YEAH Hours A6

Data Sagas

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!

Assignment logistics

- The assignment is due on **Friday February 26th at class time.** You're welcome to work in pairs.
- Try and start early! This one is rather different than the other assignments you've done so far -- you can bugs end up being much more subtle!

The Breakdown:

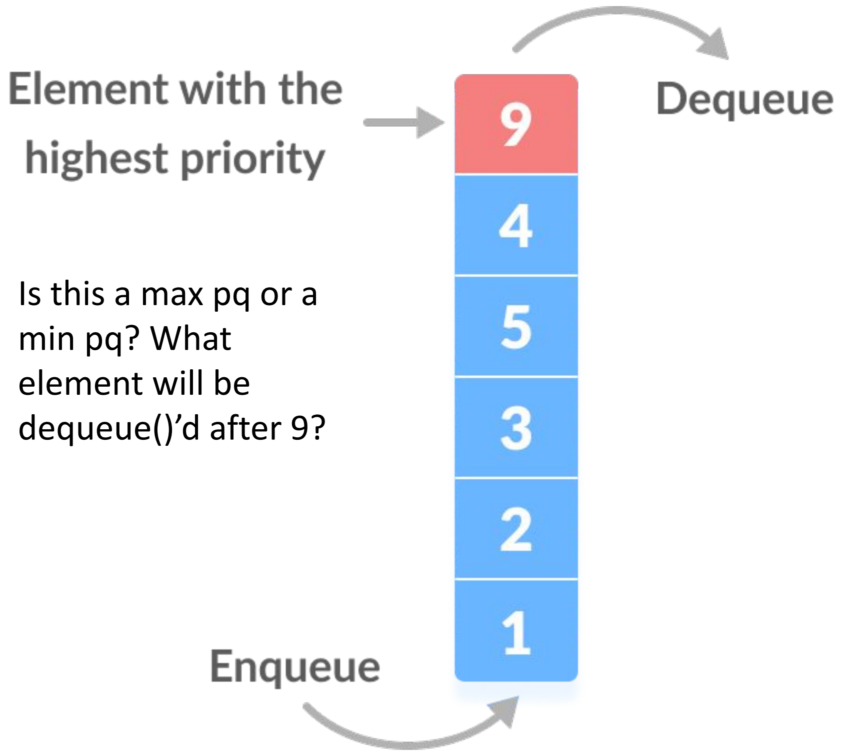
1. **Array exploration**: A debugging warmup where you'll learn all about arrays in C++ and possibly memory errors!
2. **HeapPQ**: A 106B classic, where you'll be implementing a priority min-queue using a binary heap!
3. **Streaming Top-K**: Using your new priority queue, you can do some pretty cool things!

Part 1: Array Exploration

- We want this part to be pretty straightforward, but also a great introduction to arrays and debugging for the next part of the assignment!
- Go into **`ExploreArrays.cpp`** and set a breakpoint at the top of the file.
  - From there, follow the instructions in **`ShortAnswers.txt`** and you should be good to go!

Let’s discuss: 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.
  - For this assignment, you’ll be dealing with **min** PQ, meaning that you’ll be concerned dequeue’ing the smallest priority first ☺



Is this a max pq or a min pq? What element will be dequeue()’d after 9?

Part 2: Heap PQ

- In this second part, you'll be implementing a full priority queue using a **binary min heap**!
  - As per this assignment, 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** which is a cool new data structure!
- Here's the entire PQ interface, most of which you'll implement:

```
class HeapPQueue {
public:
    HeapPQueue();
    ~HeapPQueue();
    void enqueue(const DataPoint& data);
    DataPoint dequeue();
    DataPoint peek() const;
    bool isEmpty() const;
    int  size() const;
    void printDebugInfo();
private:
    /* Up to you! */
};
```

Part 2: Heap PQ

- In this second part, you'll be implementing a full priority queue using a **binary min heap**!
  - As per this assignment, 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** which is a cool new data structure!
- Here's the entire PQ interface, most of which you'll implement:

```
class HeapPQueue {
public:
    HeapPQueue();
    ~HeapPQueue();
    void enqueue(const DataPoint& data);
    DataPoint dequeue();
    DataPoint peek() const;
    bool isEmpty() const;
    int  size() const;
    void printDebugInfo();
private:
    /* Up to you! */
};
```

Your underlying data container should be a C++ array! This means that you'll be responsible for **allocating, deleting, and resizing** your PQ.

Many of these tasks, like size() and peek() should be **very** straightforward!

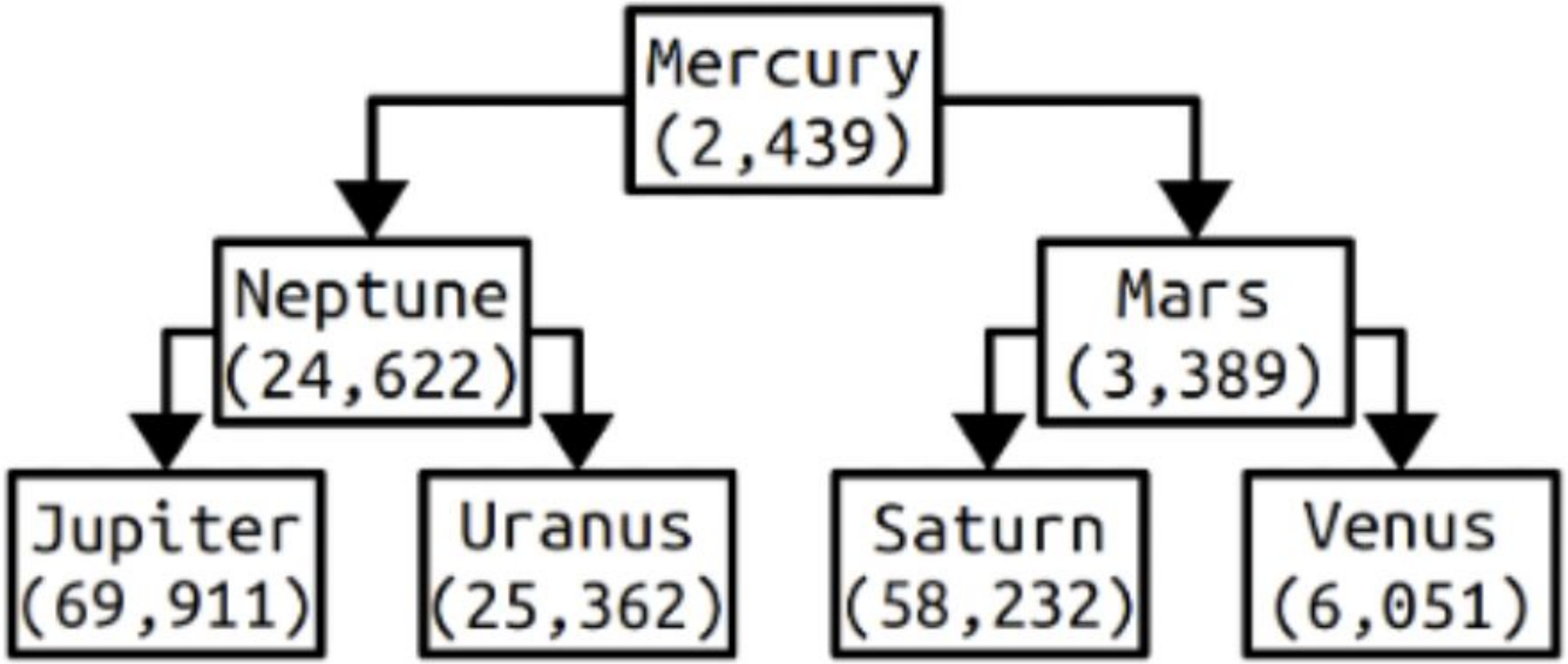
printDebugInfo() is a helper method that you can write to print the contents of your PQ at any time!

Review: Binary Heap

- Before starting this assignment, we **highly** recommend reading the handout specification about **binary heaps** multiple times.
  - Without a good understanding of how these structures work, you **will not** be able to implement the priority queue!
  - Luckily, the binary heap isn't too complex!

