YEAH Hours A5

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 and LIFO enthusiast, congratulating on a job well done so far!

An informal announcement…

- The deadline to apply to become a section leader for current 106B students is **today! (10/23)**
- It's an amazing Stanford job!
  - Incredible community + network
  - You get paid!
  - Give back to the program that supported you!
  - Amazing events with lecturers / tech recruiters!
  - This wonderful program has made me want to teach!!

Assignment logistics

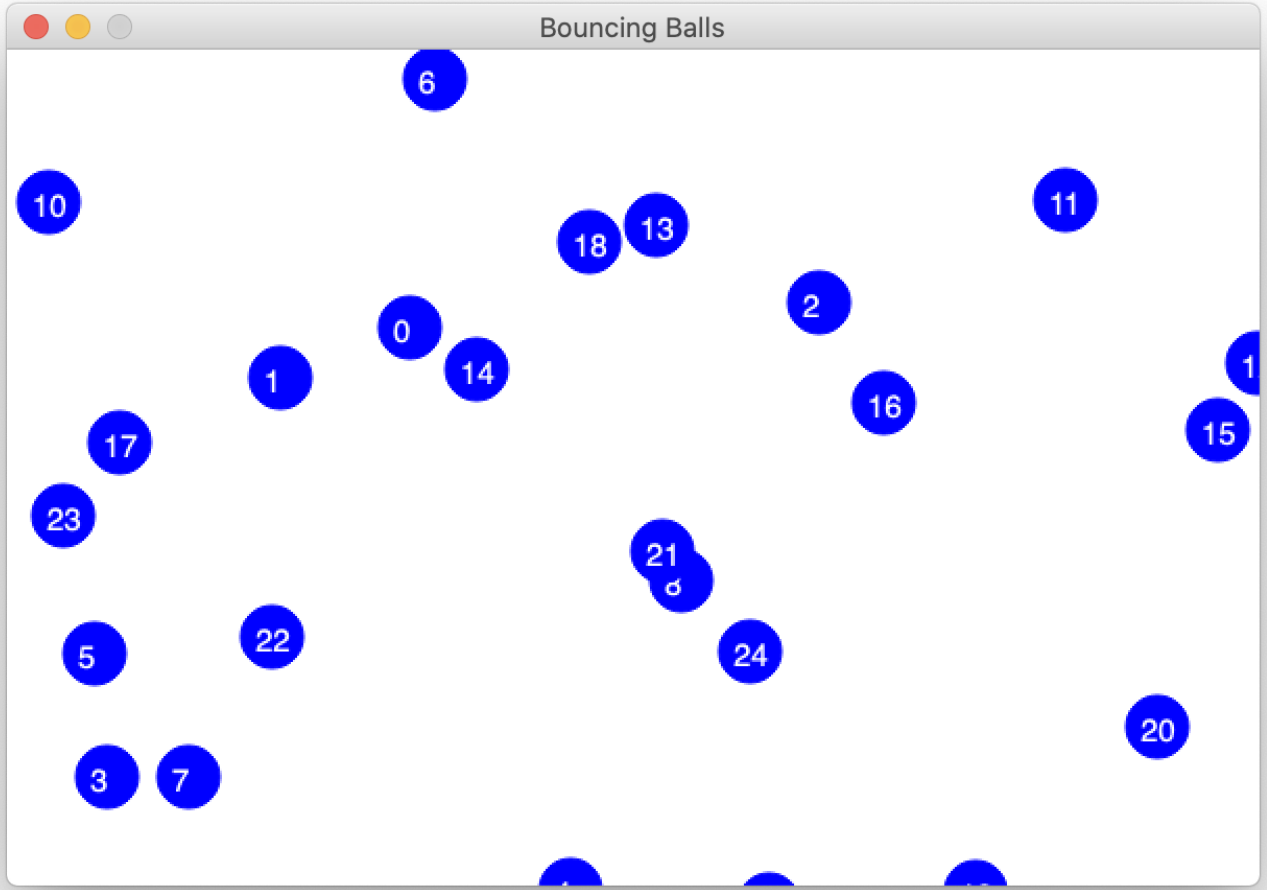
- The assignment is due on **Wednesday October 28th at 11:59PM PDT**
  - The grace period for submission expires **Friday, October 30th** at the same time.
- Try and start early! This one can be tricky to debug if you're not careful!

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: PQ Client Tasks – Using a **priorityqueue**, what kinds of powerful things can you do?
4. Part 3: Heap PQ – Implement a **priorityqueue** using a binary min-heap!
5. Part 4: Data Demos – You don't have to do any work here – watch some incredible graphics demos that showcase your hard work!

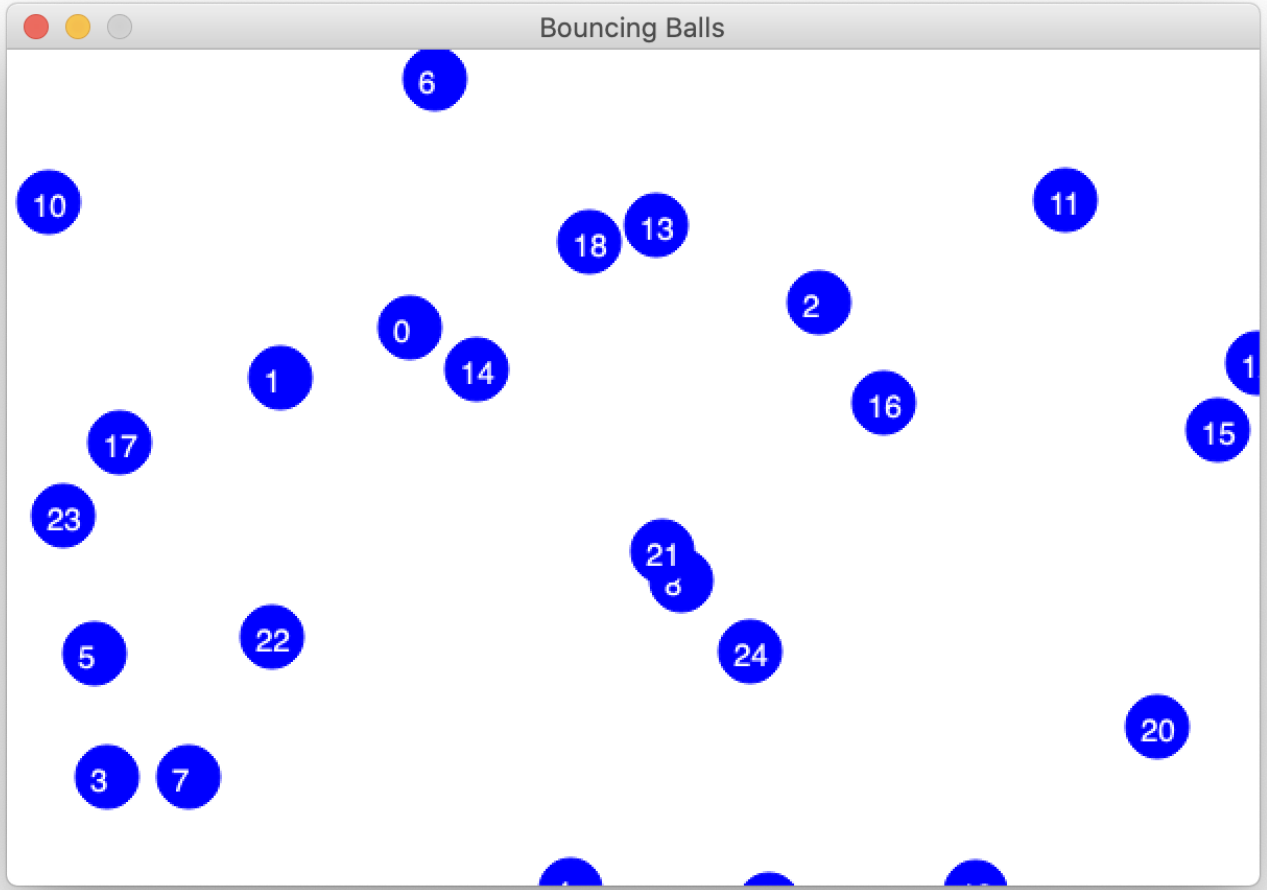
Warmup Debrief

- In this week's warmups, you'll examine a **bouncing balls** program to learn about debugging objects.
- In the above program, a number of balls are rendered on screen, and they move randomly around the screen.
  - Unfortunately, there's a rather conspicuous bug in the program. We want you to try and figure out what the issue is!



Warmup Debrief

- To debug this program, you're going to need to examine **member variables** in the debugger. Luckily, viewing these members is just like how you viewed program variables before!
- You'll also learn how to set **conditional breakpoints**, which are breakpoints that only trigger when the program is at a pre-defined state.



Warmup Debrief

- For the next part of the warmup, you'll be debugging various functions that operate on c++ arrays.
- In this assignment, array 'elements' will be defined by the struct to the right. Structs are like *lightweight* objects!

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

Warmup Debrief

- There are 4 memory error cases that you'll observe:

