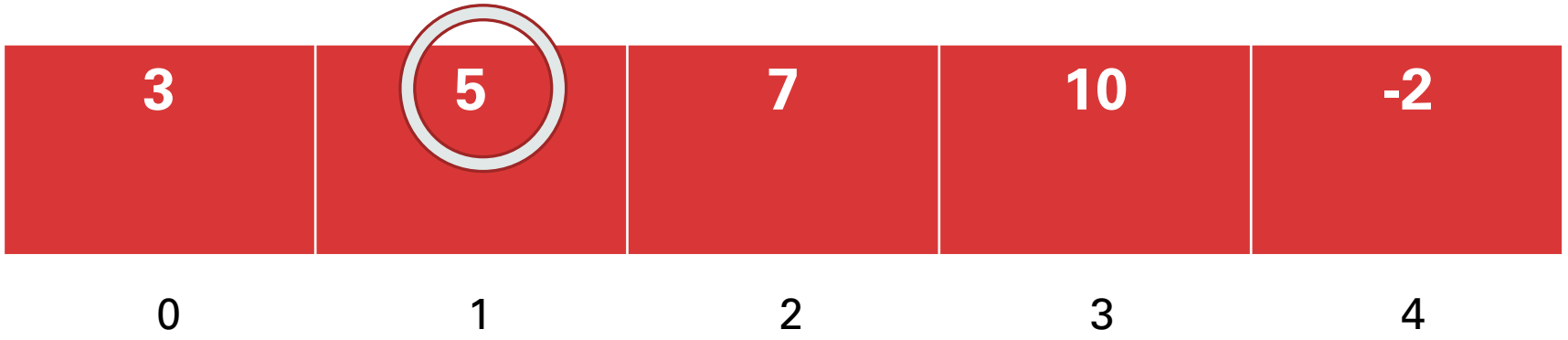
- Let's talk about dequeue()!
  - To start, **swap** your **first** and **last** elements and reduce your size by 1 (you could also just overwrite root!)
  - Next, you want to **bubble down** the root element to its correct place. Compare it with its children, who live at indices (2 * i + 1) and (2 * i + 2), and swap your element with the **smaller** of the children.




Our friend the face is back!

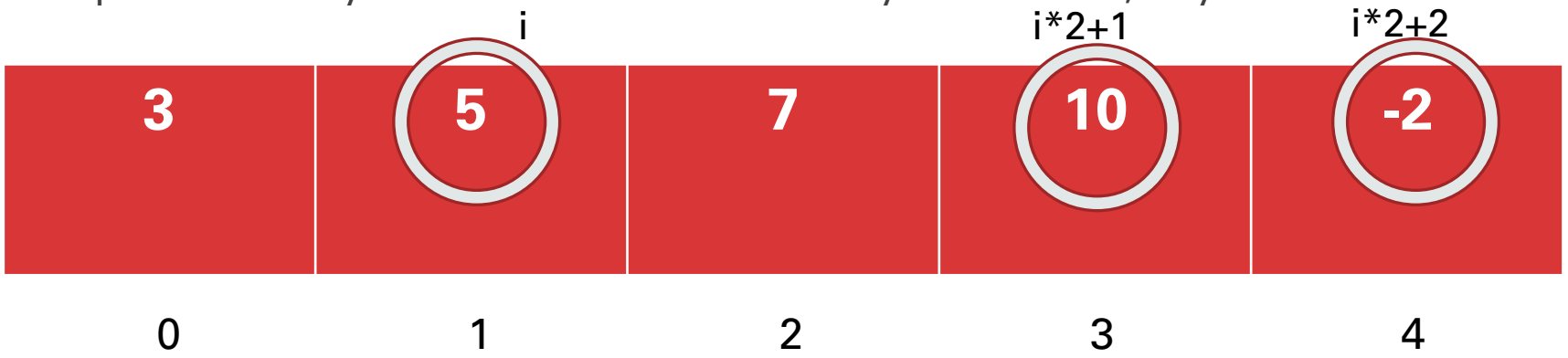
# Part 3: Heap PQ

- Let's talk about dequeue()!
  - To start, **swap** your **first** and **last** elements and reduce your size by 1 (you could also just overwrite root!)
  - Next, you want to **bubble down** the root element to its correct place. Compare it with its children, who live at indices (2 * i + 1) and (2 * i + 2), and swap your element with the **smaller** of the children.
  - Repeat this process until you are smaller than **both** of your children, or you have no more children left!