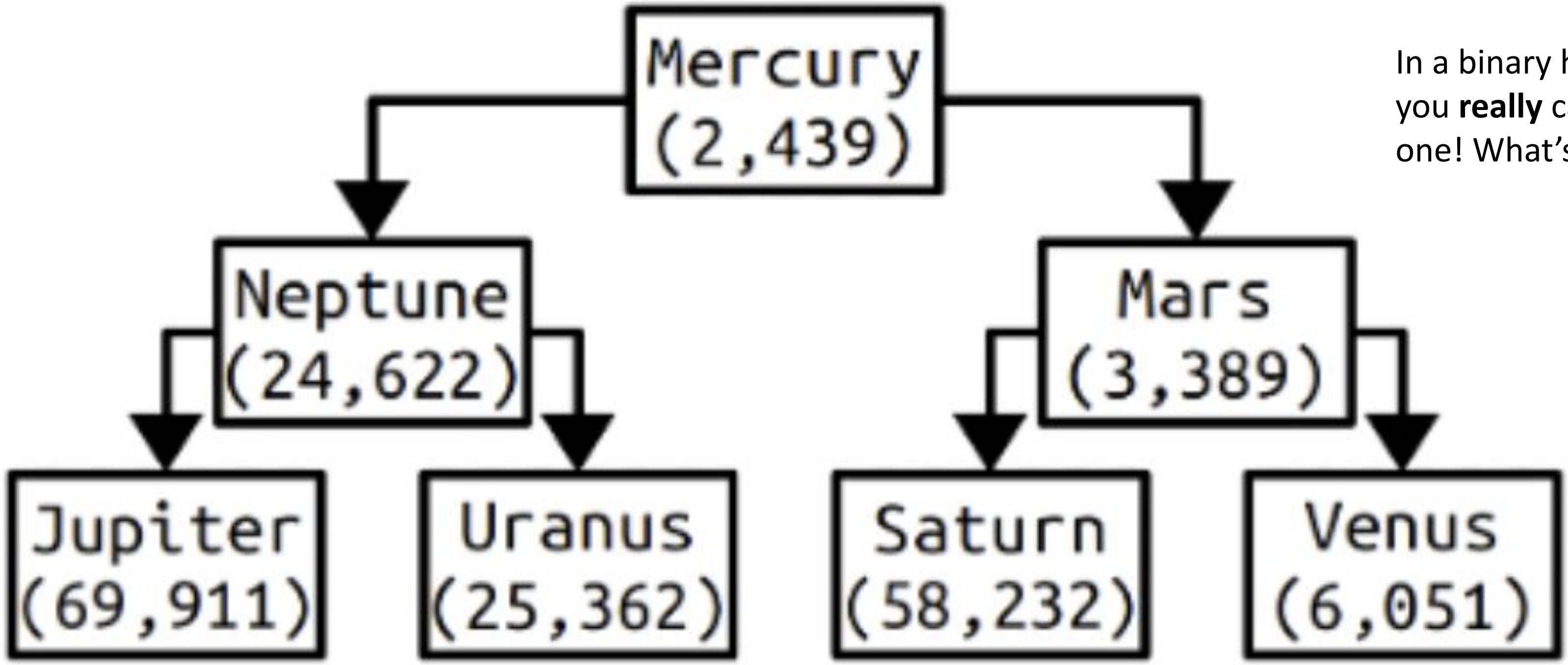
Review: Binary Heap

- One way to think about a binary heap is via a tree-hierarchy diagram like so:



Review: Binary Heap

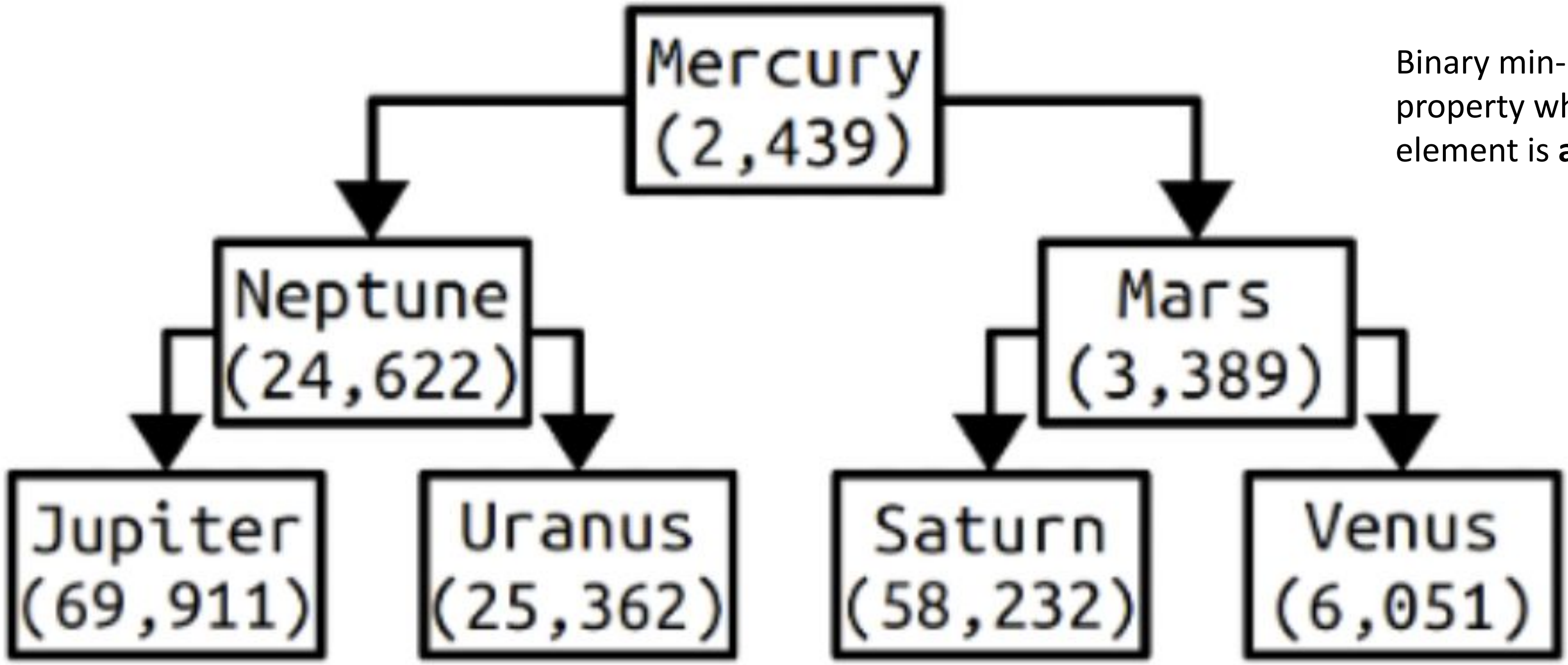
- One way to think about a binary heap is via a tree-hierarchy diagram like so:



In a binary heap, the only element you **really** care about is the very top one! What’s special about it?

Review: Binary Heap

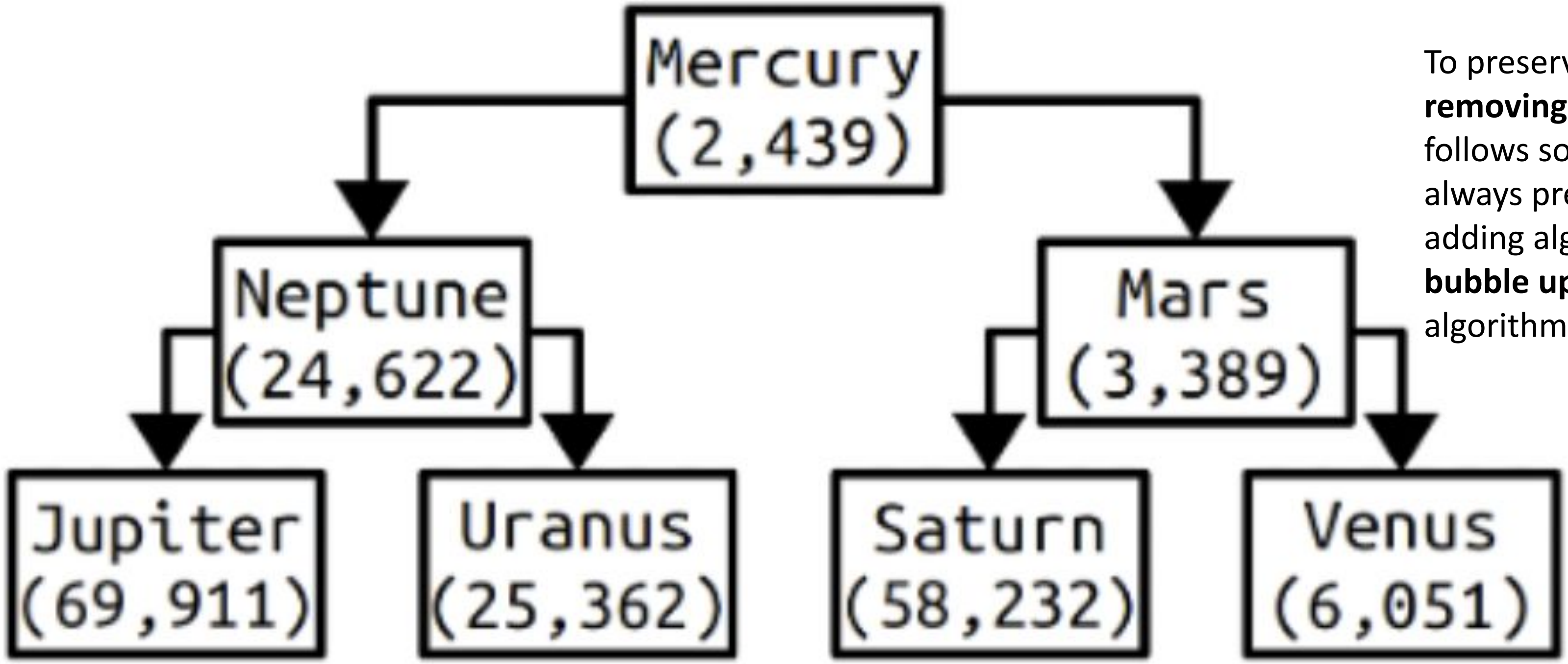
- One way to think about a binary heap is via a tree-hierarchy diagram like so:



Binary min-heaps have a special property where the topmost element is **always** the smallest!