  1. Dereferencing a nullptr address

  2. Accessing an index outside the allocated array bounds

  3. Accessing memory after it has been deallocated

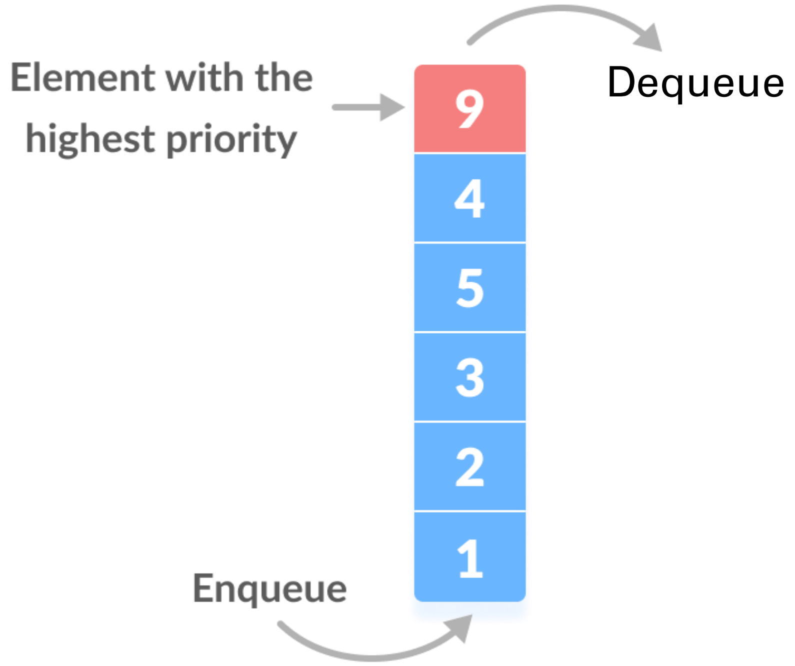
  4. Deallocating the same memory twice

- What do you think will happen in each of these cases?

```
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 structs**, that have embedded priorities
  - 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 ☺.



A “max” priotiy queue of **integers**. 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 DataPoints**, much like you've seen in lecture and section examples!
  - In this particular array, **all elements are sorted from high to low priority (front to back), and the smallest priority element will be dequeue()'d first!**
- How does this queue work?

> **Important Note:** The `priority` field is an integer value. **A smaller integer value indicates a more urgent priority than a larger integer value.** The DataPoint with the minimum integer value for priority is treated as the most urgent and is the one retrieved by peek/dequeue.

So important, I'll say it twice!

Let's see an example…

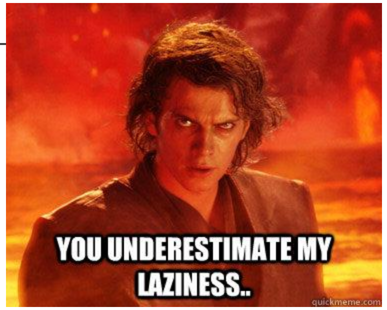
- To simplify these examples, let's just assume that my **priority queue** stores integers that represent their own priority!

Let’s see an example…

- To simplify these examples, let's just assume that my **priority queue** stores integers that represent their own priority!
- Whereas **you** will need to enqueue **DataPoint** structs and extract the priority!

Let's see an example…

- To simplify these examples, let's just assume that my **priority queue** stores integers that represent their own priority!
- Whereas **you** will need to enqueue **DataPoint** structs and extract the priority!



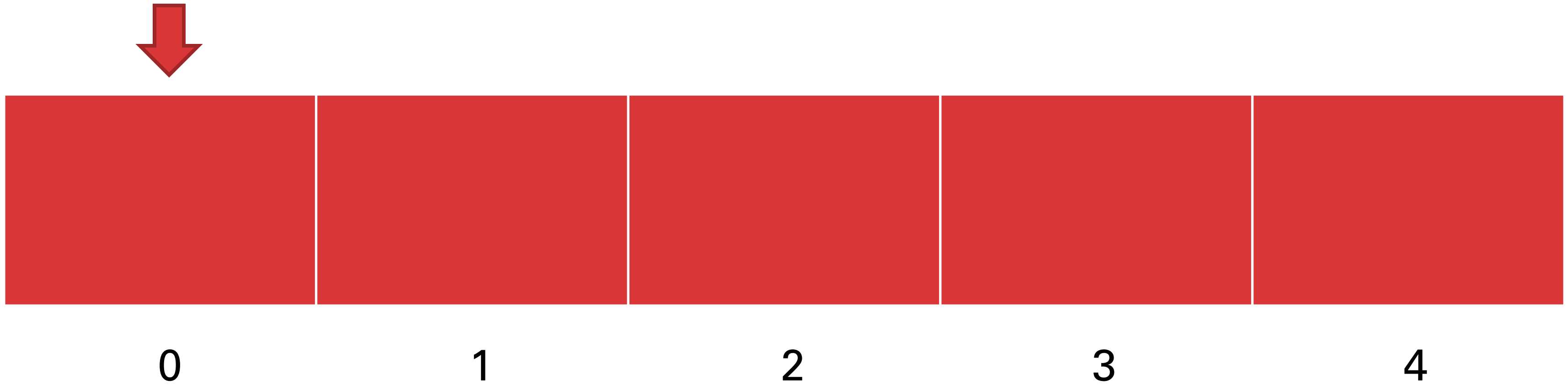
0 1 2 3 4

Let's see an example…

PQ.enqueue(10);

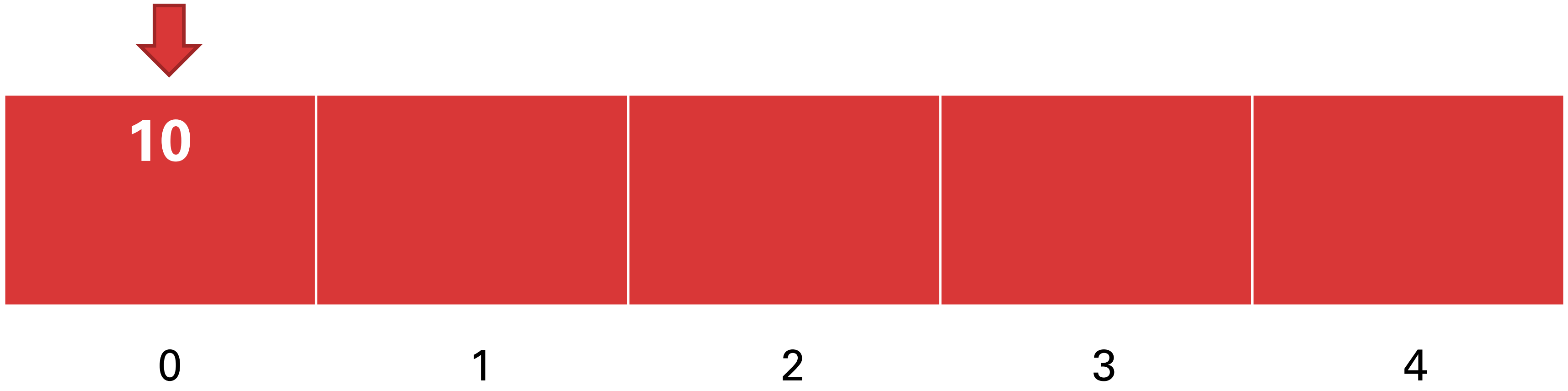
Let's see an example…

PQ.enqueue(10);



Let's see an example…

PQ.enqueue(10);



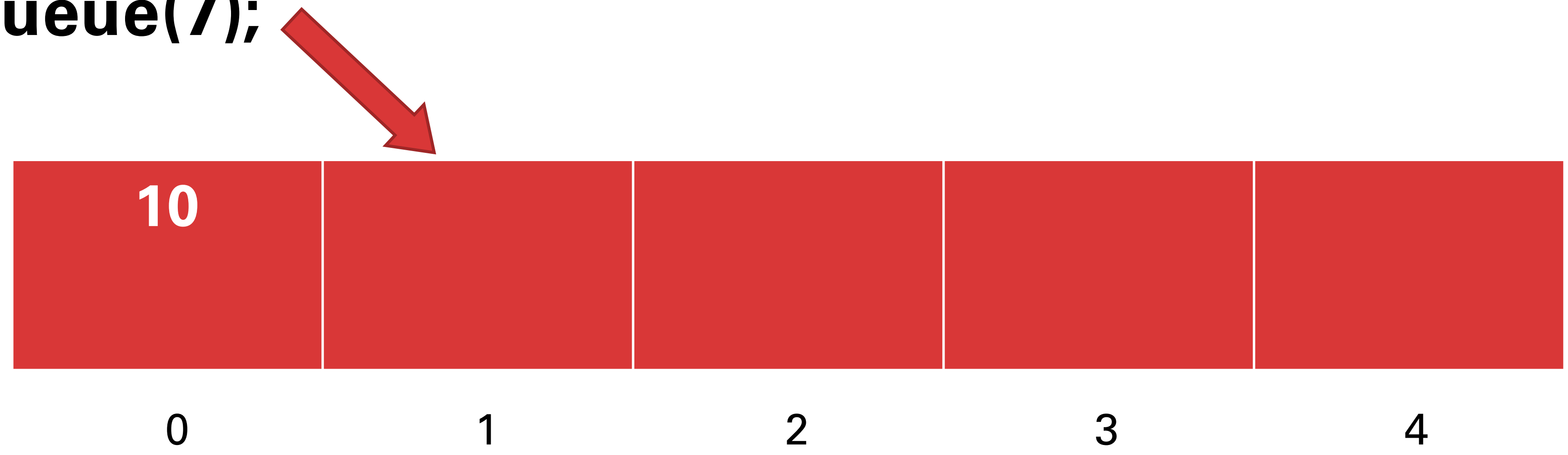
Let's see an example…

PQ.enqueue(7);

0	1	2	3	4
10				

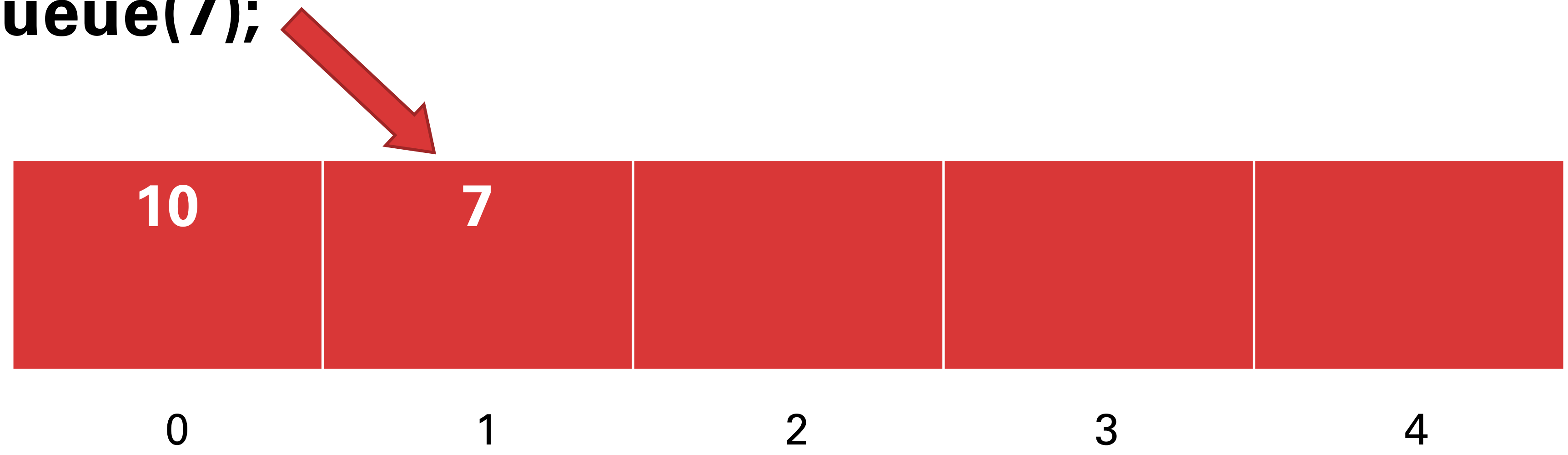
Let's see an example…

PQ.enqueue(7);



Let's see an example…

PQ.enqueue(7);



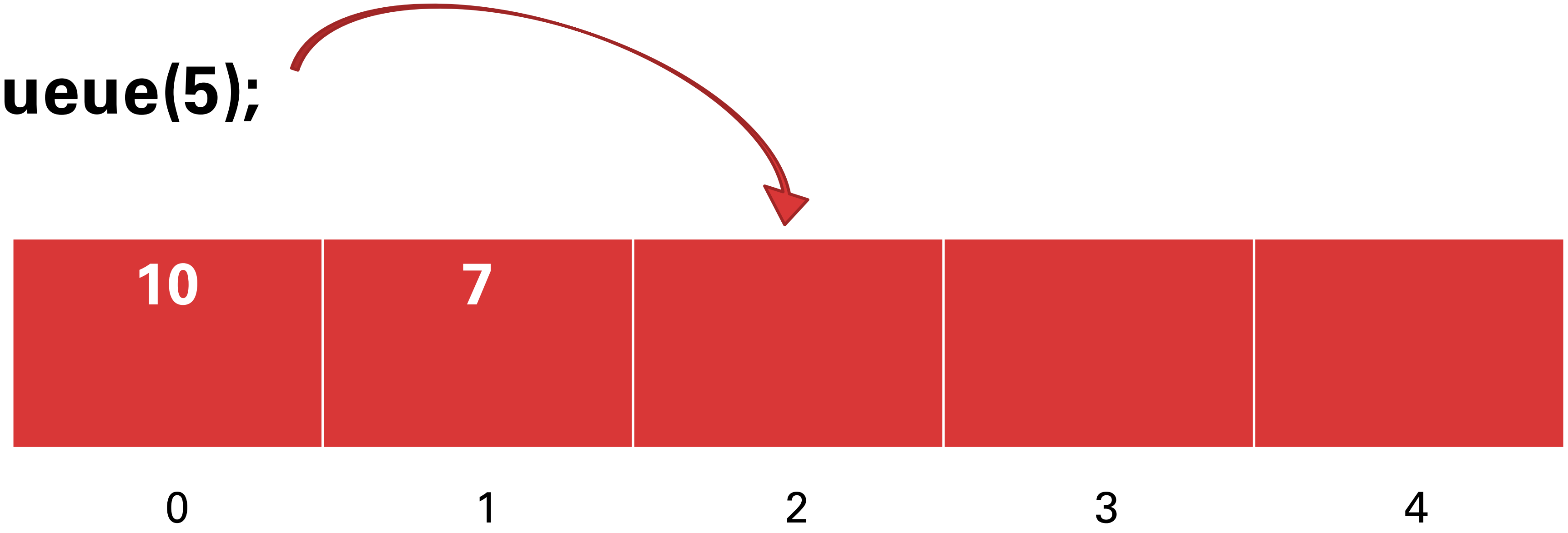
Let's see an example…

PQ.enqueue(5);

10	7			
0	1	2	3	4

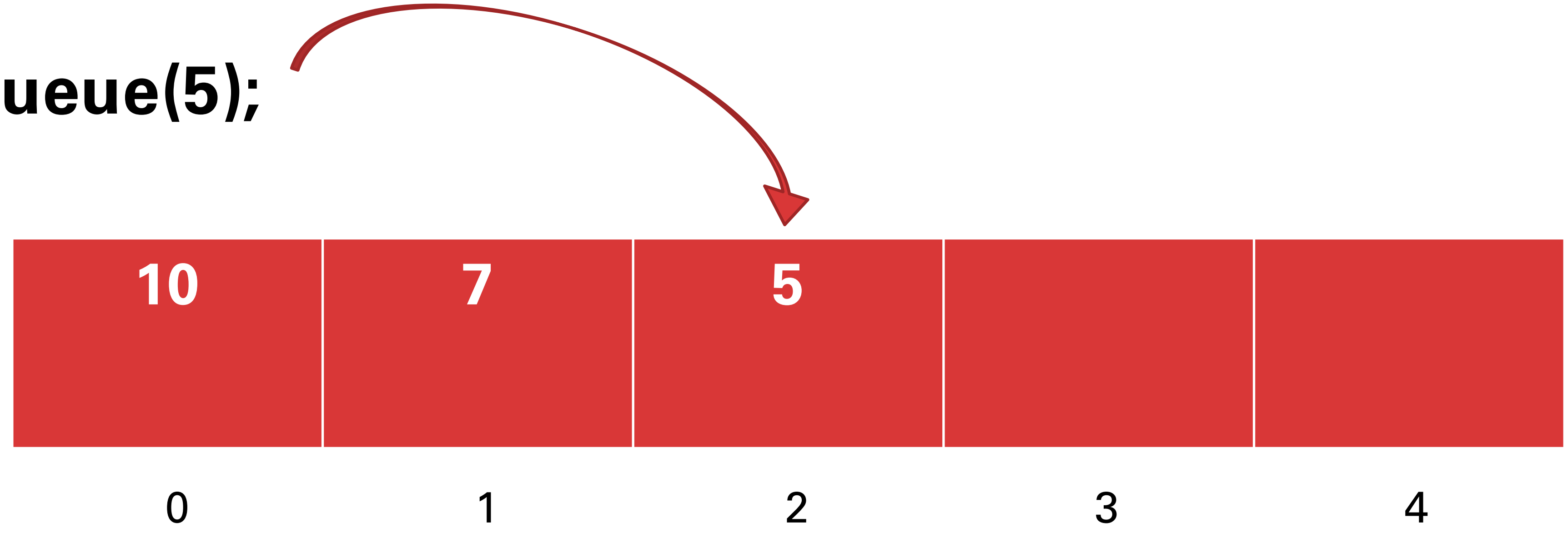
Let’s see an example…

PQ.enqueue(5);



Let's see an example…

PQ.enqueue(5);



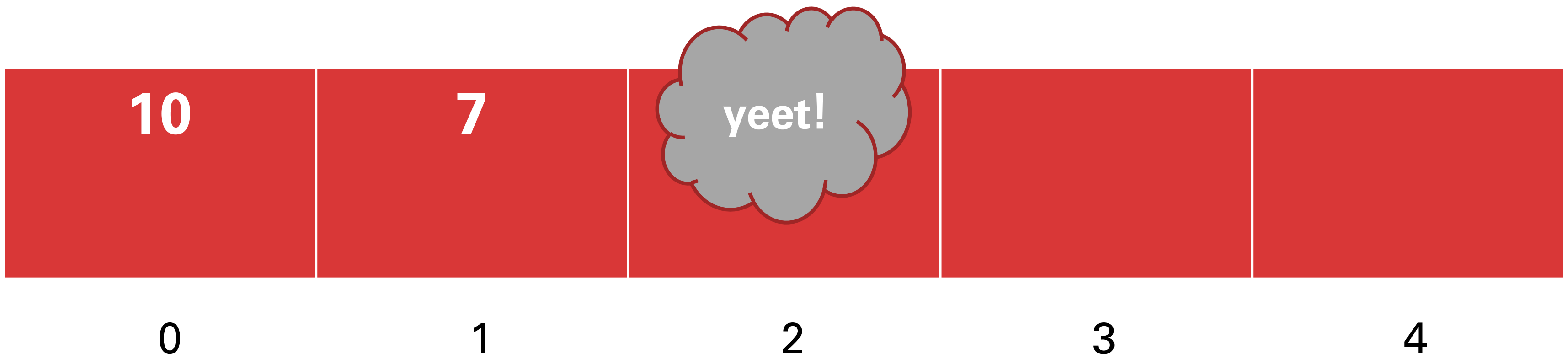
Let’s see an example…

PQ.dequeue();

10	7	5		
