# YEAH A4: Priority Queue

CS106B Summer '21

Jin-Hee Lee, Grant Bishko, Lauren Saue-Fletcher

## First of all...

CONGRATULATIONS on finishing your first Stanford mid-quarter diagnostic!

This is not easy stuff, and getting through it is an accomplishment in itself.

We've come so far since we started with smarterSum() :)

## Everyone's Favorite: *Logistics*

Assignment 4 is due Tuesday, July 27 at 11:59 pm PDT

The grace period extends until Thursday, July 29 at 11:59 pm PDT

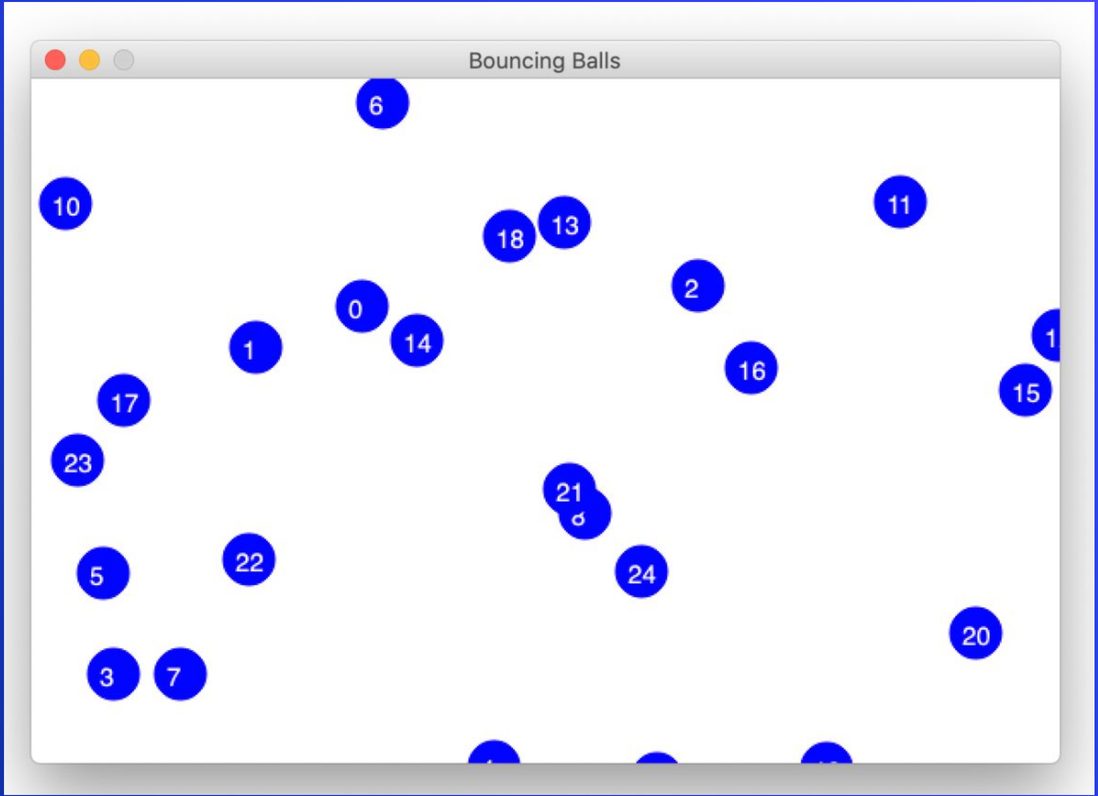
## Assignment 4

1. Warmup
2. PQ Sorted Array
3. PQ Client
4. PQ Heap

## Assignment 4

1. **Warmup**
2. PQ Sorted Array
3. PQ Client
4. PQ Heap

# Warmup



Bouncing Balls!
Fun!

## Warmup

In this first part of the warmup, we'll learn how to examine **member variables** in the debugger pane.

This will look just like any other variable in the pane -- and we've already had lots of practice looking at these!

## this

What is `this`?