Review: Binary Heap

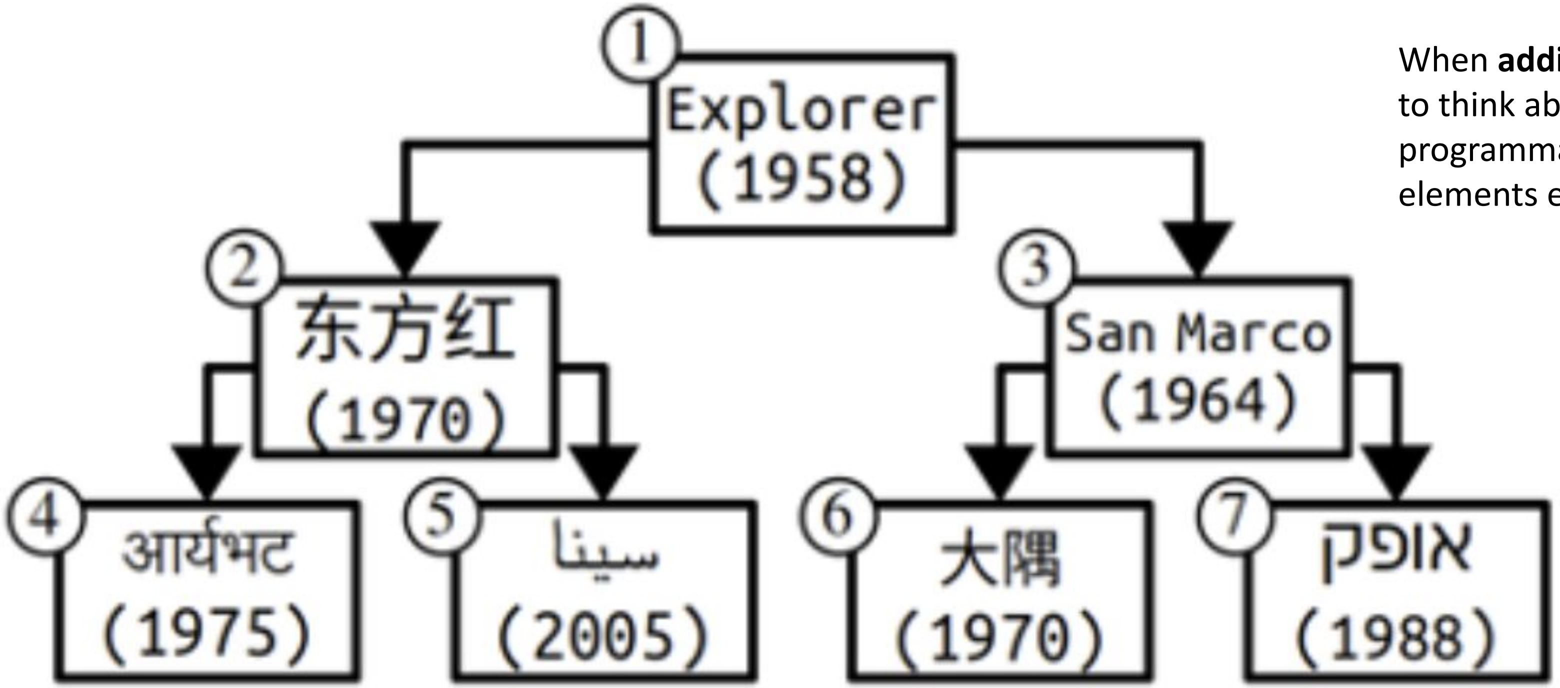
- One way to think about a binary heap is via a tree-hierarchy diagram like so:



To preserve this rule, **adding** and **removing** from a **binary heap** follows some special rules that will always preserve this order. The adding algorithm is informally called **bubble up**, and the removal algorithm is called **bubble down.**

Review: Binary Heap

- Let's look at another binary heap:



When **adding/removing**, it's easier to think about the process programmatically when these elements exist in an **array!**

Review: Binary Heap

- Let’s look at another binary heap:

As you can see, the labels 1-7 became indices in an array! Notice that our array is **1-indexed**! This will make our math easier in the future.

Explorer (1958)	东方红 (1970)	San Marco (1964)	आर्यभट (1975)	سينا (2005)	大隅 (1970)	אופק (1988)
1	2	3	4	5	6	7

Review: Binary Heap

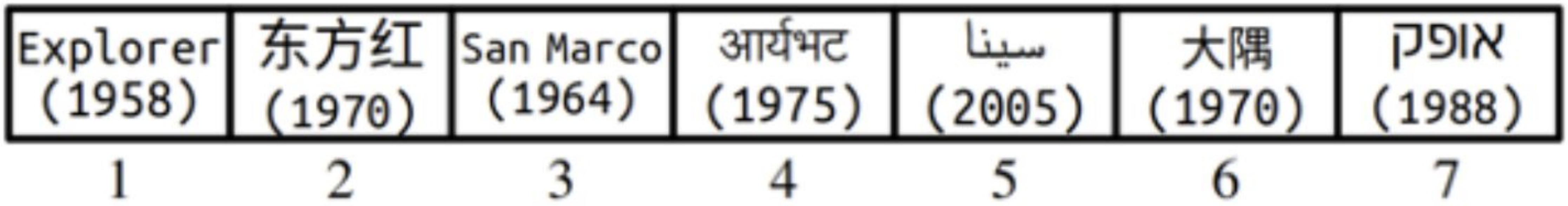
- Let’s look at another binary heap:

Notice that the element with the smallest weight is at the **front** of our array (index 1)! That’ll make our life easier when peeking / dequeueing

Explorer (1958)	东方红 (1970)	San Marco (1964)	आर्यभट (1975)	سينا (2005)	大隅 (1970)	אופק (1988)
1	2	3	4	5	6	7

Review: Binary Heap

- Let’s discuss how to add / remove from a **binary heap** in array-form (like you’ll do in this assignment!)

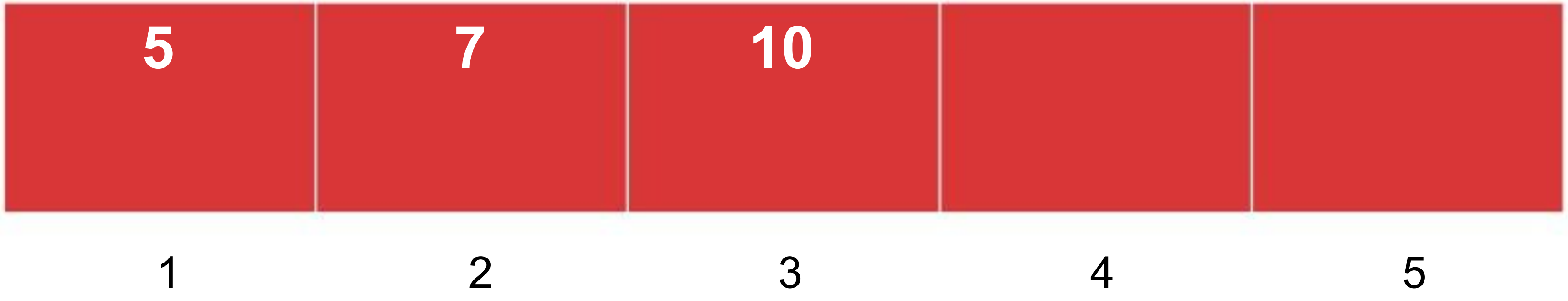


Part 2: Heap PQ

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

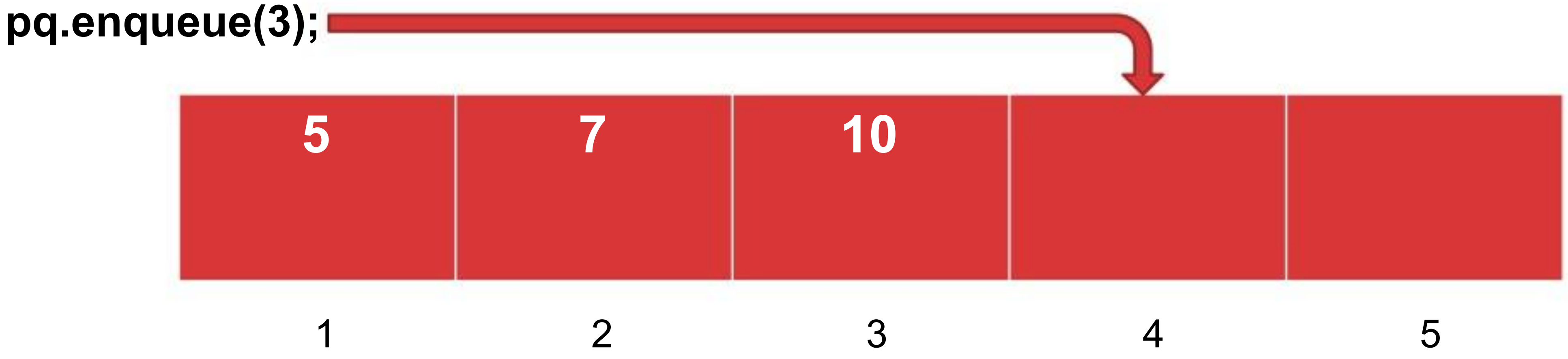
For this demo, I'll use integers to represent **DataPoint** weights for clarity.

pq.enqueue(3);