0	1	2	3	4

Let's see an example…

PQ.dequeue();



Let's see an example…

PQ.dequeue();

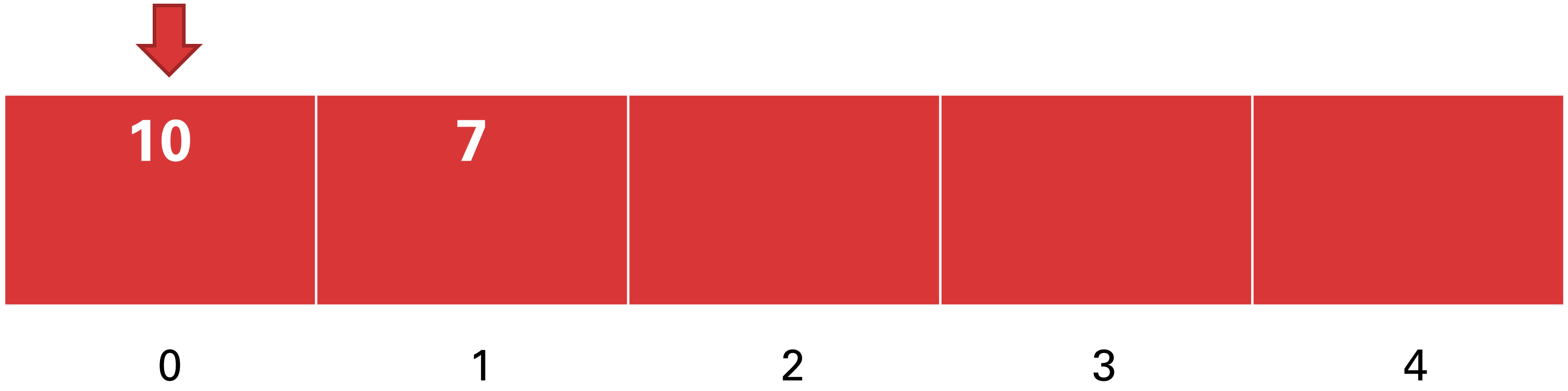
10	7			
0	1	2	3	4

Let's see an example…

PQ.enqueue(20);

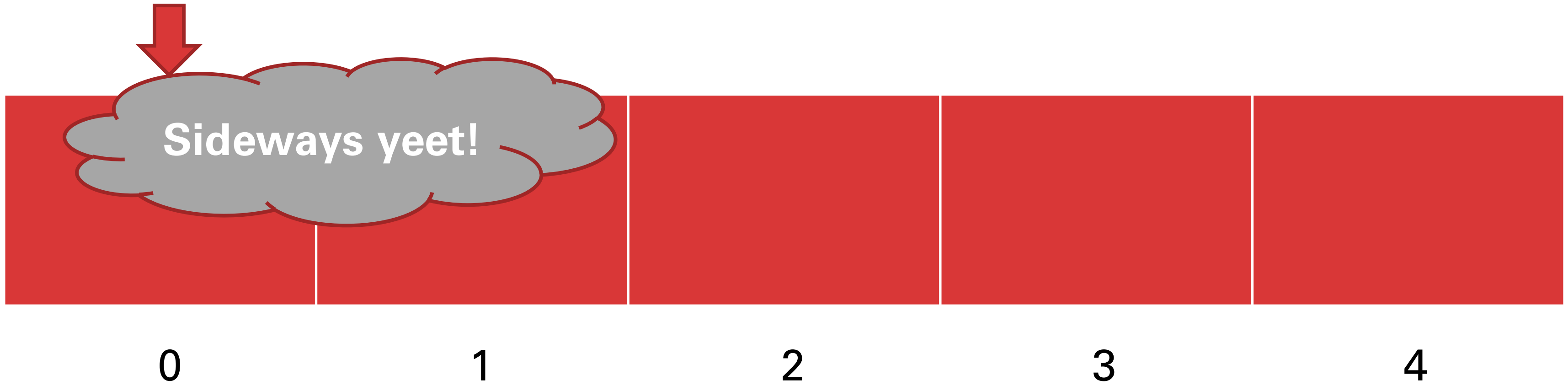
Let's see an example…

PQ.enqueue(20);



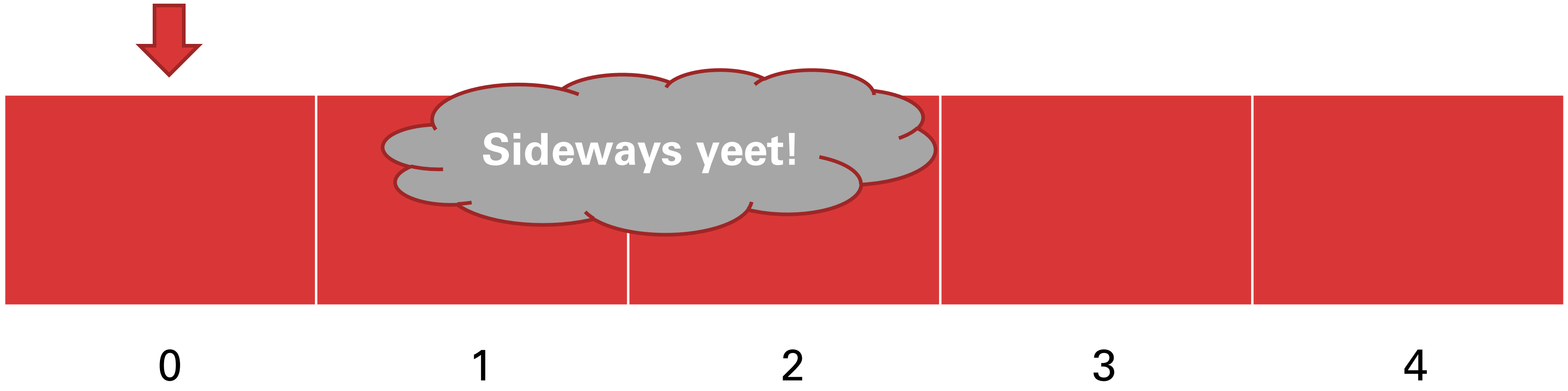
Let's see an example…

PQ.enqueue(20);



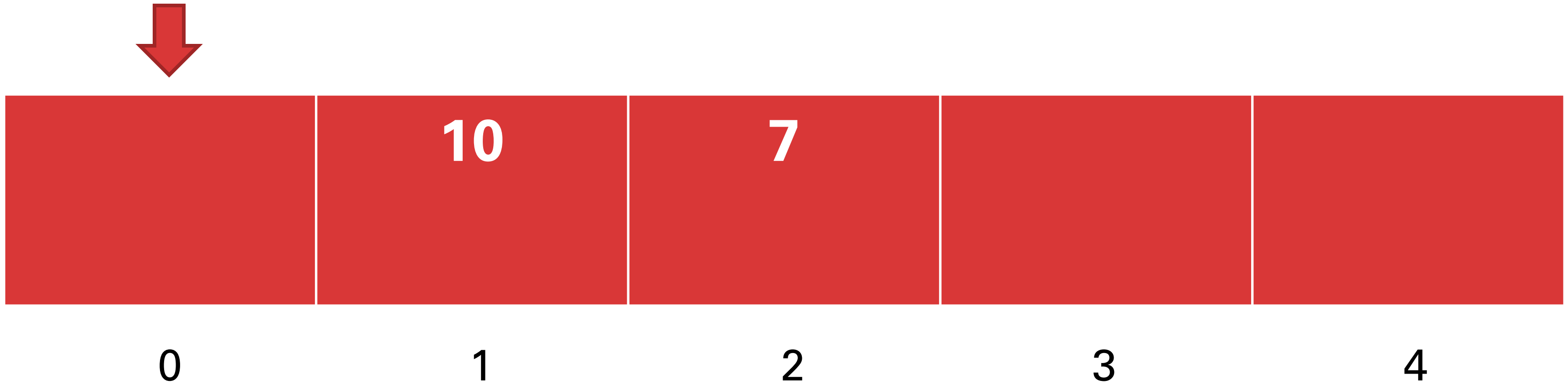
Let’s see an example…

PQ.enqueue(20);



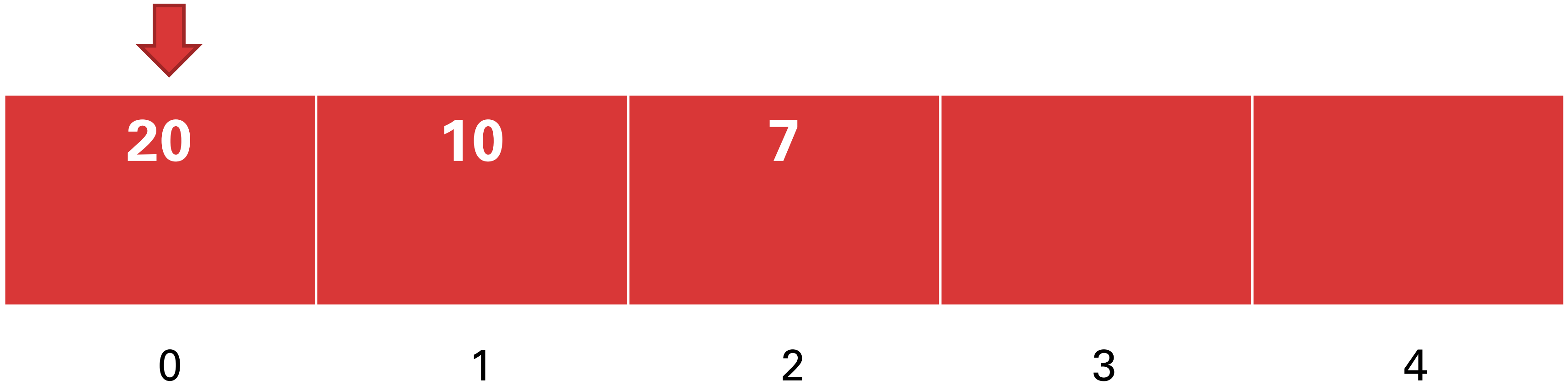
Let's see an example…

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

Helpful hints:

- You might want to make the resize() method a private helper method – it makes for a cleaner implementation.
- Apart from enqueue(), **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 from section!

Part 1: PQ Sorted Array

Debugging advice:

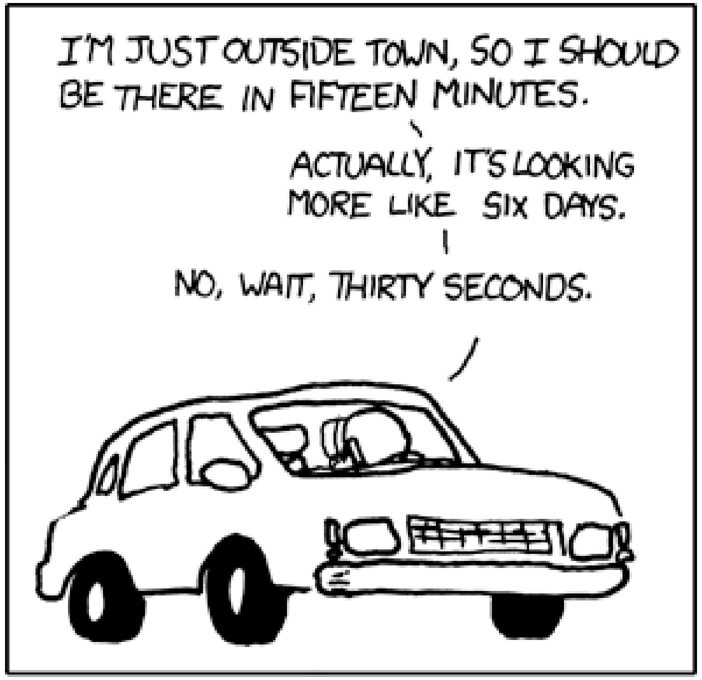
- Debugging this assignment is a little different than debugging others, because you can't interact with the internals of your PQSortedArray when you're testing it be default
  - Verify to yourself that you shouldn't be able to access the object's internal state when debugging!
- To get around this, we have given you a function called **validateInternalState()**, which goes through the values in your underlying array to ensure that everything is in proper sorted order; else it throws an error.
- An additional debugging function we provide you is **printDebugInfo()**, which prints out the contents of your array, if you prefer a more hands-on approach to debugging.
- Both of these functions are **public** member functions, so you can call them in your student tests!

Part 1: PQ Sorted Array

More Debugging advice:

- Think about where good places to call **validateInternalState()** or **printDebugInfo**() might be!
- We also encourage that you use the debugger, as a way to easily see all of your data at runtime!
- Be very careful about your array indexing – out of bounds errors are common here! Perhaps a helper function verifying that an index is in bounds would be helpful
- Also be mindful of your use of **delete[]**. There should be a single **delete[]** for every invocation of a **new** keyword!

Questions about Part 1?



THE AUTHOR OF THE WINDOWS FILE COPY DIALOG VISITS SOME FRIENDS.

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.
- *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. Store elements back into vector, now in sorted order.
     */
    for (int i = 0; i < v.size(); i++) {
        v[i] = pq.dequeue();
    }
}
```