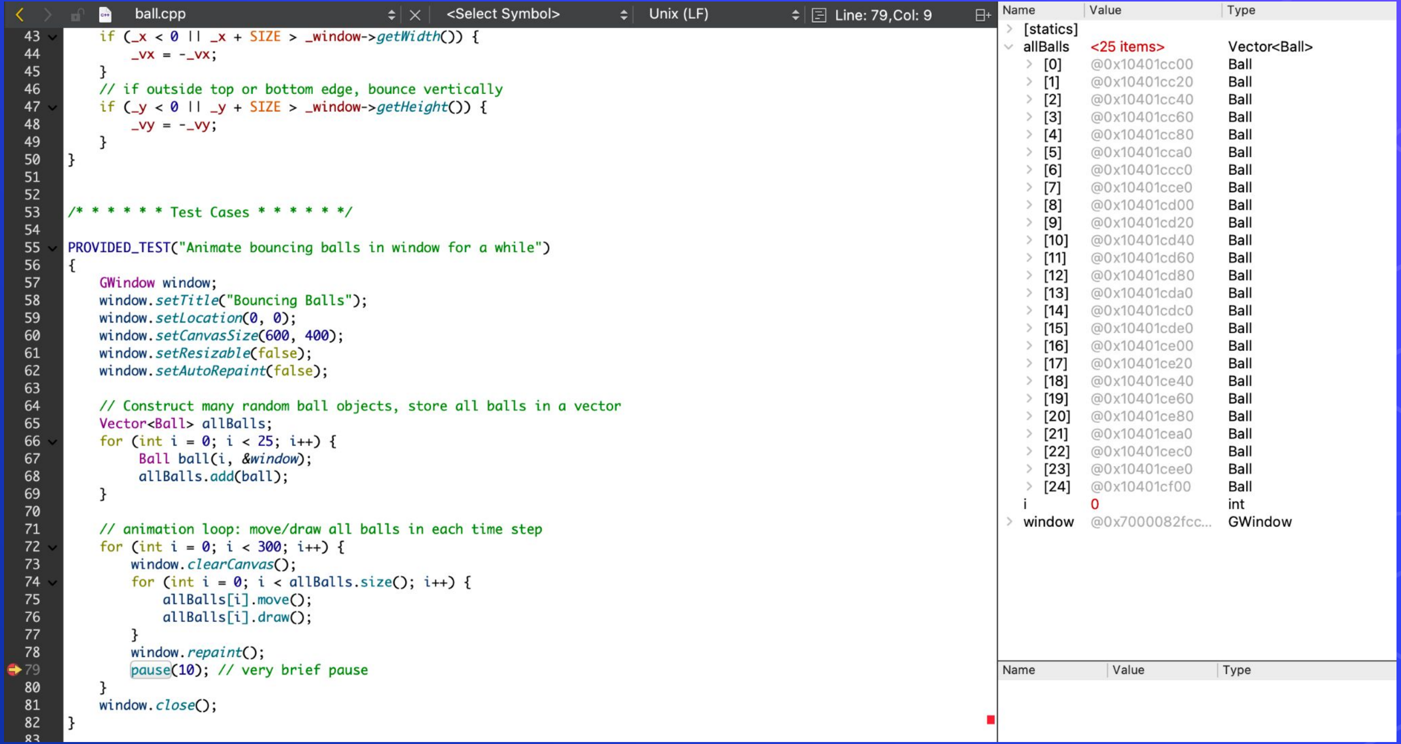
(You'll find that `this` appears in the debugging variables pane.)

`this` is a special variable that refers to the current object that is executing the member function.

You can expand `this` (as you can any other variable) and poke around to see the details of `this` Ball!

# Warmup

Poking around here



ball.cpp <Select Symbol> Unix (LF) Line: 79,Col: 9

```cpp
    if (_x < 0 || _x + SIZE > _window->getWidth()) {
        _vx = -_vx;
    }
    // if outside top or bottom edge, bounce vertically
    if (_y < 0 || _y + SIZE > _window->getHeight()) {
        _vy = -_vy;
    }
}


/* * * * * * Test Cases * * * * * */

PROVIDED_TEST("Animate bouncing balls in window for a while")
{
    GWindow window;
    window.setTitle("Bouncing Balls");
    window.setLocation(0, 0);
    window.setCanvasSize(600, 400);
    window.setResizable(false);
    window.setAutoRepaint(false);

    // Construct many random ball objects, store all balls in a vector
    Vector<Ball> allBalls;
    for (int i = 0; i < 25; i++) {
         Ball ball(i, &window);
         allBalls.add(ball);
    }

    // animation loop: move/draw all balls in each time step
    for (int i = 0; i < 300; i++) {
        window.clearCanvas();
        for (int i = 0; i < allBalls.size(); i++) {
            allBalls[i].move();
            allBalls[i].draw();
        }
        window.repaint();
        pause(10); // very brief pause
    }
    window.close();
}
```

| Name | Value | Type |
|---|---|---|
| [statics] | | |
| allBalls | <25 items> | Vector<Ball> |
| [0] | @0x10401cc00 | Ball |
| [1] | @0x10401cc20 | Ball |
| [2] | @0x10401cc40 | Ball |
| [3] | @0x10401cc60 | Ball |
| [4] | @0x10401cc80 | Ball |
| [5] | @0x10401cca0 | Ball |
| [6] | @0x10401ccc0 | Ball |
| [7] | @0x10401cce0 | Ball |
| [8] | @0x10401cd00 | Ball |
| [9] | @0x10401cd20 | Ball |
| [10] | @0x10401cd40 | Ball |
| [11] | @0x10401cd60 | Ball |
| [12] | @0x10401cd80 | Ball |
| [13] | @0x10401cda0 | Ball |
| [14] | @0x10401cdc0 | Ball |
| [15] | @0x10401cde0 | Ball |
| [16] | @0x10401ce00 | Ball |
| [17] | @0x10401ce20 | Ball |
| [18] | @0x10401ce40 | Ball |
| [19] | @0x10401ce60 | Ball |
| [20] | @0x10401ce80 | Ball |
| [21] | @0x10401cea0 | Ball |
| [22] | @0x10401cec0 | Ball |
| [23] | @0x10401cee0 | Ball |
| [24] | @0x10401cf00 | Ball |
| i | 0 | int |
| window | @0x7000082fcc... | GWindow |