# Part 3: Heap PQ

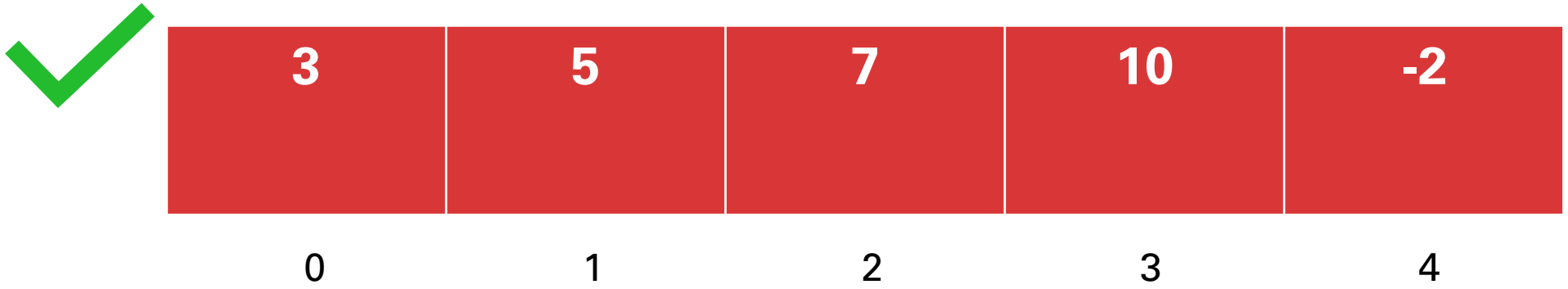
- Let's talk about dequeue()!
  - To start, **swap** your **first** and **last** elements and reduce your size by 1 (you could also just overwrite root!)
  - Next, you want to **bubble down** the root element to its correct place. Compare it with its children, who live at indices (2 * i + 1) and (2 * i + 2), and swap your element with the **smaller** of the children.
  - Repeat this process until you are smaller than **both** of your children, or you have no more children left!



# Part 3: Heap PQ

- Let's talk about dequeue()!
  - To start, **swap** your **first** and **last** elements and reduce your size by 1 (you could also just overwrite root!)
  - Next, you want to **bubble down** the root element to its correct place. Compare it with its children, who live at indices (2 * i + 1) and (2 * i + 2), and swap your element with the **smaller** of the children.
  - Repeat this process until you are smaller than **both** of your children, or you have no more children left!

**Done!**




Why can't we swap with -2?

# Part 3: Heap PQ

Helpful hints:

- I recommend writing a **swap**() method and **bubbleUp()** and **bubbleDown()** methods.
- **dequeue**() is a little more heap-y than **enqueue()**, so I'd recommend doing **enqueue()** first to get your feet wet!
- Don't worry too much about ties – swapping identical elements effectively does nothing.
- The debugger and validateInternalState() can be life-savers here!
- Notice that validateInternalState() might be trickier to write here – you now have to verify that your state is a correct heap state, not a sorted array state…

# Part 3: Heap PQ

One particular edge case I want to point out:

- In **dequeue()**, be cognizant of the fact that it's possible to **only have one child** within the bounds of the array!
  - In this case, the second child should be ignored. If you don't check for this, your bubble down will read in a potentially bogus value that can cause wacky behavior in your program.

# Questions about Part 3?