Part 2: Client Tasks

- You'll be implementing the function **Vector**<DataPoint> **topK**(istream& stream, int k);

Part 2: Client Tasks

- 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.**

Part 2: Client Tasks

- You'll be implementing the function **Vector**<Datapoint> **topK**(istream& stream, int k);
- In the above function, your job is harness the power of the PQ in order to return a **Vector**<DataPoint> of the largest *k* elements 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** priority 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, but it's an O(N) operation. Can you do better?

Part 2: Client Tasks

Tips / Tricks

- Here’s how you can loop through every dataPoint in the stream ->
- 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()?
- If the stream contains fewer than *k* elements, simply return those elements in the Vector as you would if there were more than *k* elements in the stream.

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

Questions about Top-K?

Part 3: Heap PQ

- In this final part, you'll be implementing a full priority queue using a binary min heap!

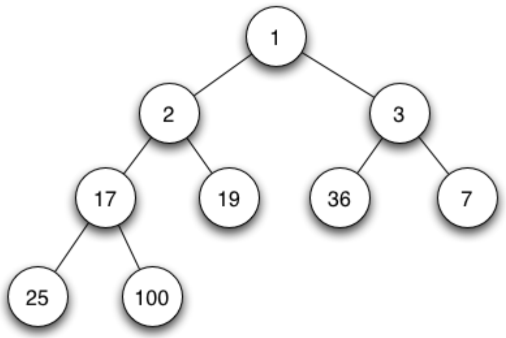
Part 3: Heap PQ

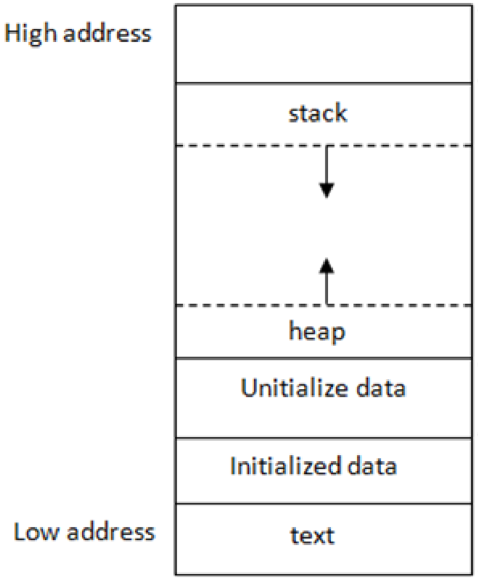
- In this final part, you'll be implementing a full priority queue using a binary min heap!

But Trip, aren't there two things called the heap?

It's time for...

Imogen Heap's data disambiguation!





[Aside] Heap vs. Heap

- Indeed, the **heap data structure** is <u>**completely**</u> different from the **heap region in memory.**
- Moreover, the naming origins don't seem to be linked. A **heap data structure** was conceived and named in the 1960's, whereas the **heap region in memory** was named in the mid 1970's.
- At a high level, both may have been named due to their behaviors. The **heap data structure** is optimized to provide a single element at request (in our case the dequeue()'d element), and the **heap region in memory** is frequently split into blocks that are allocated by requests made by the **new** keyword.
- In this sense, both concepts fundamentally **provide** something to a client on a **per-request basis**, like picking something off a heap of clothes, for example.

Back to the action!

Part 3: Heap PQ

- In this final part, you'll be implementing a full priority queue using a binary min heap!
  - As usual, we mean that the "highest priority" element is the element with the smallest value.
  - In order to keep that property in your queue, you will be using a min heap like you've seen in lecture!
- Lecture 17 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 of how a binary min heap works!