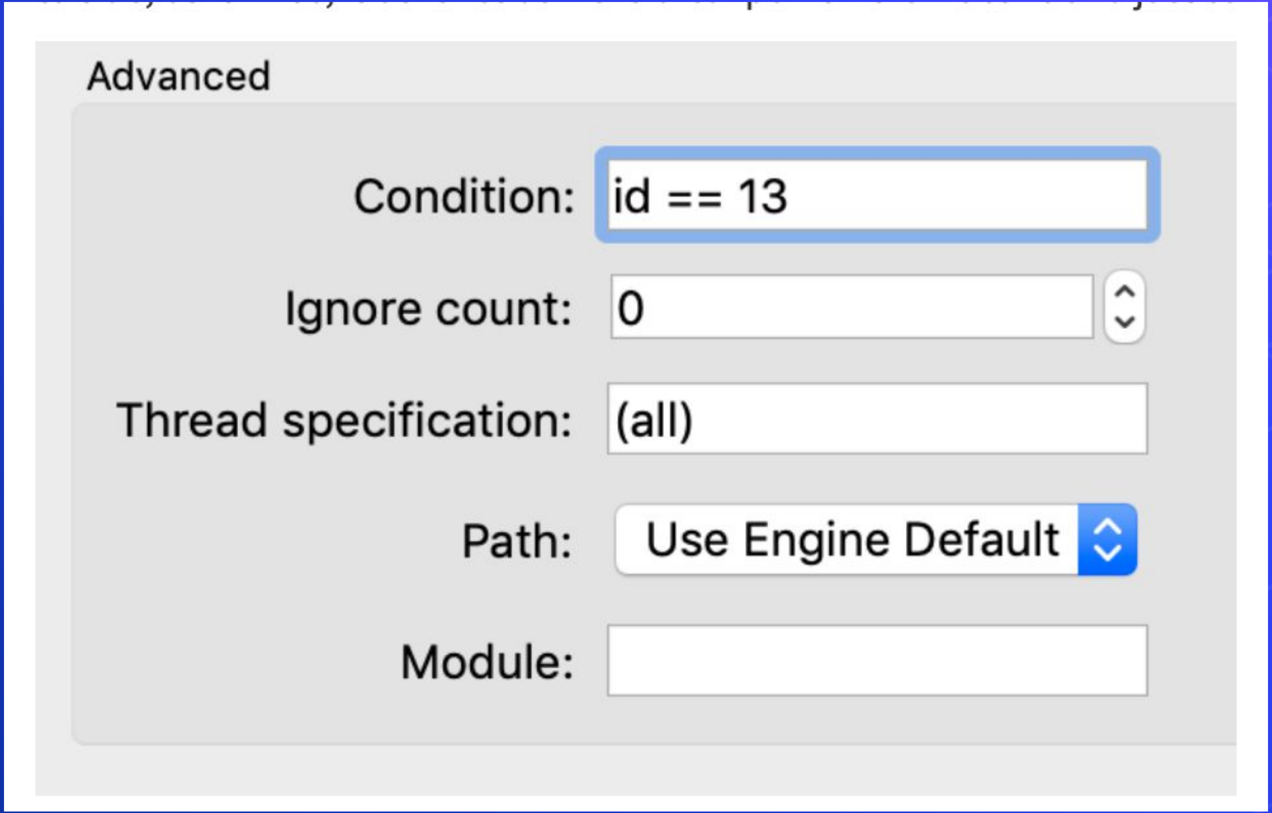
| Name | Value | Type |
|---|---|---|

## Conditional Breakpoints

Control-clicking on the red dot of a breakpoint allows us to **Edit breakpoint**, and set a **Condition**.

From here, you're allowed to specify a **conditional breakpoint**: a breakpoint that will only break if a given condition is true.

# Conditional Breakpoints



Advanced

Condition: id == 13

Ignore count: 0

Thread specification: (all)

Path: Use Engine Default

Module:

## Examining Array Memory

We can change the display format of the variables pane to allow us to examine array memory!

What are the initialized values for elements that we haven't set?

What kind of errors get "caught" -- and in what way?

## Assignment 4

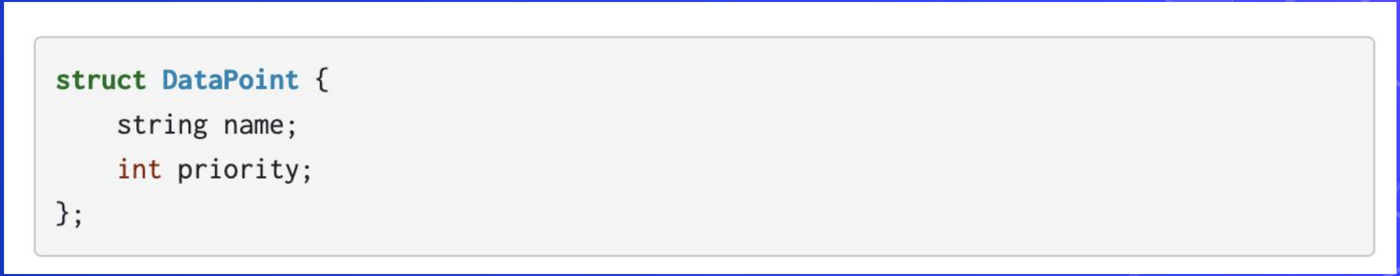
1. Warmup
2. **PQ Sorted Array**
3. PQ Client
4. PQ Heap

# PQ Sorted Array

A **priority queue** is a queue that handles its elements in order of priority.