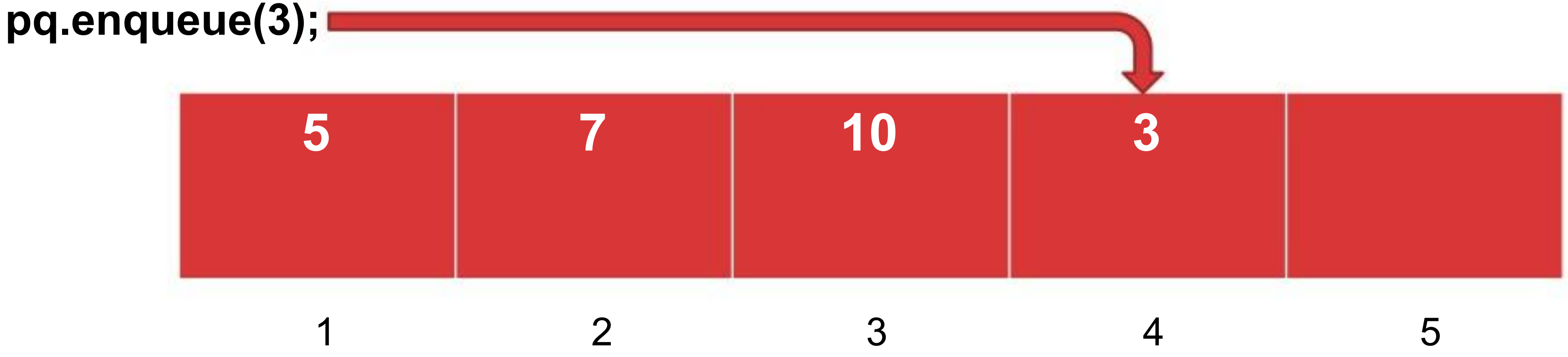
Part 2: Heap PQ

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



Part 2: Heap PQ

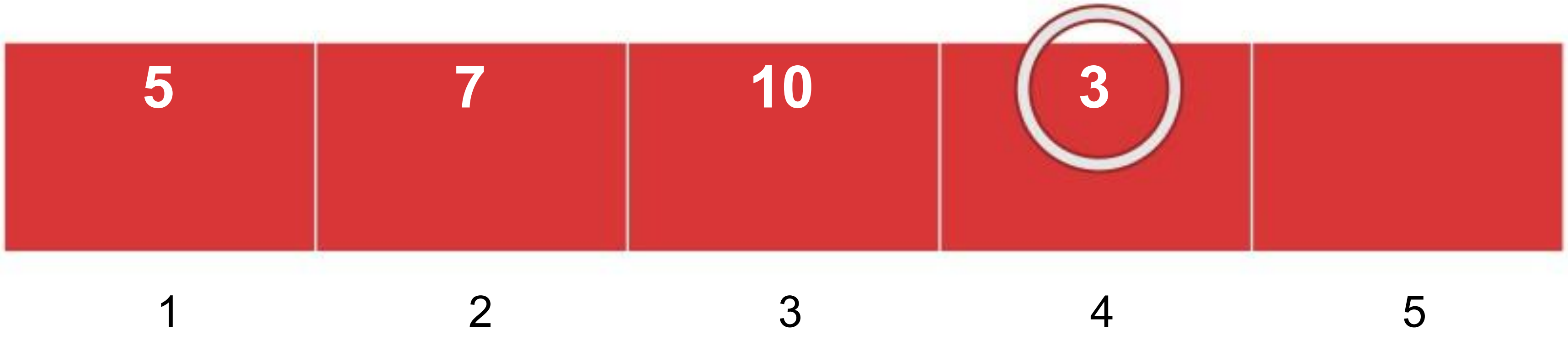
- Let's talk about enqueue()!
  - To enqueue an element, first add it to the end of your pqueue!



Part 2: 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/2. Swap if your element **is less than** its parent!

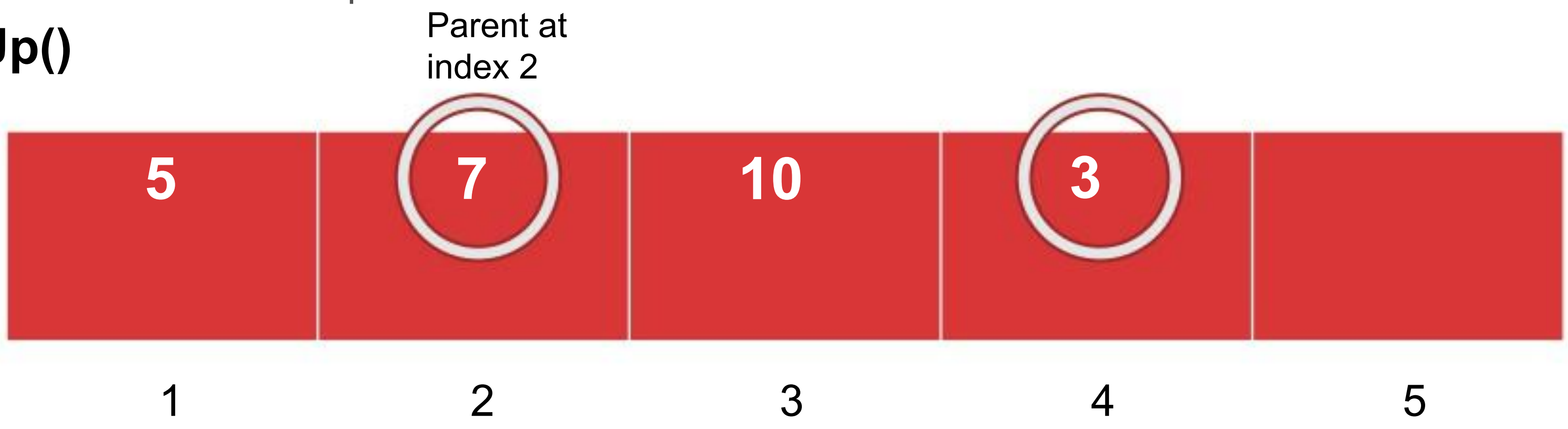
bubbleUp()



Part 2: 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/2. Swap if your element **is less than** its parent!

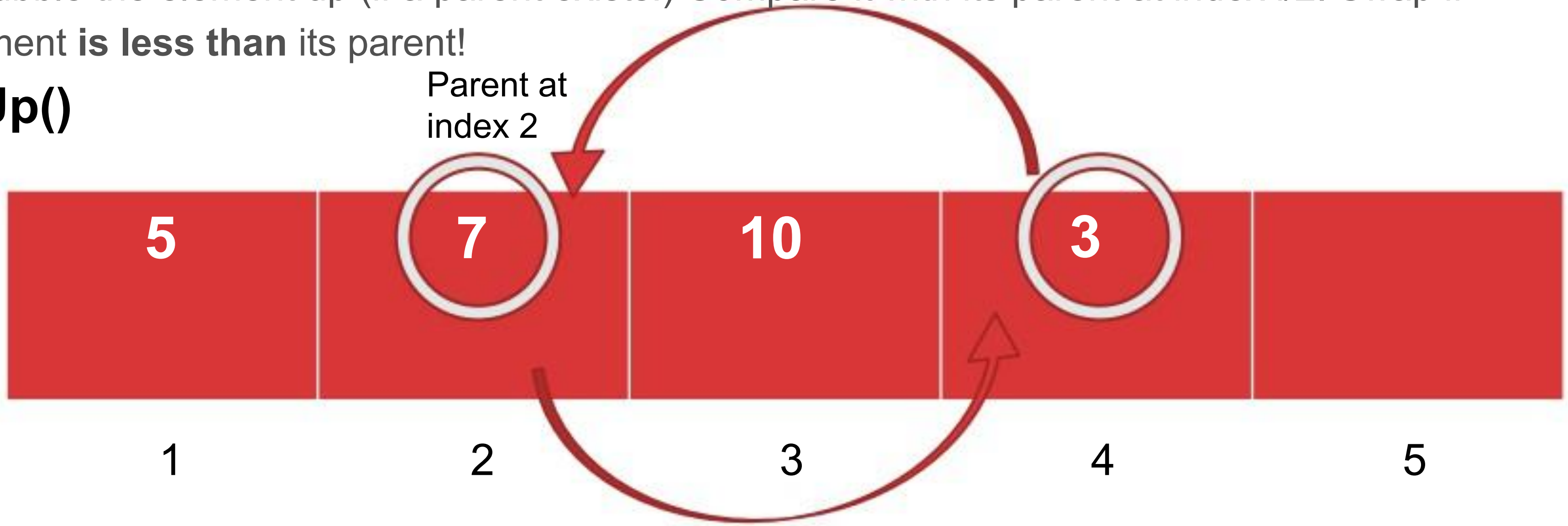
bubbleUp()



Part 2: 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/2. Swap if your element **is less than** its parent!

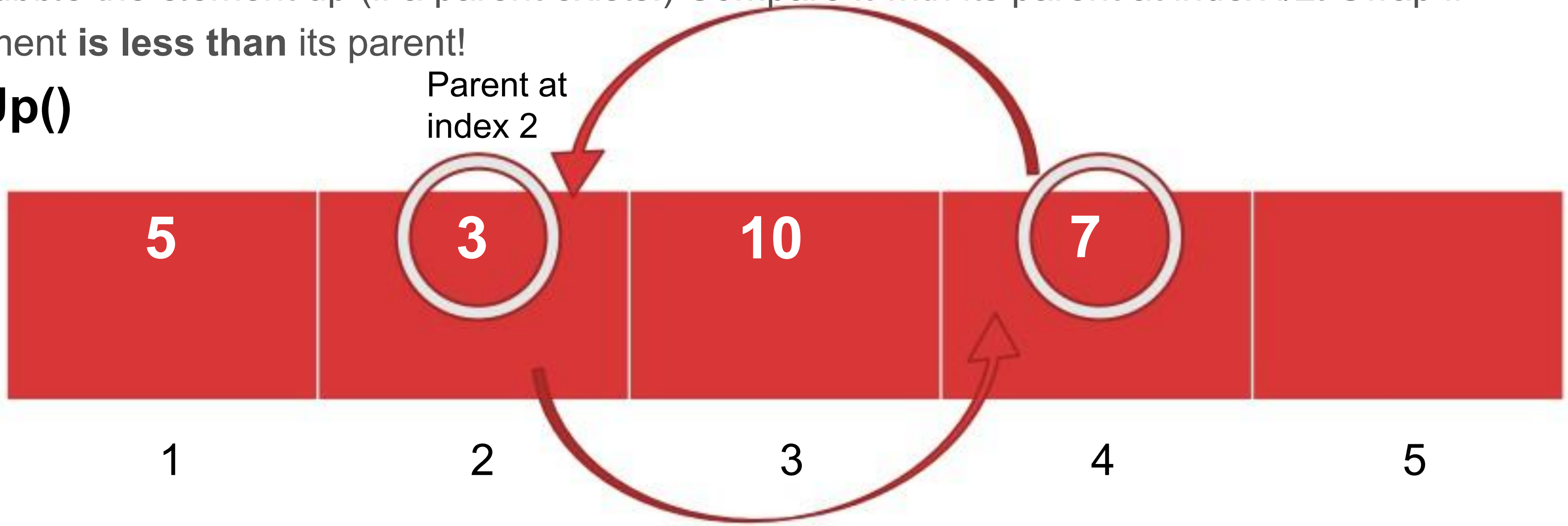
bubbleUp()



Part 2: 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/2. Swap if your element **is less than** its parent!