Part 3: Heap PQ

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

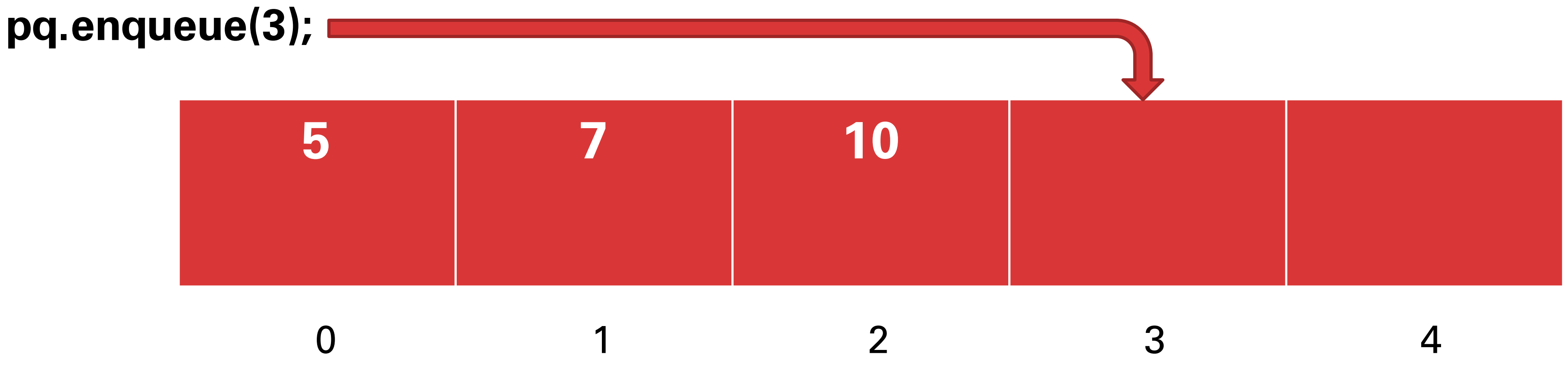
Once again, I'm using **integers** instead of **dataPoints** for clarity

pq.enqueue(3);

5	**7**	**10**		
0	1	2	3	4

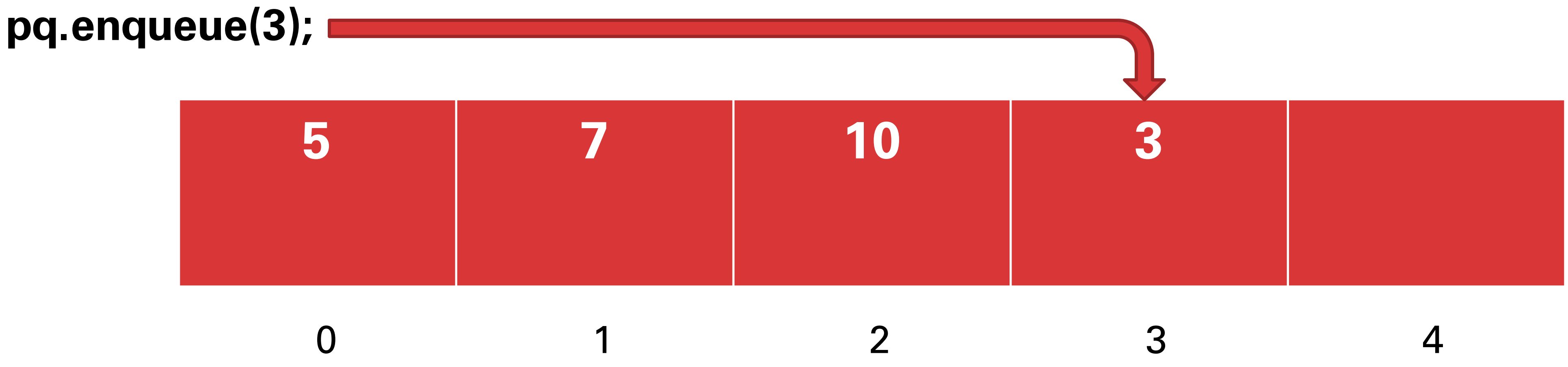
Part 3: Heap PQ

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



Part 3: Heap PQ

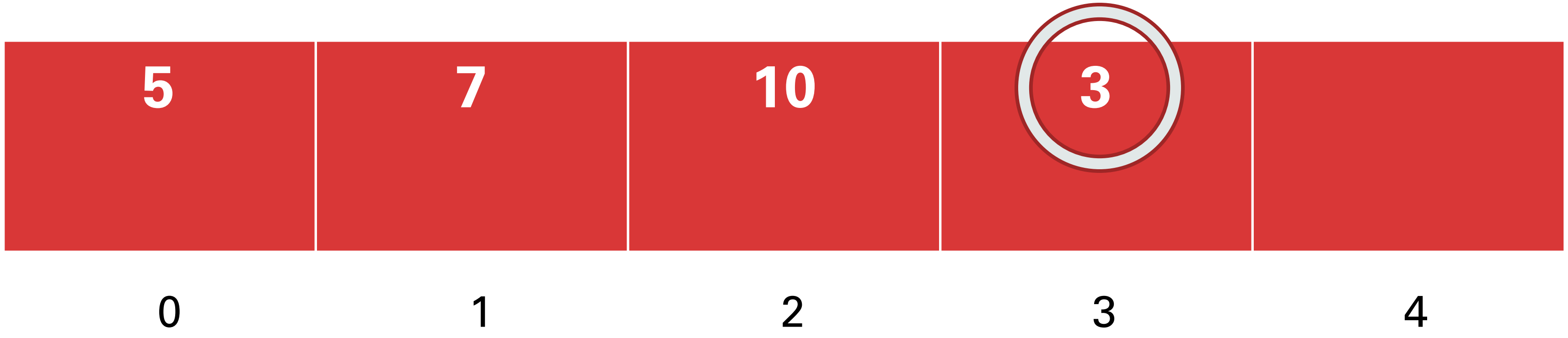
- Let's talk about enqueue()!
  - To enqueue an element, first add it to the end of your pqueue!



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!

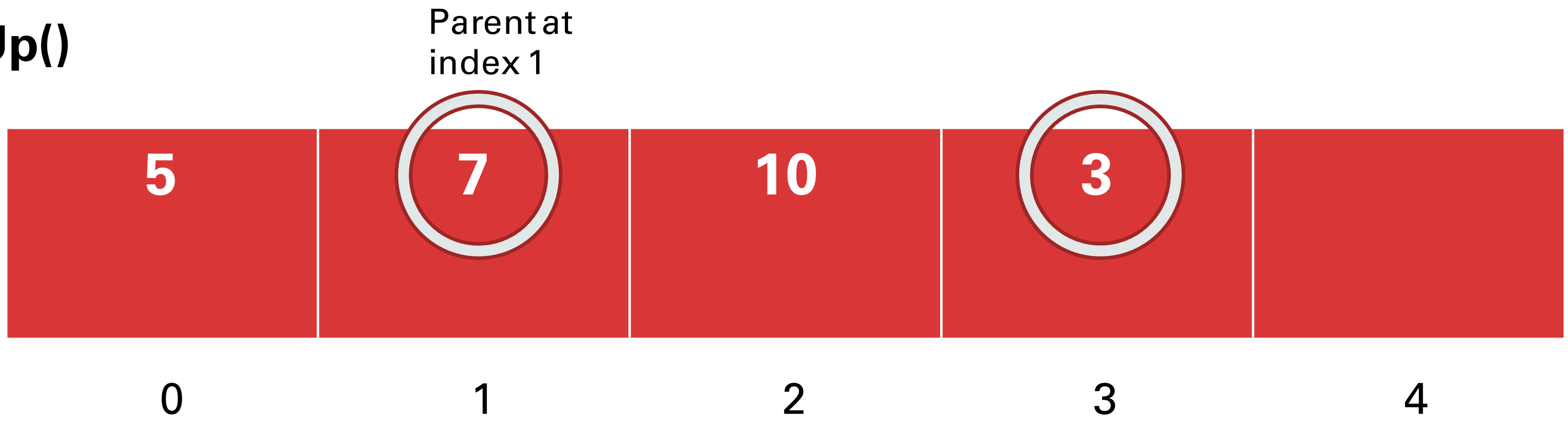
bubbleUp()



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!

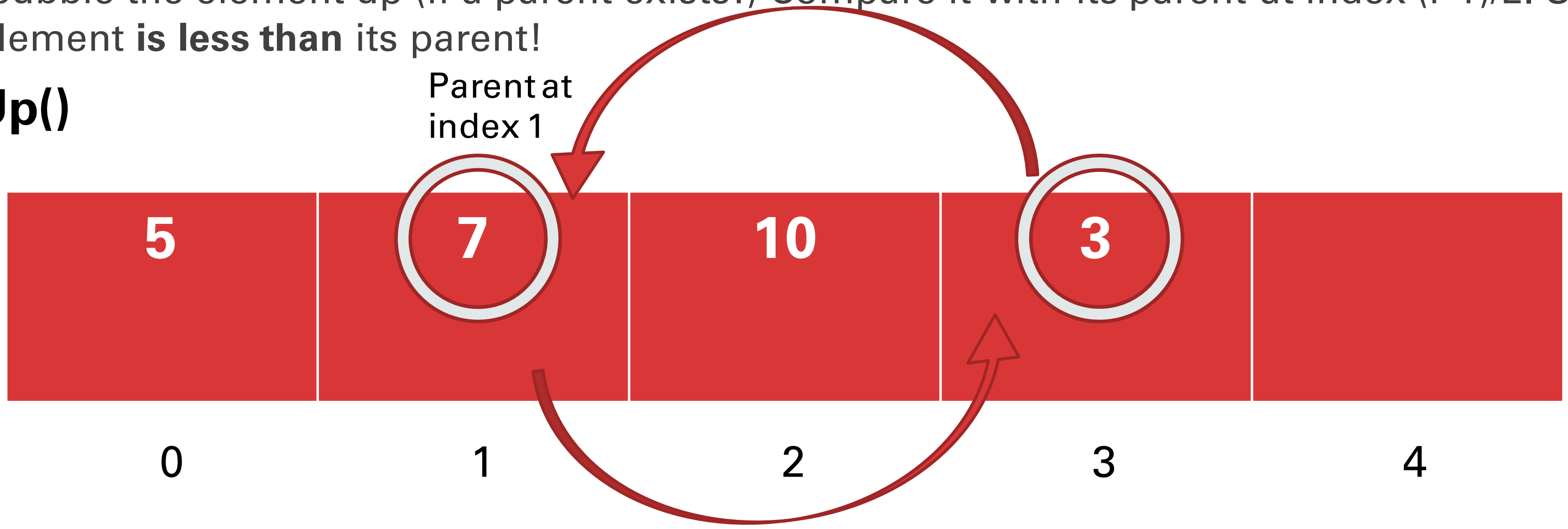
bubbleUp()



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!