This means that each element has its own priority.

In our assignment, we'll be using the DataPoint struct:



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

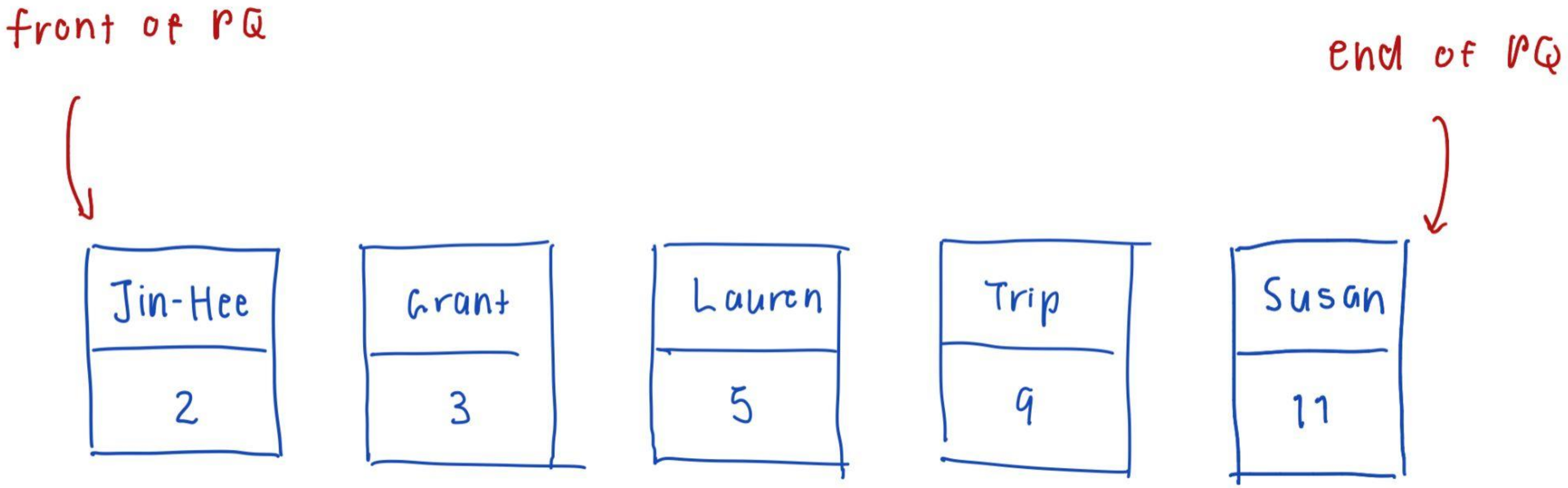
## Priority

IMPORTANT!