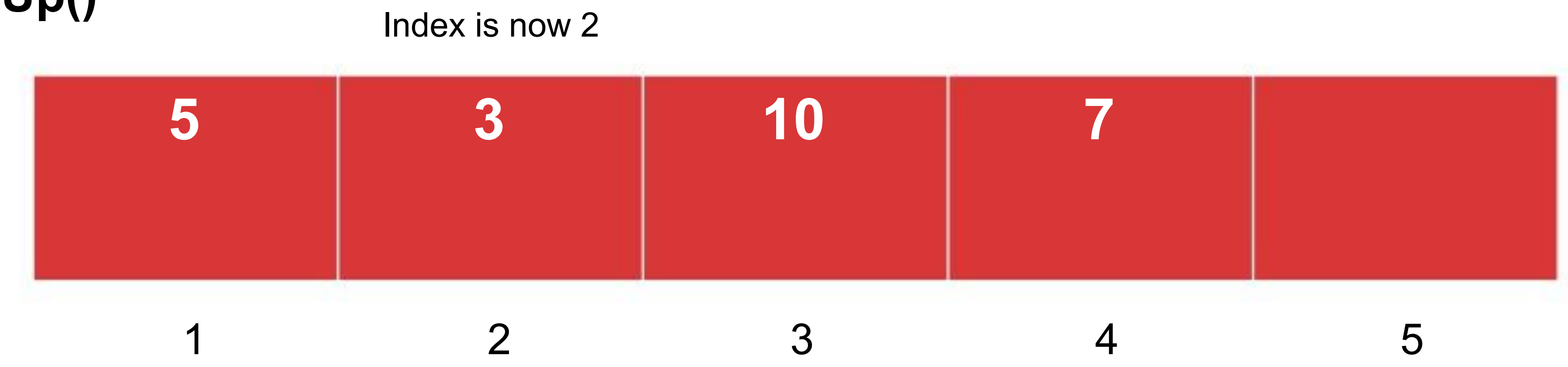
bubbleUp()



Part 2: 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/2. Swap if your element **is less than** its parent! Be sure to update your element's current index!

bubbleUp()



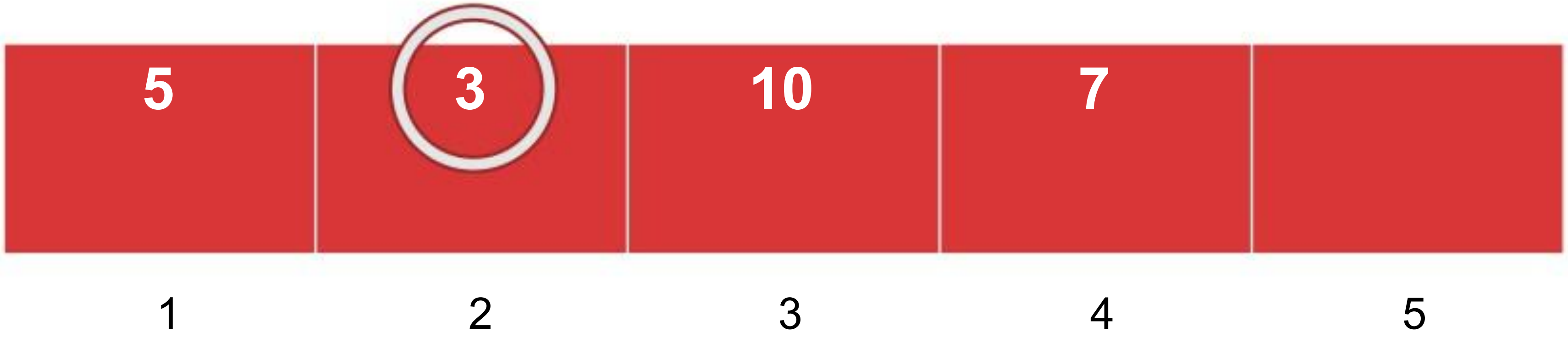
Part 2: Heap PQ

- 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/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**	
1	2	3	4	5

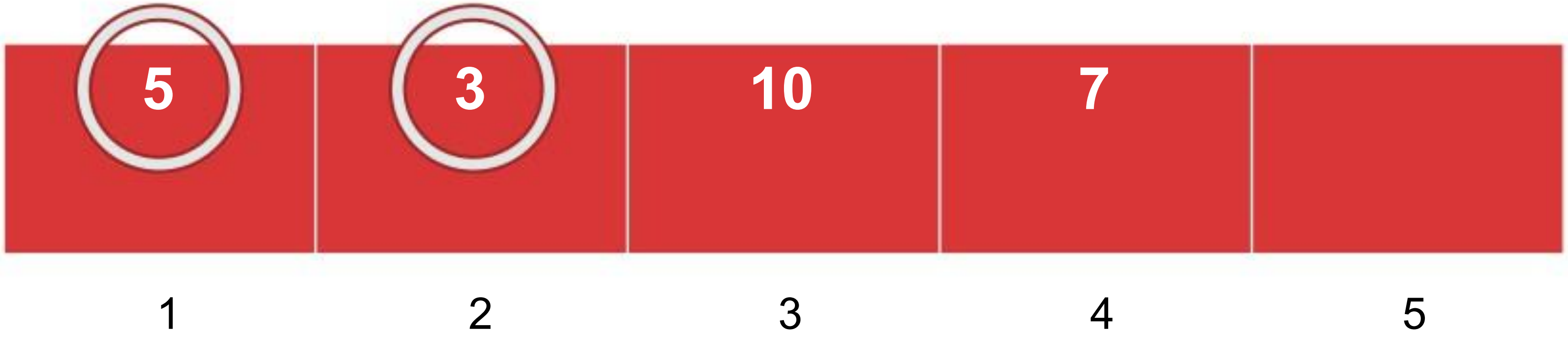
Part 2: Heap PQ

- 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/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!



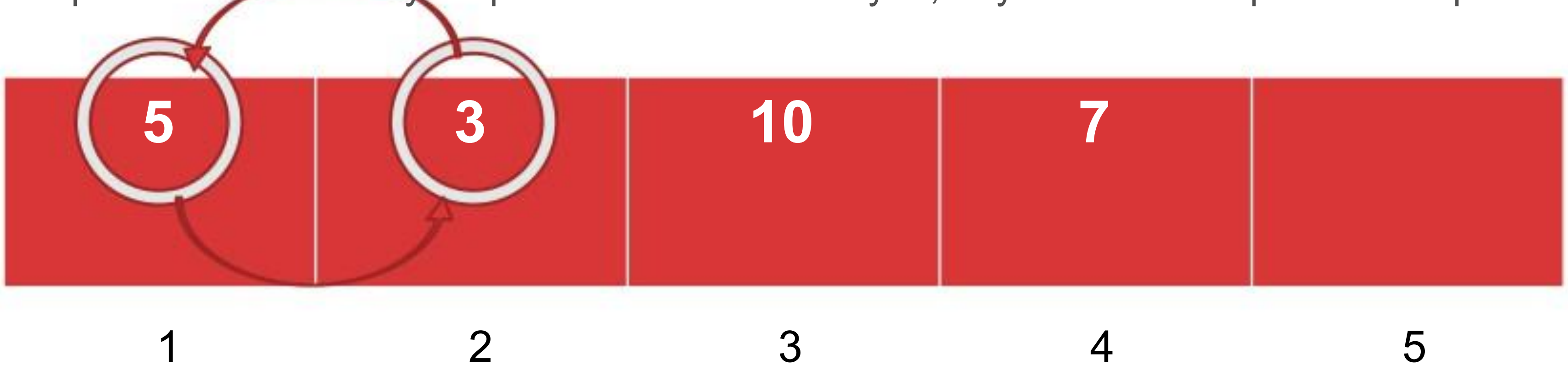
Part 2: Heap PQ

- 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/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 2: Heap PQ

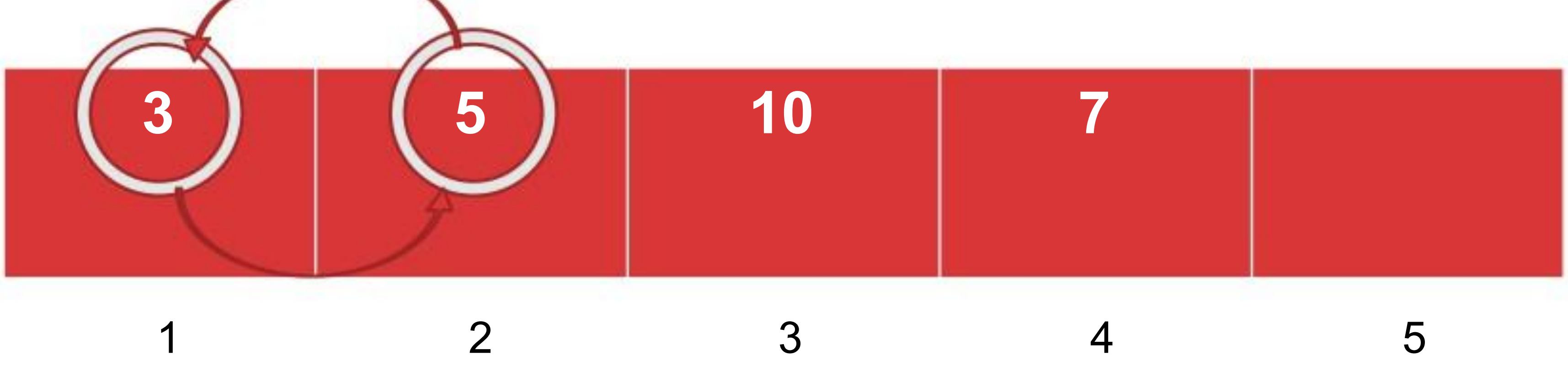
- Let's talk about enqueue()!
  - To enqueue an element, first add it to the end of your pqueue!
  - Next, bubble the element up (if a parent exists!) Compare it with its parent at index i/2. Swap if your element **is less than** its parent! Be sure to update your element's current index!
  - Repeat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