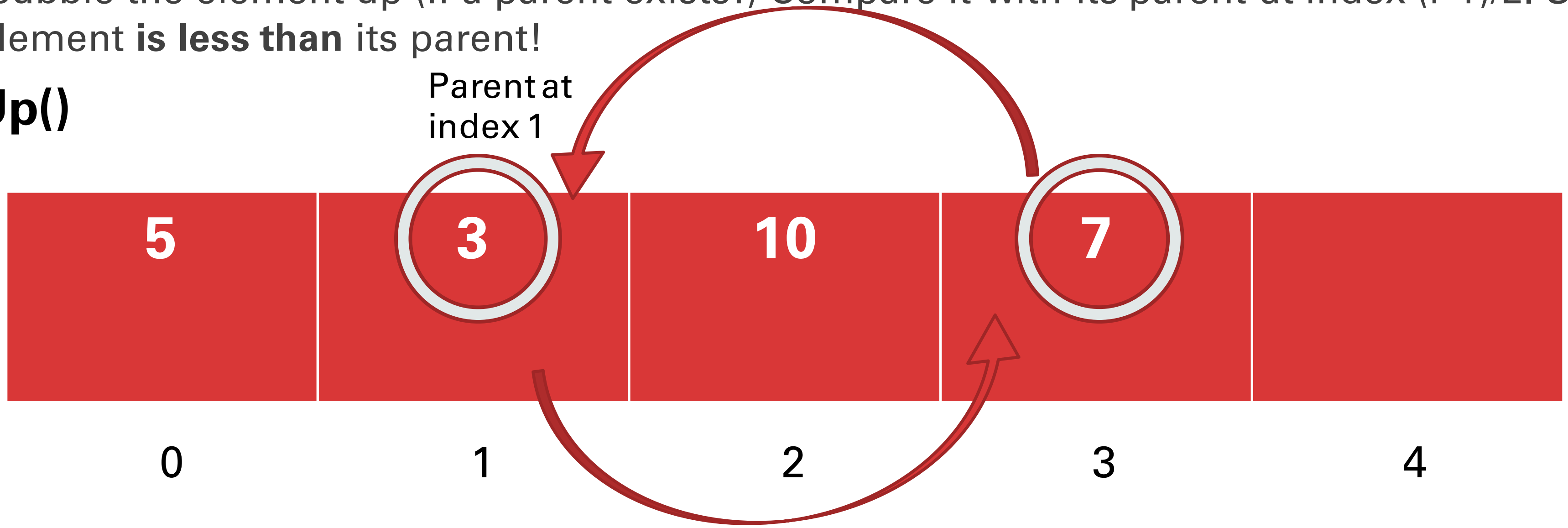
bubbleUp()



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! Be sure to update your element's current index!

bubbleUp()

Index is now 1

5	**3**	**10**	**7**	
0	1	2	3	4

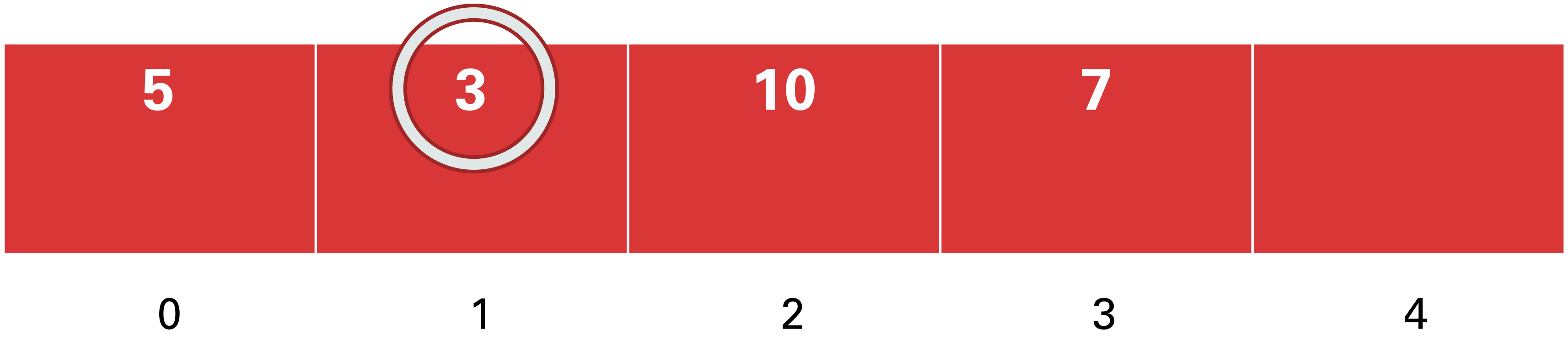
Part 3: Heap PQ

- 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! Be sure to update your element's current index!
  - Repeat this process until either your parent is smaller than you, or you’re at the top of the heap!

5	3	10	7	
0	1	2	3	4

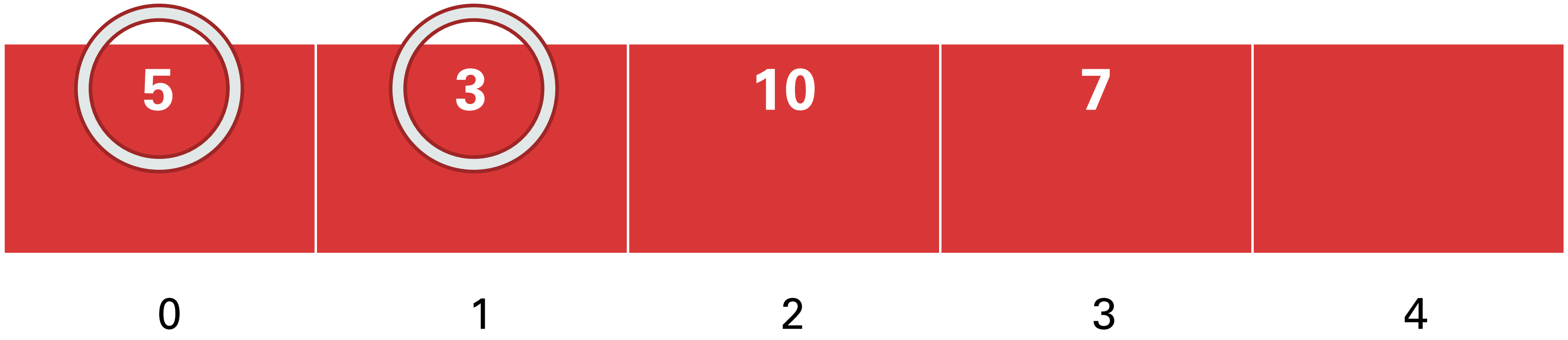
Part 3: Heap PQ

- 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! Be sure to update your element's current index!
  - Repeat this process until either your parent is smaller than you, or you're at the top of the heap!



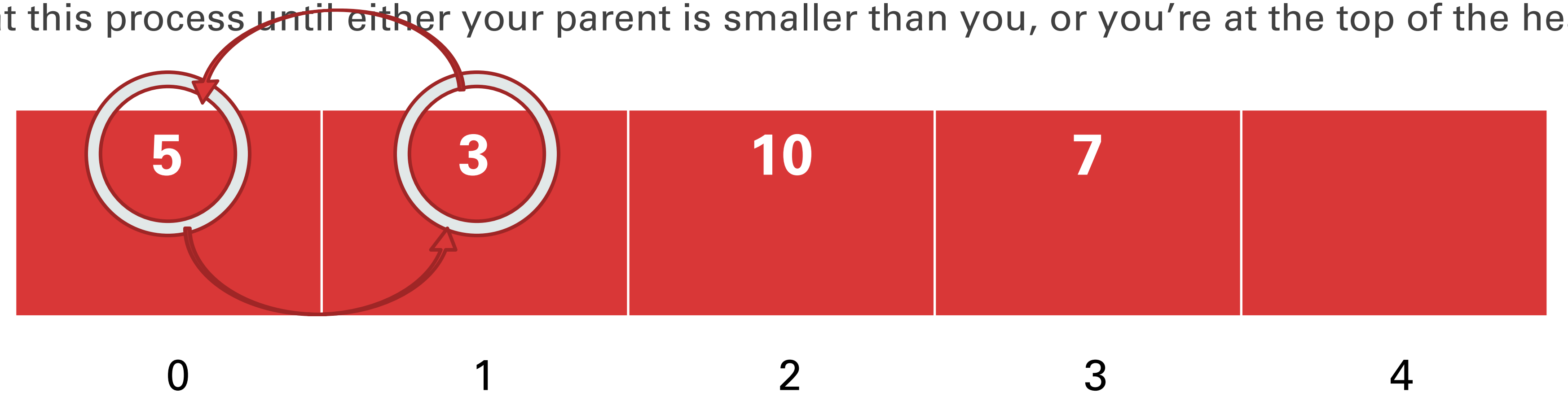
Part 3: Heap PQ

- 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! Be sure to update your element's current index!
  - Repeat this process until either your parent is smaller than you, or you’re at the top of the heap!



Part 3: Heap PQ

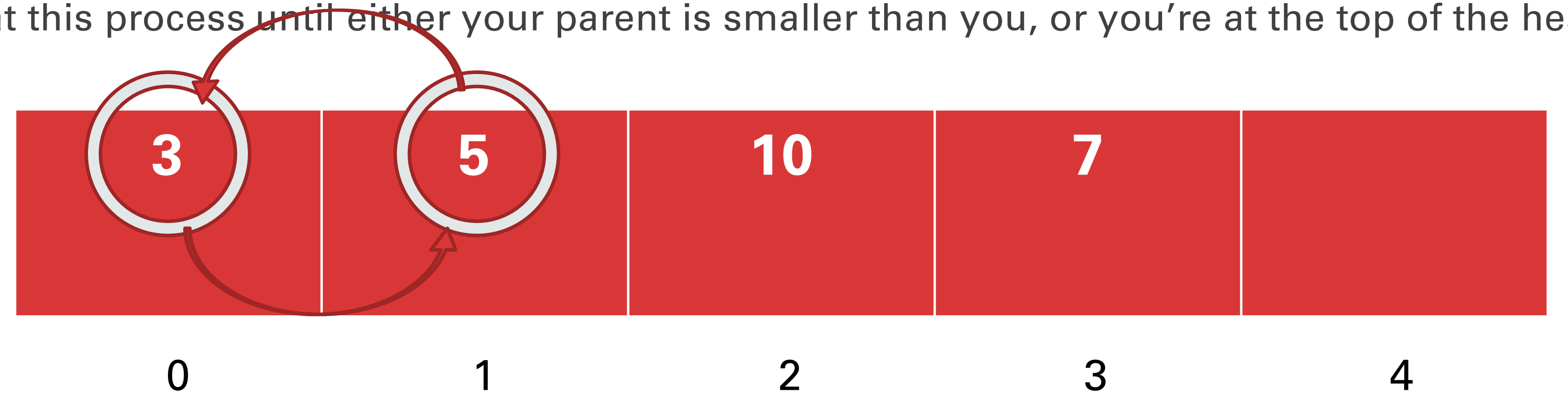
- Let's talk about enqueue()!
  - To enqueue an element, first add it to the end of your pqueue!
  - Next, bubble the element up (if a parent exists!) Compare it with its parent at index (i-1)/2. Swap if your element **is less than** its parent! Be sure to update your element's current index!
  - Repeat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



^ this looks like a face, doesn't it? :p

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! Be sure to update your element's current index!
  - Repeat this process until either your parent is smaller than you, or you’re at the top of the heap!



^ this looks like a face, doesn’t it? :p

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! Be sure to update your element's current index!
  - Repeat this process until either your parent is smaller than you, or you’re at the top of the heap!

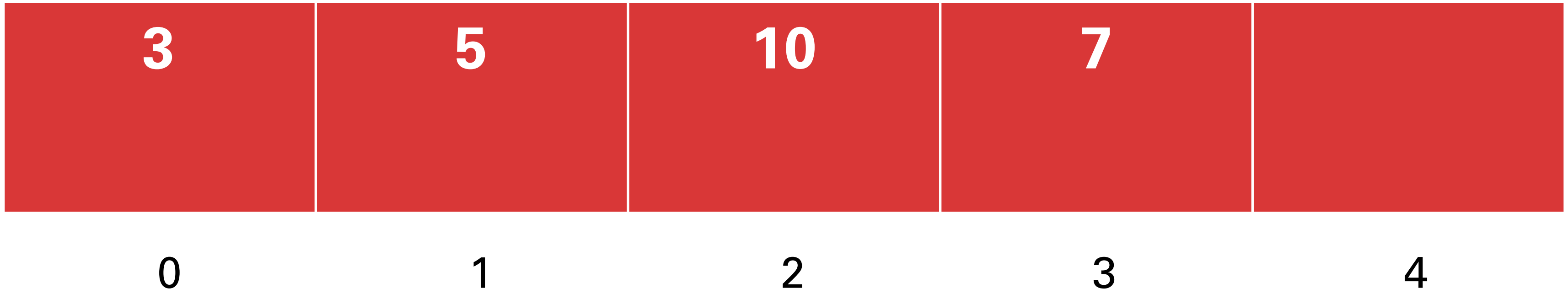
Are we done?

3	5	10	7	
0	1	2	3	4

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! Be sure to update your element's current index!
  - Repeat this process until either your parent is smaller than you, or you’re at the top of the heap!