In this assignment,

SMALLER INTEGER VALUE -> HIGHER PRIORITY

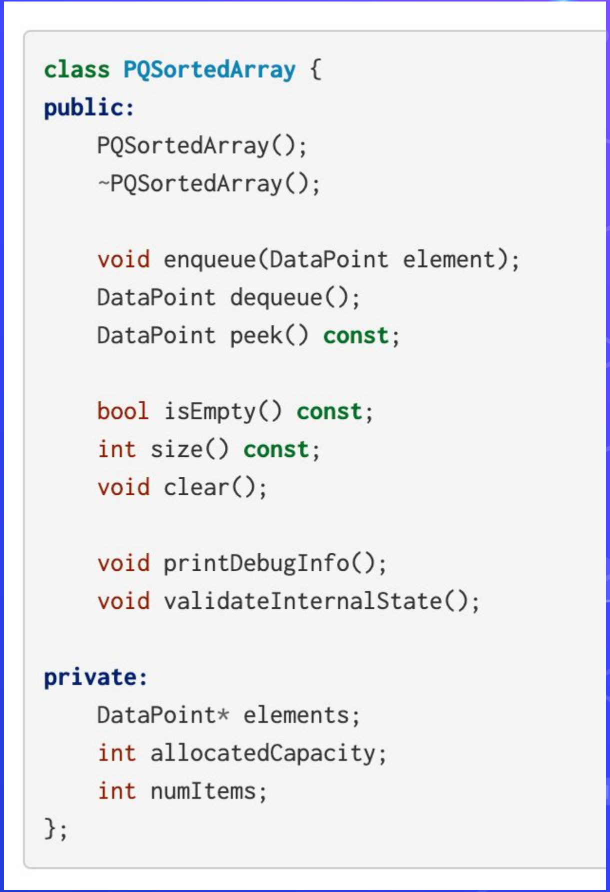
LARGER INTEGER VALUE -> LOWER PRIORITY


front of PQ
end of PQ
Jin-Hee
2
Grant
3
Lauren
5
Trip
9
Susan
11

## Our task…

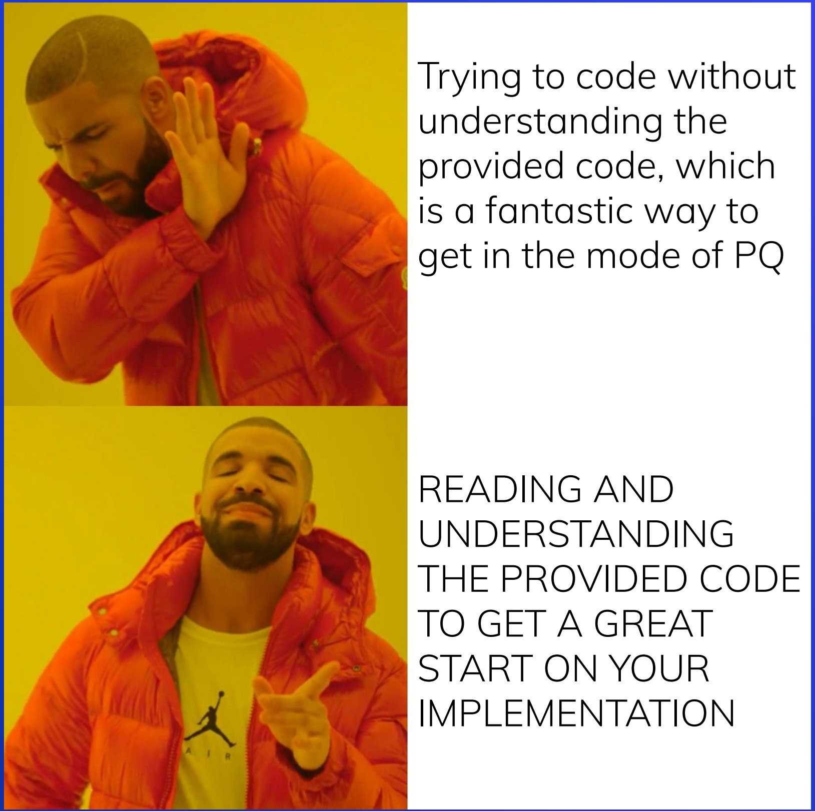
Implement the PQSortedArray class! -------------->

Good news: most of these functions have already been written for you! 😮



```
class PQSortedArray {
public:
    PQSortedArray();
    ~PQSortedArray();

    void enqueue(DataPoint element);
    DataPoint dequeue();
    DataPoint peek() const;

    bool isEmpty() const;
    int size() const;
    void clear();

    void printDebugInfo();
    void validateInternalState();

private:
    DataPoint* elements;
    int allocatedCapacity;
    int numItems;
};
```

## Understanding Code


Trying to code without understanding the provided code, which is a fantastic way to get in the mode of PQ
READING AND UNDERSTANDING THE PROVIDED CODE TO GET A GREAT START ON YOUR IMPLEMENTATION

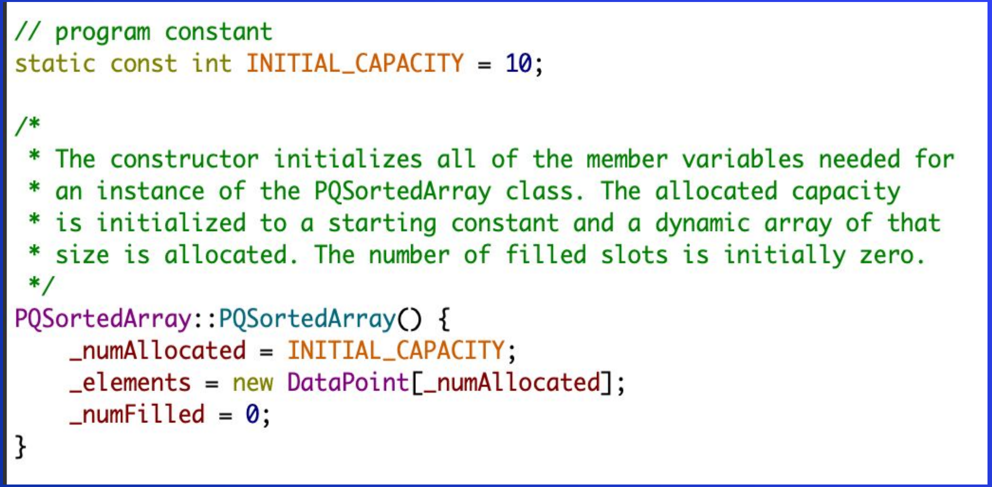
## Understanding Code

For the first time in our assignments, we will be harnessing the power of both the interface **and** the implementation.

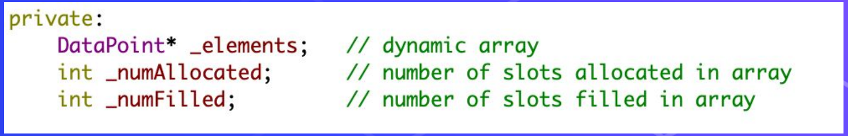
How are the comments different between the .h and the .cpp file? Who is supposed to see and interact with each type of file?