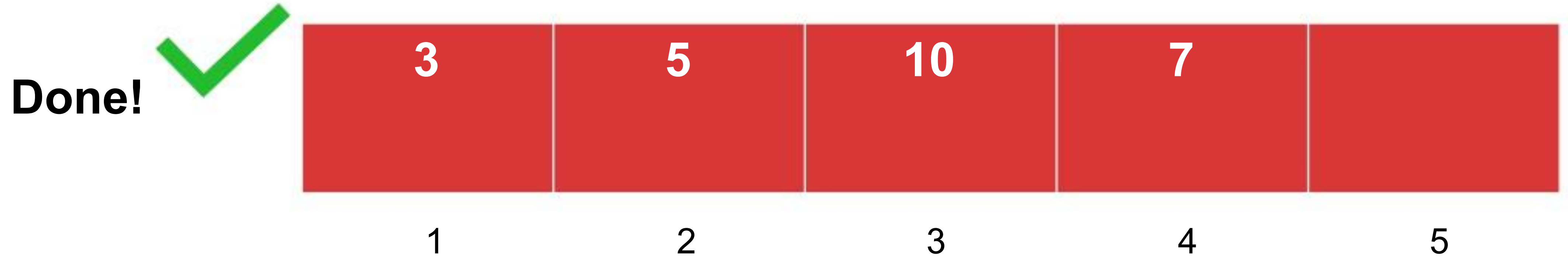
- Let's talk about enqueue()!
  - To enqueue an element, first add it to the end of your pqueue!
  - Next, bubble the element up (if a parent exists!) Compare it with its parent at index i/2. Swap if your element **is less than** its parent! Be sure to update your element's current index!
  - Repeat 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**	
1	2	3	4	5

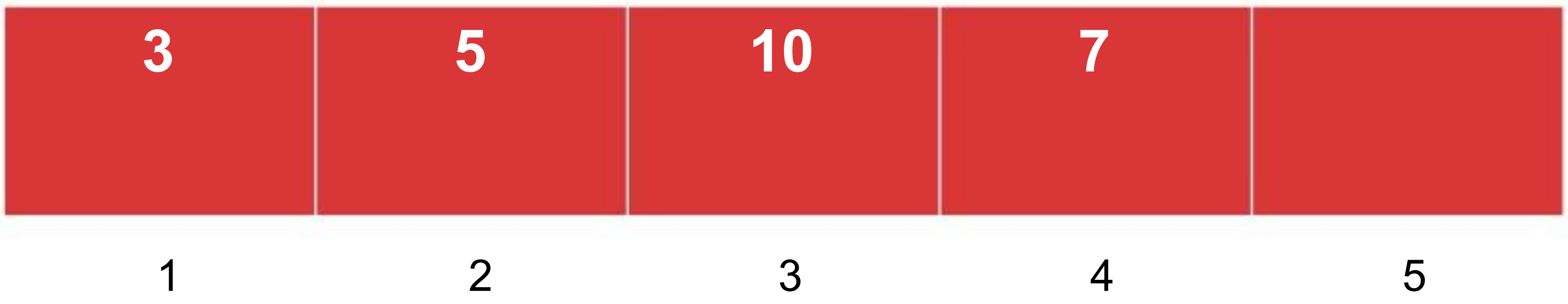
Part 2: Heap PQ

- 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/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 (index 1)!



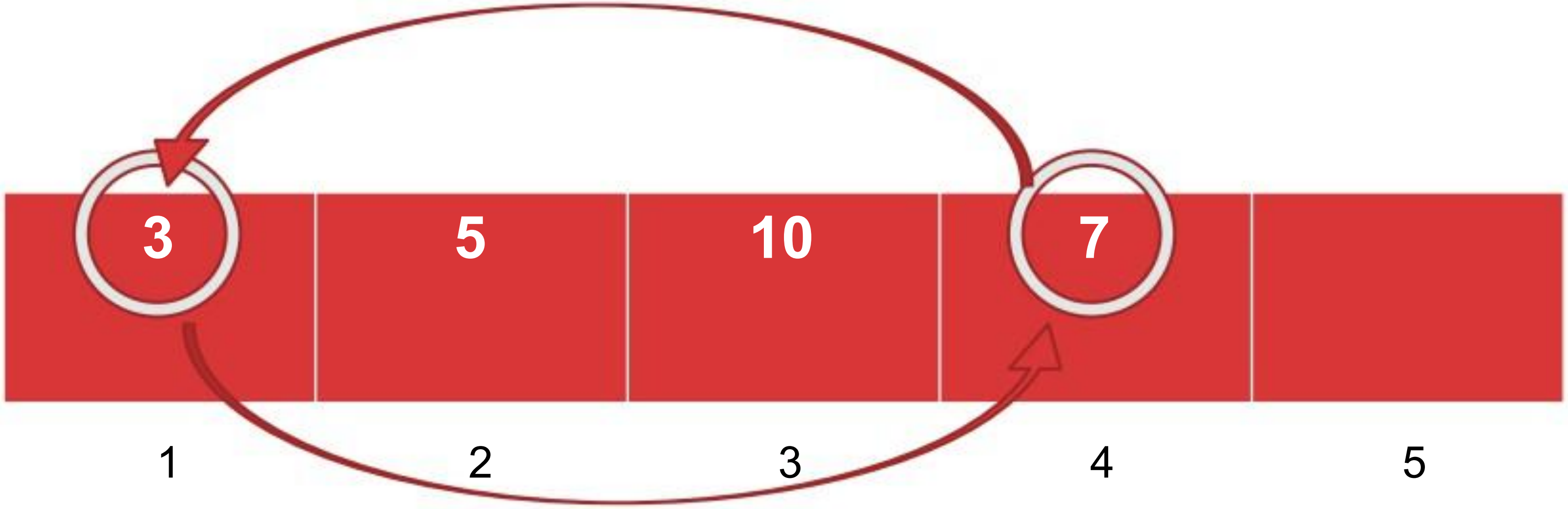
Part 2: Heap PQ

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



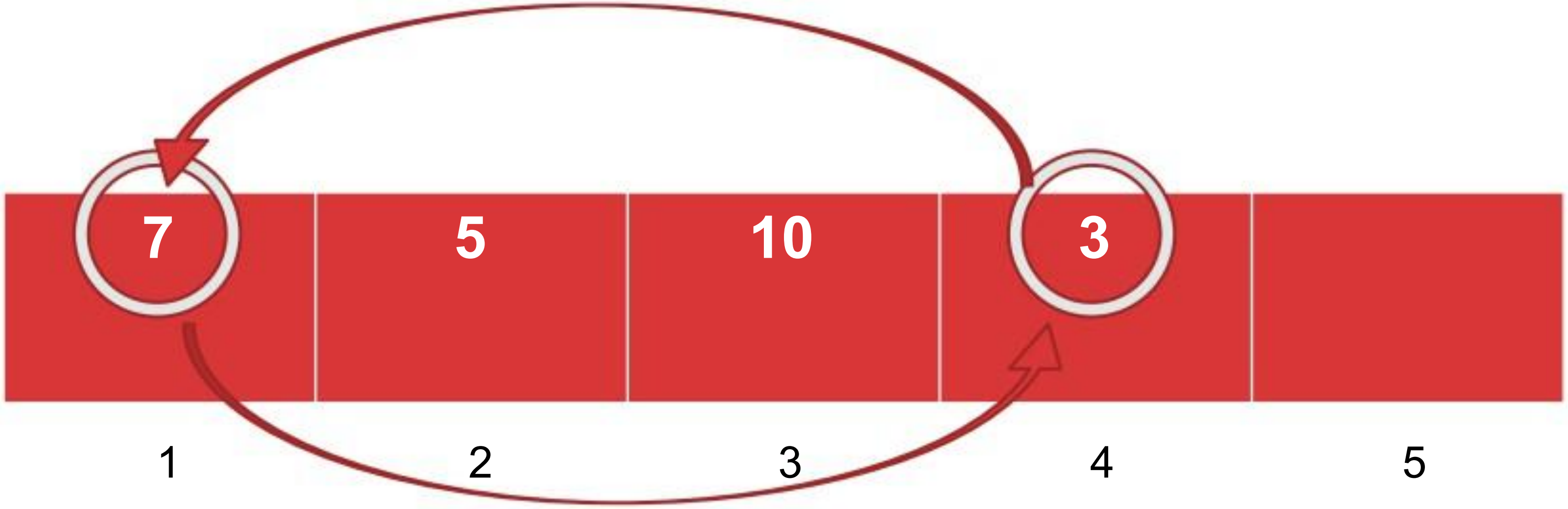
Part 2: Heap PQ

- 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 2: 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!)

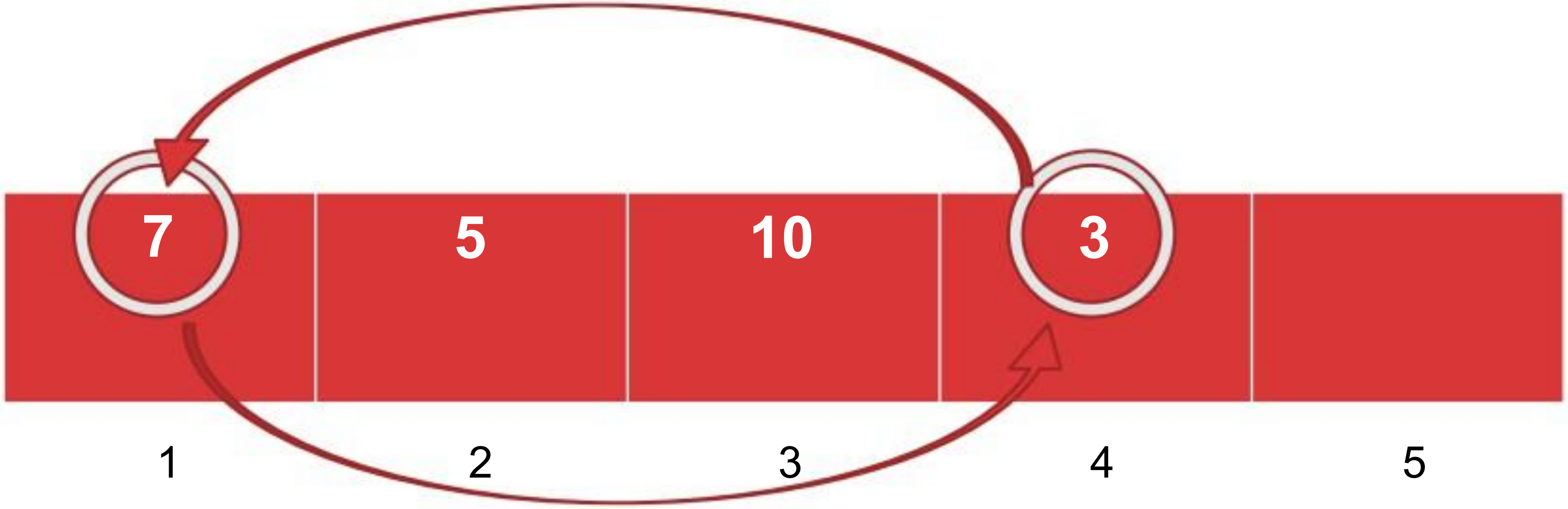


Part 2: Heap PQ

- 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!)