Done!



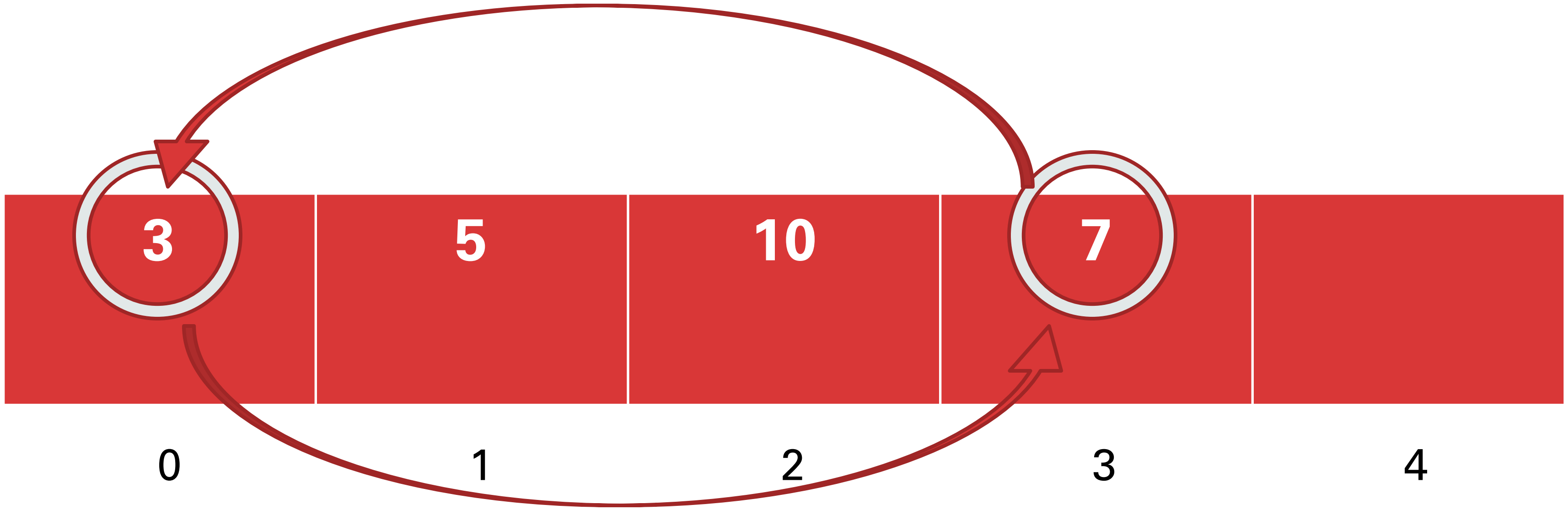
Part 3: Heap PQ

- 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!)

3	5	10	7	
0	1	2	3	4

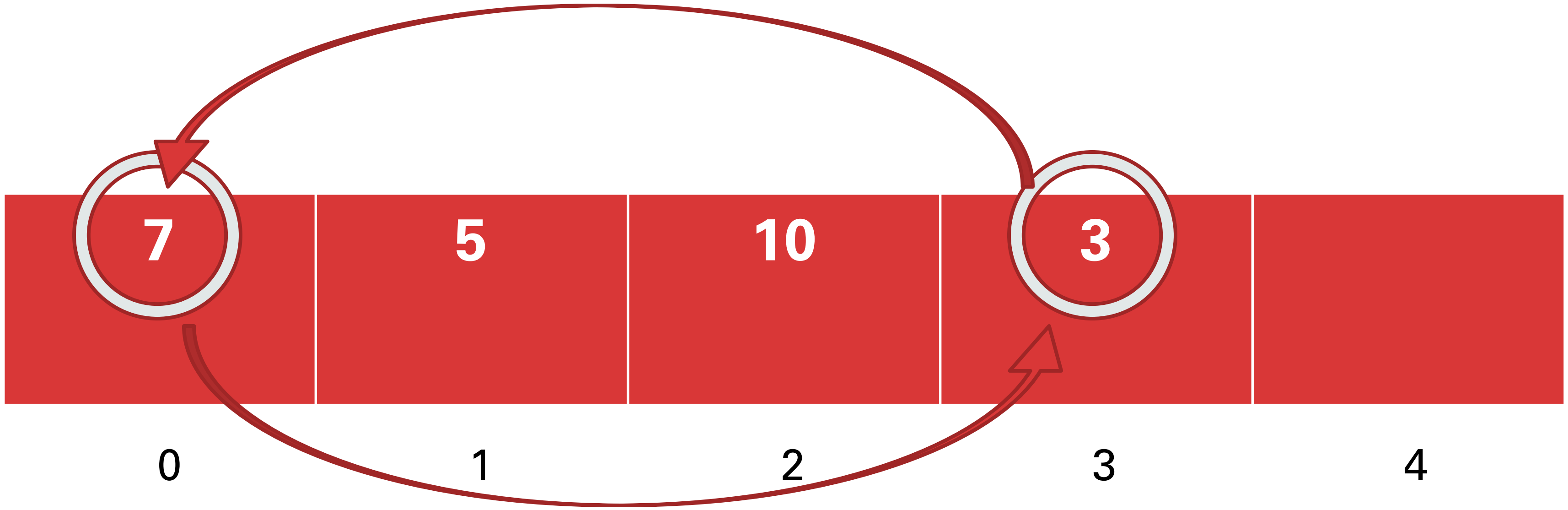
Part 3: Heap PQ

- 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!)



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!)

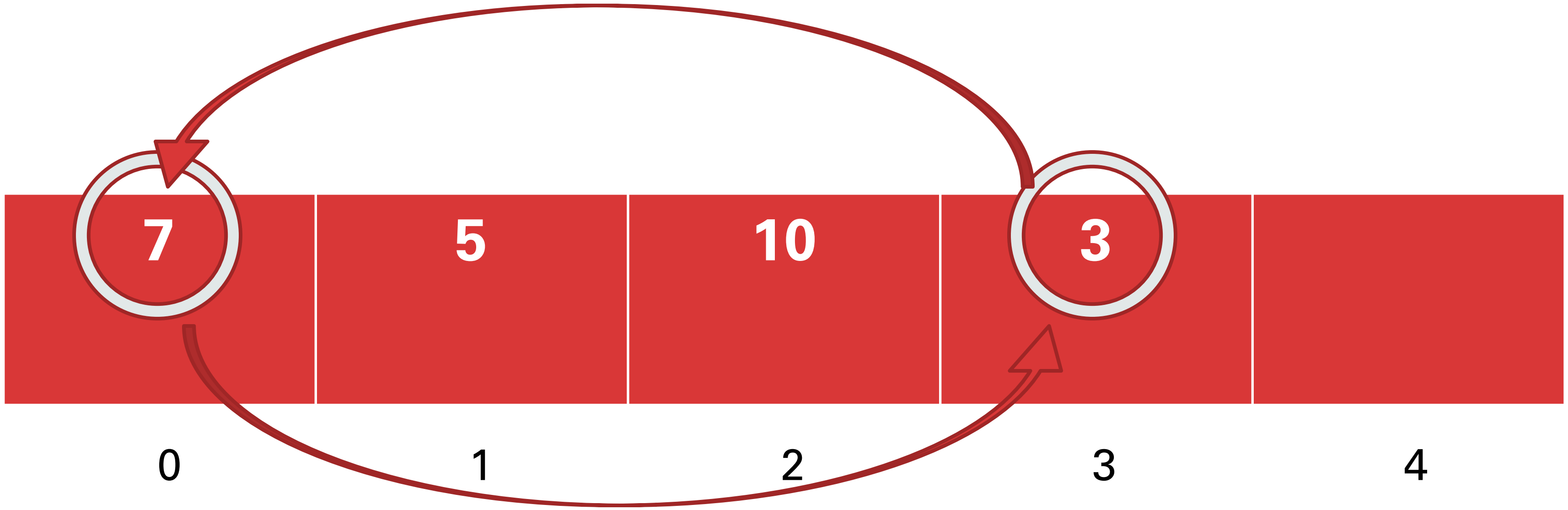


Part 3: Heap PQ

- 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() = 3

Question: what is the PQ's internal capacity?



Part 3: Heap PQ

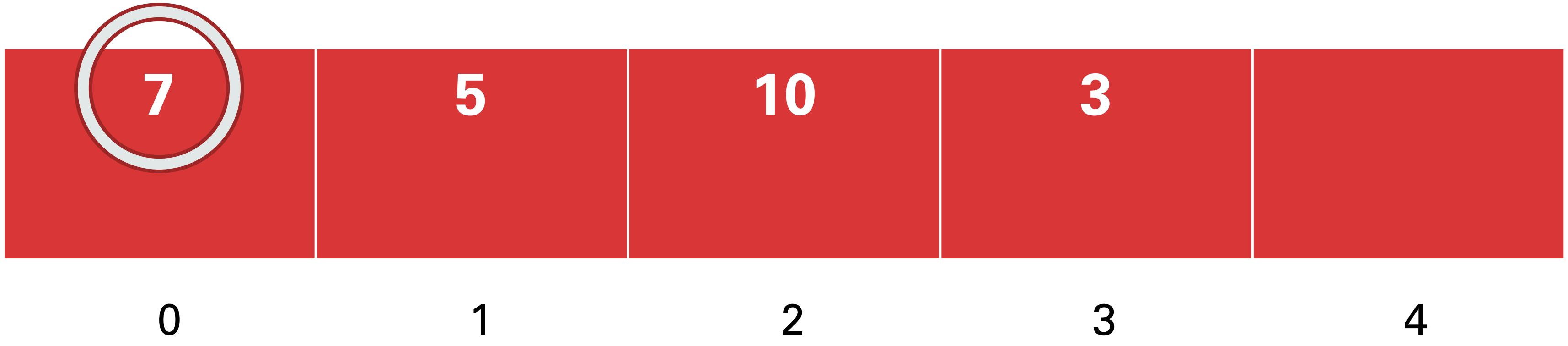
- Let's talk about dequeue()!
  - To start, **swap** your **first** and **last** elements and reduce your size by 1 (you could also just overwrite root!)
  - Next, you want to **bubble down** the root element to its correct place. Compare the root element with its children, who live at indices (2 * i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

7	5	10	3	
0	1	2	3	4

Part 3: Heap PQ

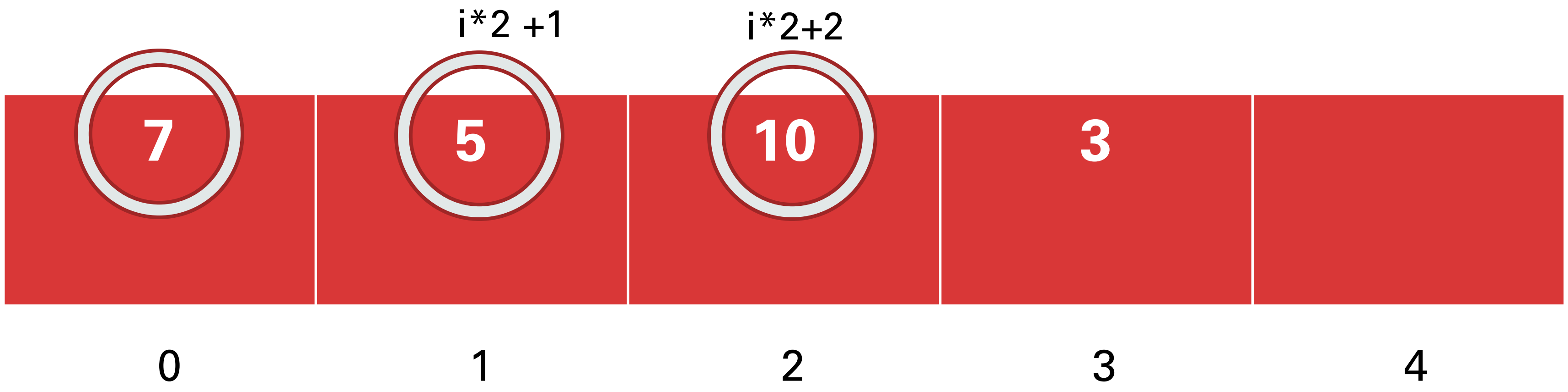
- Let's talk about dequeue()!
  - To start, **swap** your **first** and **last** elements and reduce your size by 1 (you could also just overwrite root!)
  - Next, you want to **bubble down** the root element to its correct place. Compare the root element with its children, who live at indices (2 * i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

Check your understanding: why does swapping with the smaller child matter?