## Understanding Code

*What is the purpose of each variable?*

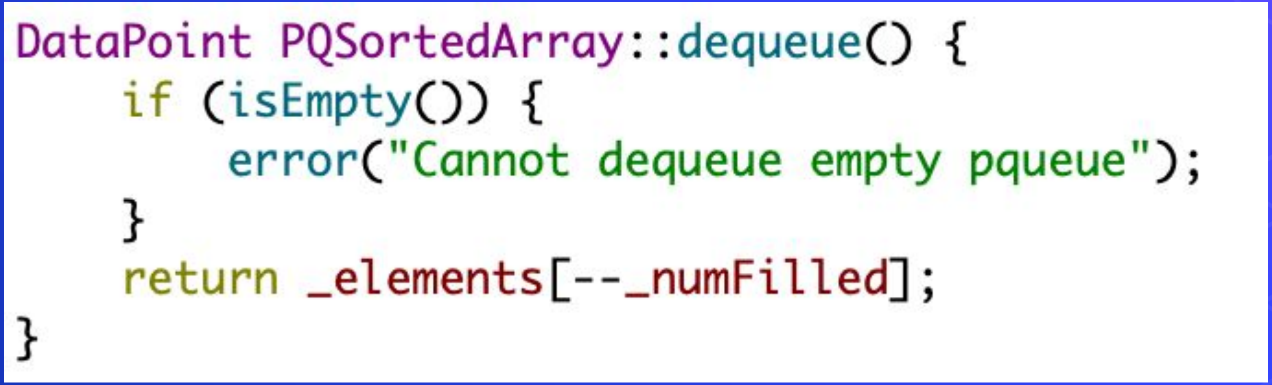


```
// program constant
static const int INITIAL_CAPACITY = 10;

/*
 * The constructor initializes all of the member variables needed for
 * an instance of the PQSortedArray class. The allocated capacity
 * is initialized to a starting constant and a dynamic array of that
 * size is allocated. The number of filled slots is initially zero.
 */
PQSortedArray::PQSortedArray() {
    _numAllocated = INITIAL_CAPACITY;
    _elements = new DataPoint[_numAllocated];
    _numFilled = 0;
}
```



```
private:
    DataPoint* _elements;    // dynamic array
    int _numAllocated;       // number of slots allocated in array
    int _numFilled;          // number of slots filled in array
```

# Understanding Code



```cpp
DataPoint PQSortedArray::dequeue() {
    if (isEmpty()) {
        error("Cannot dequeue empty pqueue");
    }
    return _elements[--_numFilled];
}
```

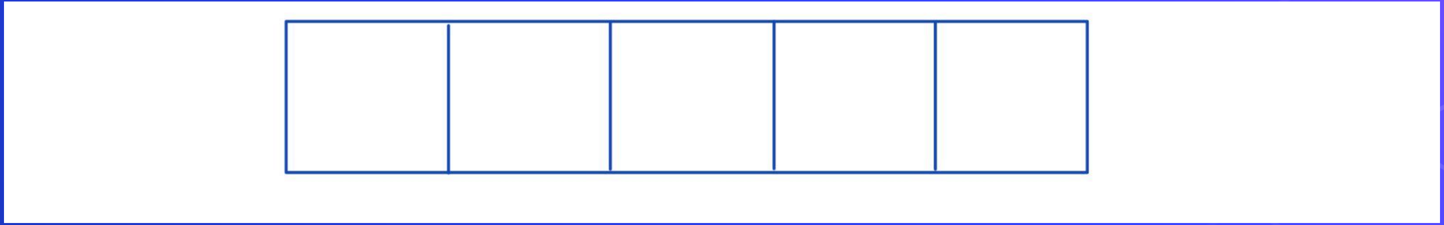
This last line looks kind of funky! Our job is to detangle what each part means.

## Time to get our hands dirty!

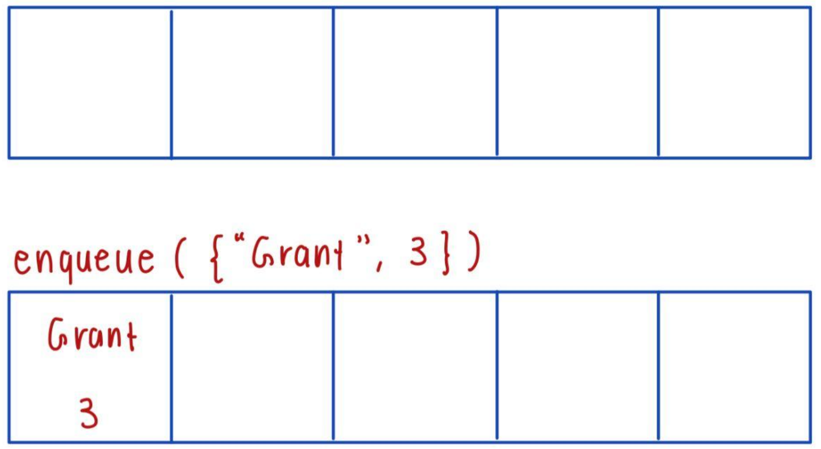
## Enqueue

Let's see how this should work.