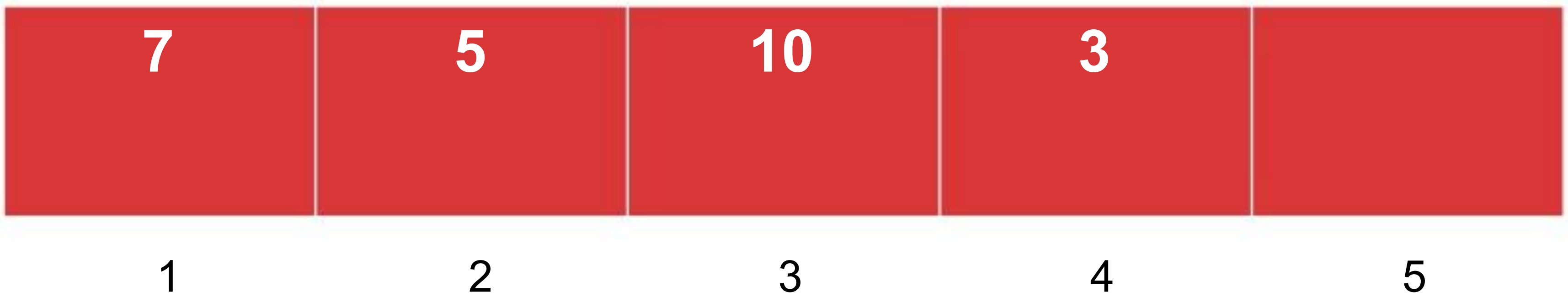
pq.size() = 3

Question: what is the PQ's internal capacity?



Part 2: Heap PQ

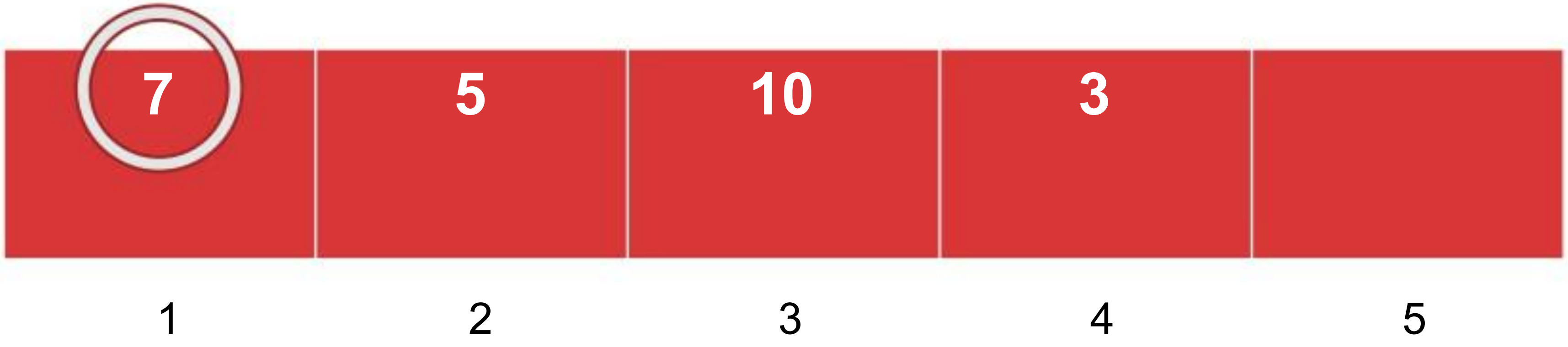
- Let's talk about dequeue()!
  - To start, **swap** your **first** and **last** elements and reduce your size by 1 (you could also just overwrite root!)
  - Next, you want to **bubble down** the root element to its correct place. Compare the root element with its children, who live at indices (2 * i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



Part 2: Heap PQ

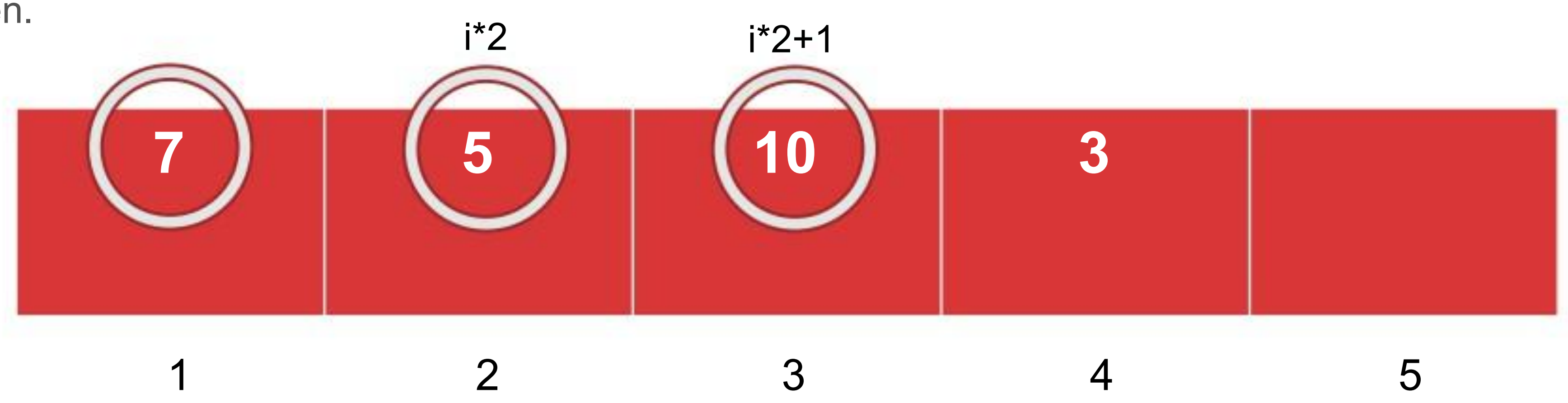
- Let's talk about dequeue()!
  - To start, **swap** your **first** and **last** elements and reduce your size by 1 (you could also just overwrite root!)
  - Next, you want to **bubble down** the root element to its correct place. Compare the root element with its children, who live at indices (2 * i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

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



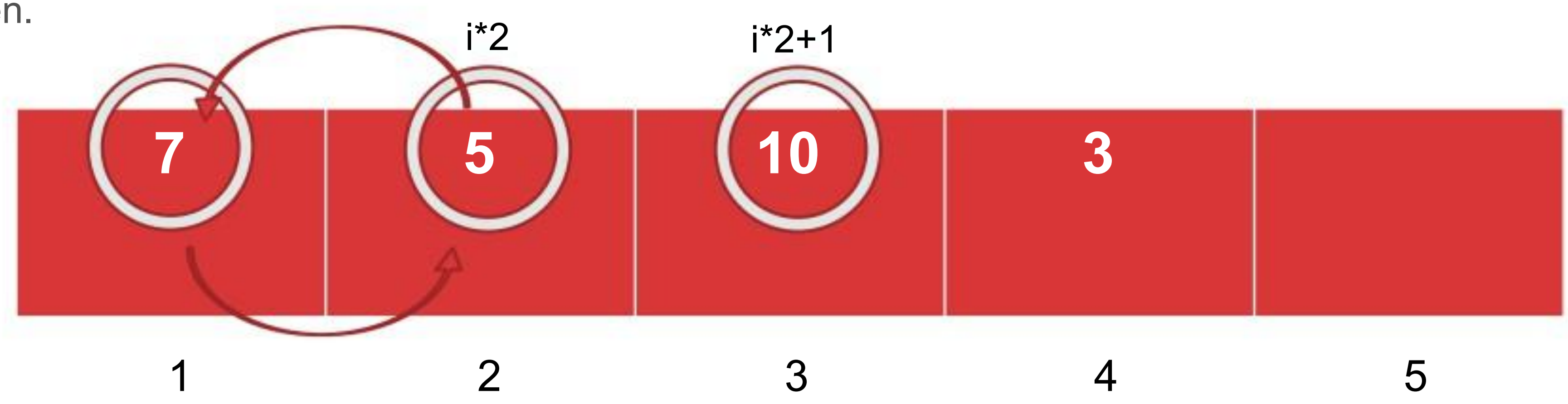
Part 2: Heap PQ

- 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), and swap your element with the **smaller** of the children.