Part 3: Heap PQ

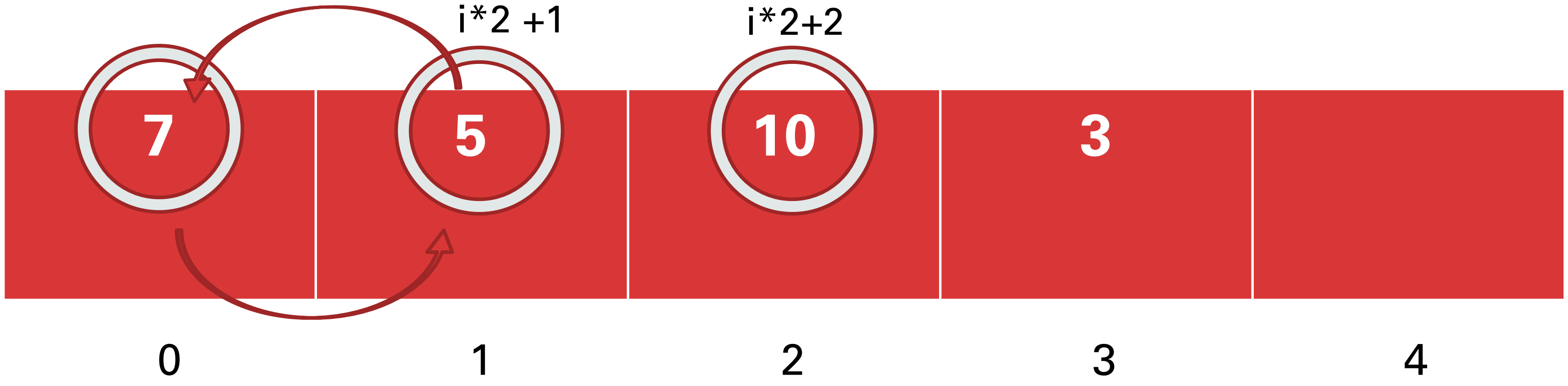
- Let’s talk about dequeue()!
  - To start, **swap** your **first** and **last** elements and reduce your size by 1 (you could also just overwrite root!)
  - Next, you want to **bubble down** the root element to its correct place. Compare the root element with its children, who live at indices (2 * i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

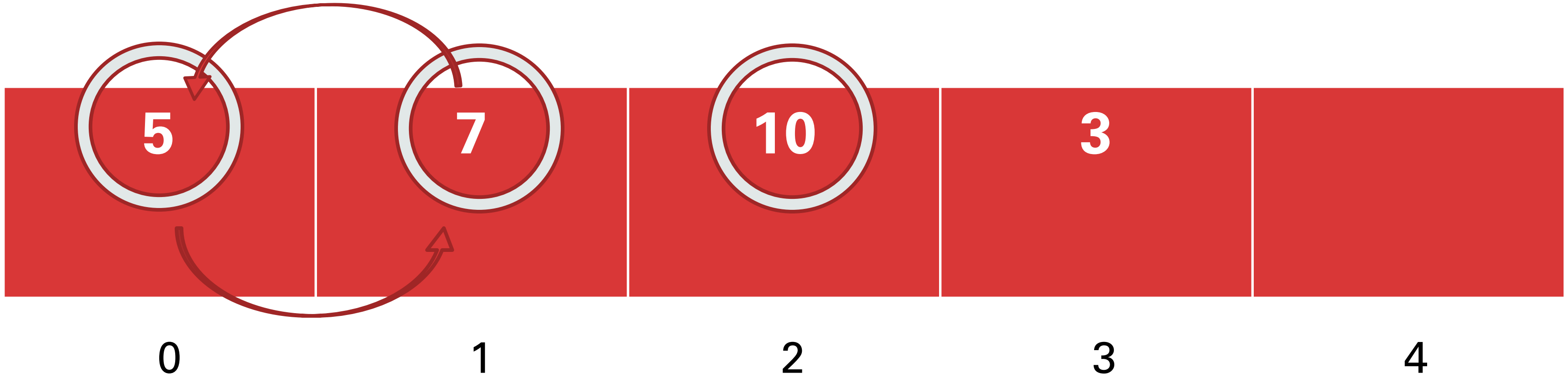
- Let’s talk about dequeue()!
  - To start, **swap** your **first** and **last** elements and reduce your size by 1 (you could also just overwrite root!)
  - Next, you want to **bubble down** the root element to its correct place. Compare the root element with its children, who live at indices (2 * i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

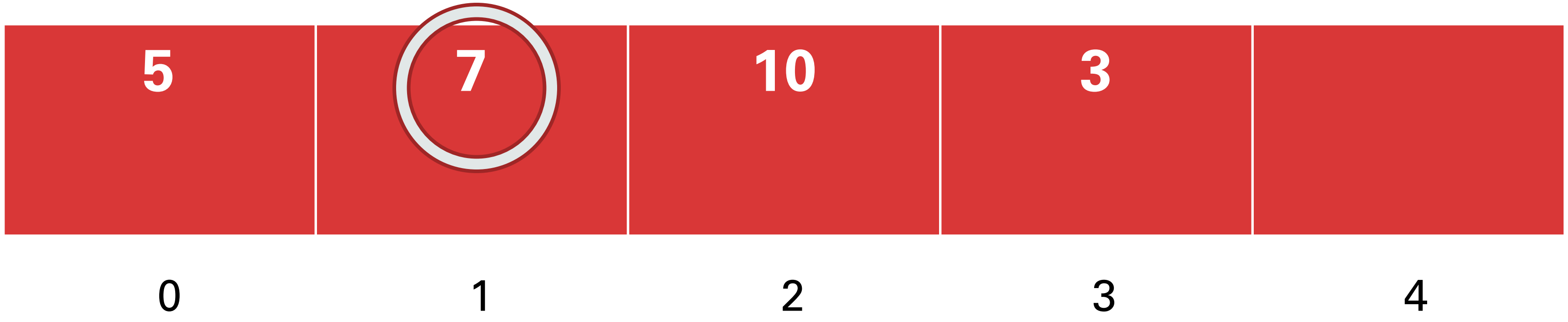
- Let’s talk about dequeue()!
  - To start, **swap** your **first** and **last** elements and reduce your size by 1 (you could also just overwrite root!)
  - Next, you want to **bubble down** the root element to its correct place. Compare the root element with its children, who live at indices (2 * i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

Are we done?

5	7	10	3	
0	1	2	3	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 the root element 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** children left!

Done!

5	7	10	3	
0	1	2	3	4

Part 3: Heap PQ

Helpful hints:

- Be aware that you're implementing a **full** class now! Although you will see overlap between this code and your PQSortedArray code, be mindful about what you copy over!
- Like the other parts of this assignment, you'll be using the **DataPoint** struct to represent elements.
- You will need to **resize** this priority queue if your active size exceeds capacity.
- The bubble functions can be implemented iteratively or recursively.

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.
  - Verify to yourself – why is this true?
- The **validateInternalState**() and **printDebugInfo**() methods can be life-savers here, but they aren't implemented. You'll have to write them yourself!

Part 3: Heap PQ

Helpful hints:

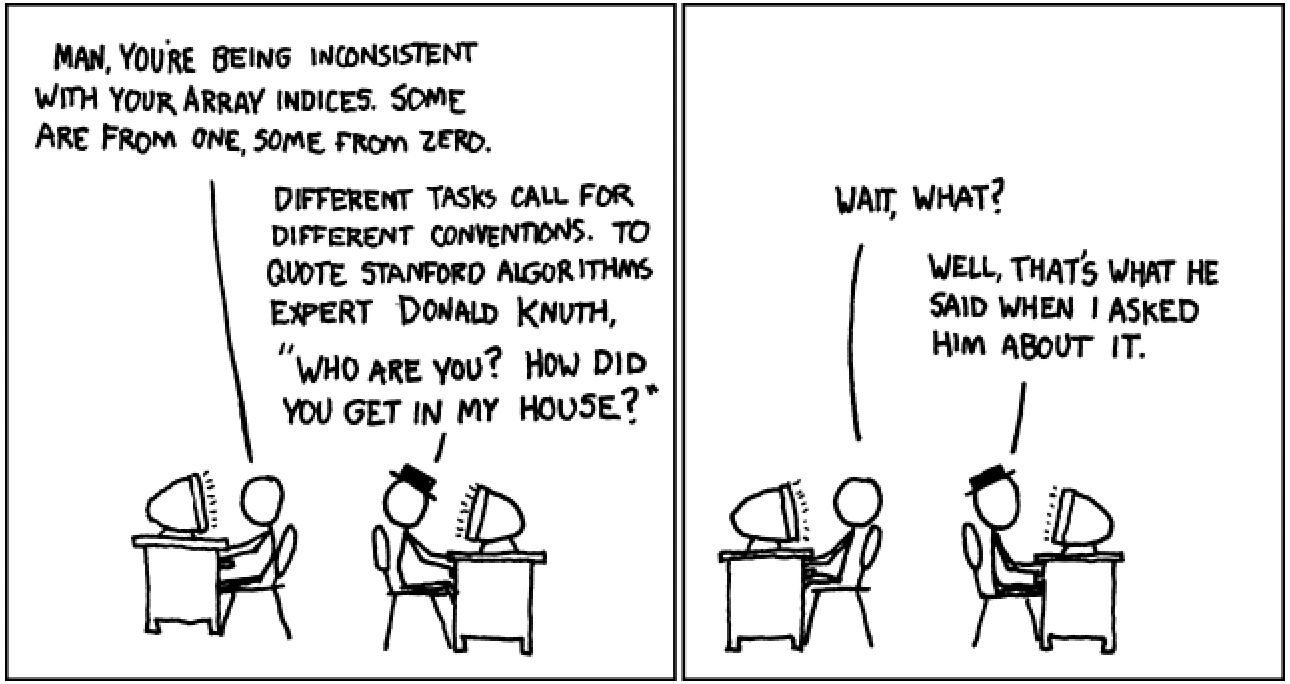
- Verify that the bubble functions work individually before trying to run robustness tests! It can be **very** difficult to locate bugs if they have multiple potential sources.
- Recall the debugging work you did in the first parts of this assignment to help you here – we strongly encourage that you use the debugger and/or the debug helper member functions to hammer out your bugs.
  - Look to the warmups if you think you're getting weird memory errors!

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?



Part 4: Extra Demos!

- You don't have to do *any* extra coding here! Once your program is done, try running tests from the **demos.cpp** file to view representations of large real-world data sets that use your new data structure!
- It's an amazing graphical demo, so be sure to check it out **after** you've finished the assignment. It won't work before ;)

You did it!

Best of luck on this assignment!

Think about what you've just made – you can now *create* the data structures that we taught you about in the beginning of the class. Go you!