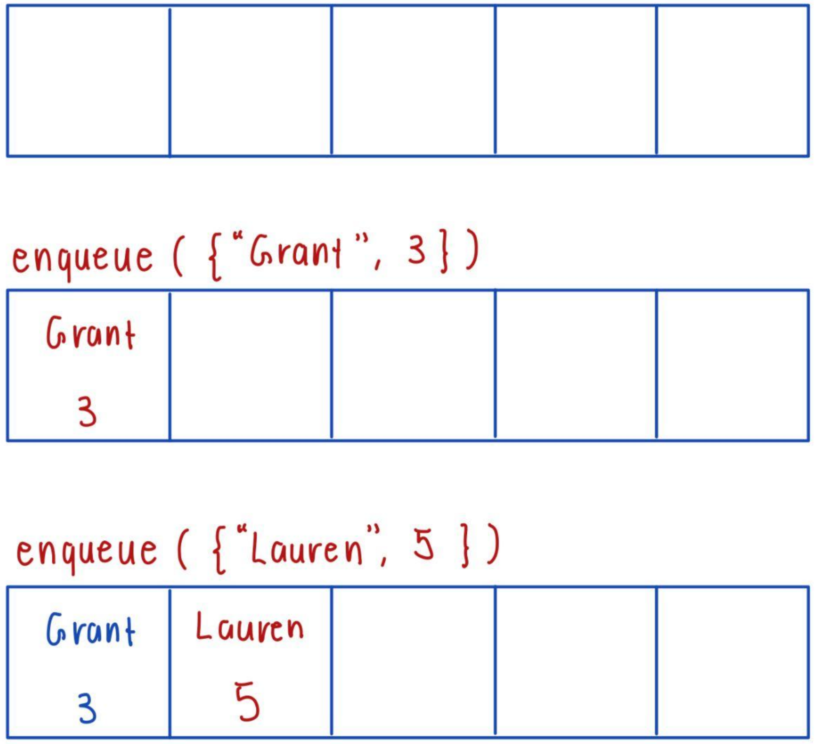
## Enqueue



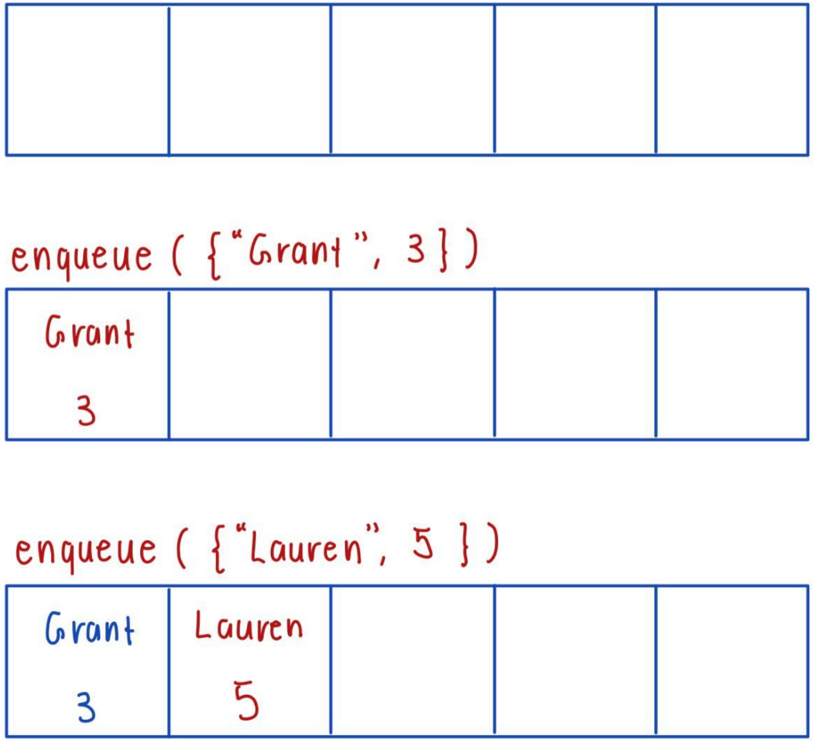
## Enqueue



## Enqueue

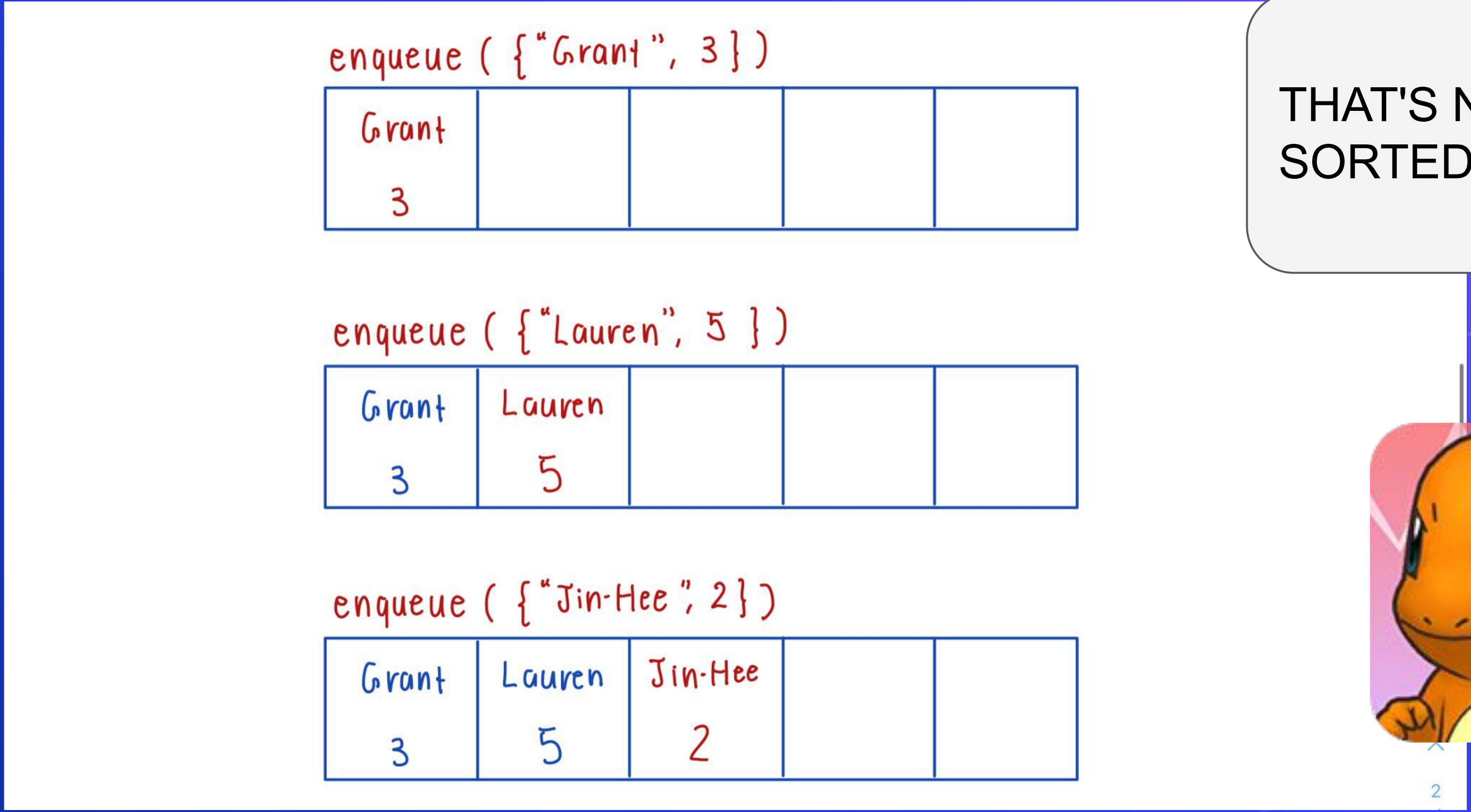


## Enqueue


enqueue ( {"Grant", 3} )
Grant
3
enqueue ( {"Lauren", 5 } )
Grant
3
Lauren
5