Part 2: Heap PQ

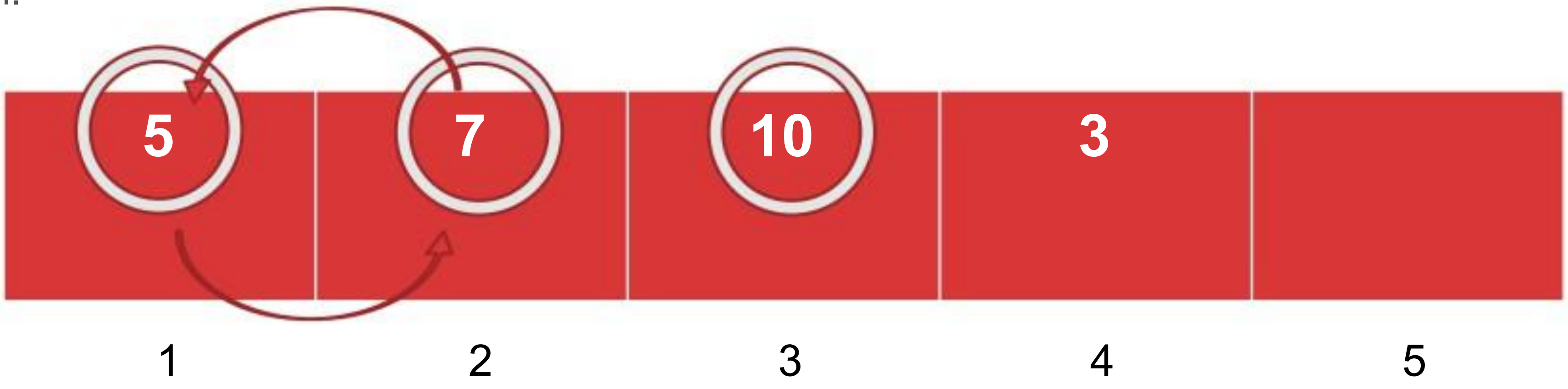
- Let's talk about dequeue()!
  - To start, **swap** your **first** and **last** elements and reduce your size by 1 (you could also just overwrite root!)
  - Next, you want to **bubble down** the root element to its correct place. Compare the root element with its children, who live at indices (2 * i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



Our friend the face is back!

Part 2: Heap PQ

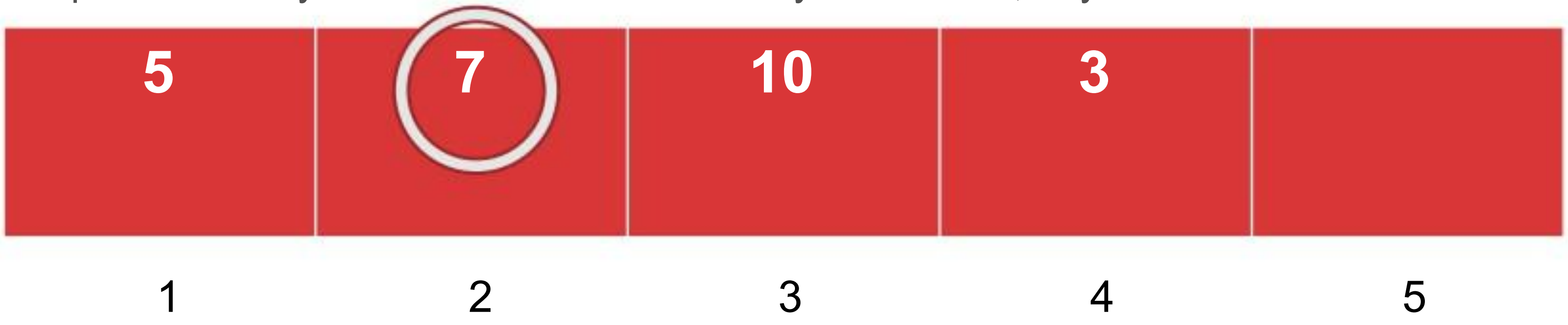
- Let’s talk about dequeue()!
  - To start, **swap** your **first** and **last** elements and reduce your size by 1 (you could also just overwrite root!)
  - Next, you want to **bubble down** the root element to its correct place. Compare the root element with its children, who live at indices (2 * i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



Our friend the face is back!

Part 2: 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), 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 2: 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), 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!

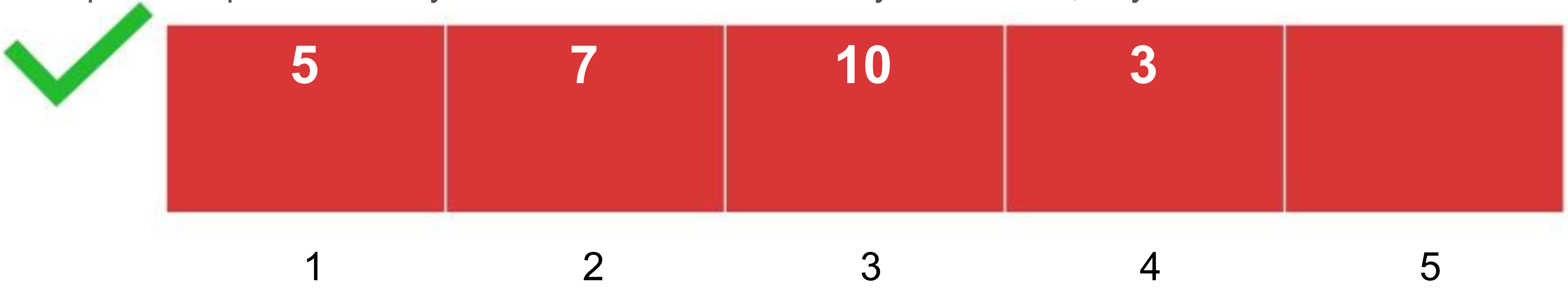
Are we done?

5	7	10	3	
1	2	3	4	5

Part 2: 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), 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!



Part 2: Heap PQ

Helpful hints:

- Like the other parts of this assignment, you'll be using the **DataPoint** struct to represent elements.
- Read thru **`HeapPQueue.h`** before writing any code! The .h file will tell you exactly what is expected of each method you write!
- You will need to **resize** your array if you try and **`enqueue()`** and element that you don't have room for!
- Once you think your enqueue and dequeue functions work, run the provided time tests to verify that they run in O(nlogn) (i.e. enqueueing and/or dequeueing ***n*** elements should take log(n) time per element, where ***n*** is roughly the size of the queue).

Part 2: Heap PQ

Helpful hints:

- I recommend writing a **`swap`**`()` method and explicit **`bubbleUp()`** and **`bubbleDown()`** helper functions.
- **`dequeue()`** is a little more heap-y than **`enqueue()`**, so I'd recommend doing **`enqueue()`** first to get your feet wet!
- Don't worry about ties - swapping identical elements effectively does nothing.
  - Verify to yourself - why is this true?
- The **`printDebugInfo`**`()` method can be a life-saver, but it isn't implemented. You'll have to write them yourself!

Part 2: Heap PQ

Helpful hints:

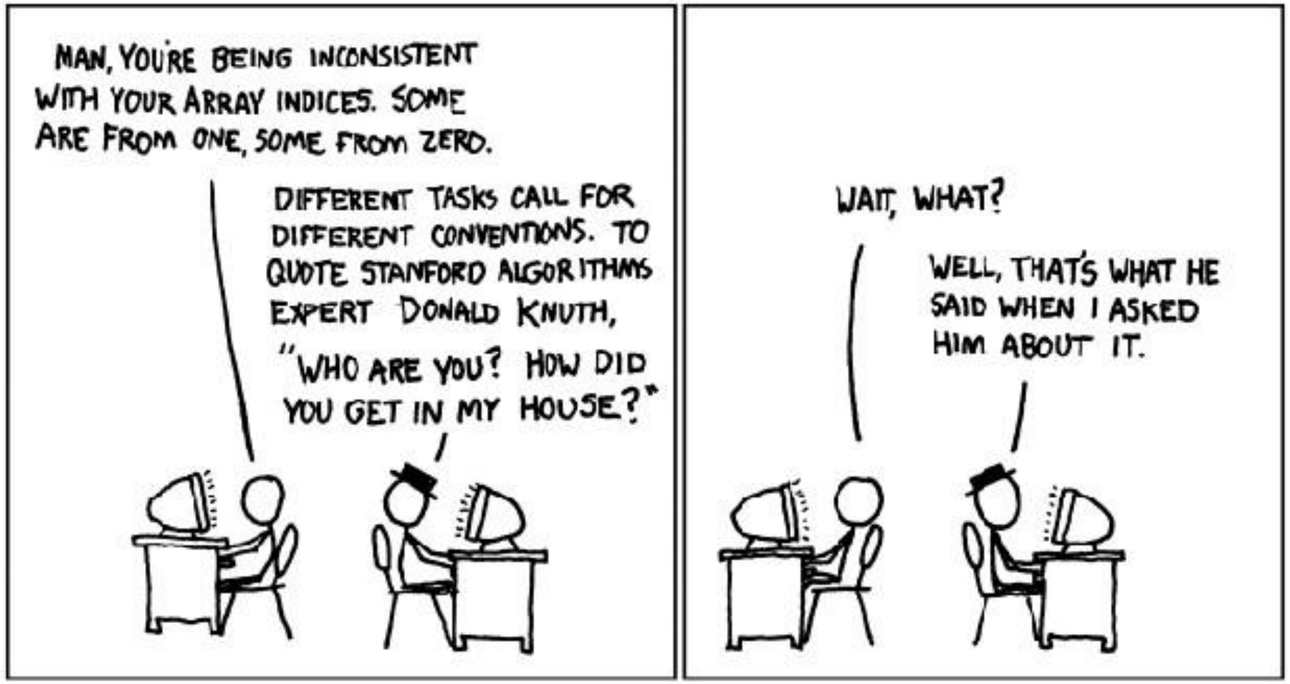
- Verify that the bubble functions work individually before trying to run robustness tests! It can be **very** difficult to locate bugs if they have multiple potential sources.
- Recall the debugging work you did in the first parts of this assignment to help you here - we strongly encourage that you 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 2: 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 2?



Part 3:Top K

A PQSortedArray is just another kind of PQ (you won't need to worry about it). Just assume it's a priority queue that works just like your PQHeap!

- 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.

```
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 3:Top K

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

Part 3:Top K

- 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 3:Top K

- 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 3:Top K

- 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?
- This function will need to run in O(nlogk) time for ***n*** elements in the stream and ***k*** top elements. **Given that your PQ add/removal functions run in O(logk) for a size of k**, what might this imply?

- It's worth noting that you can only view an element from the input stream once. You should never need to revisit it.

Part 3:Top K

Tips / Tricks

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

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

- 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.

Questions about Top K?

Part 4: Extra Demos!

- You don't have to do any extra coding here! Once your program is done, try running the provided demos 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!