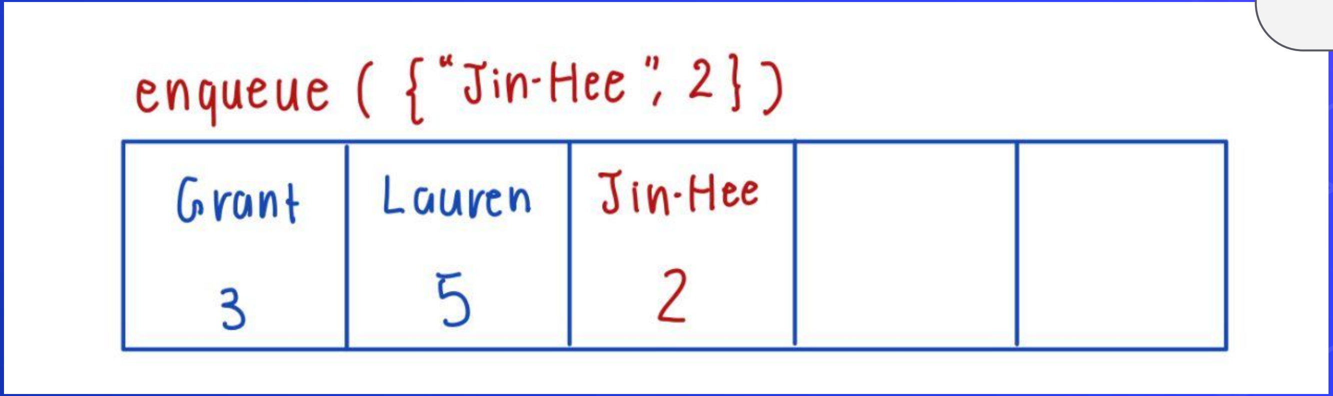
Looks good, the priorities are sorted so far!

# Enqueue

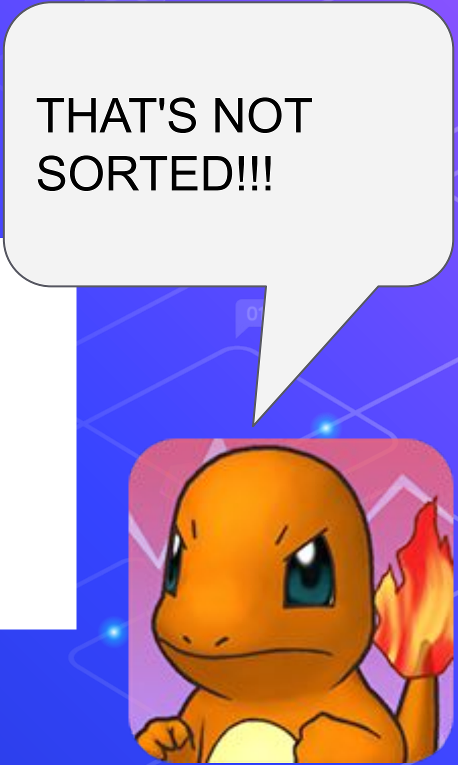


THAT'S NOT SORTED!!!

## Enqueue


enqueue ( {"Jin-Hee", 2} )

|  |  |  |  |  |
|---|---|---|---|---|
| Grant<br>3 | Lauren<br>5 | Jin-Hee<br>2 |  |  |



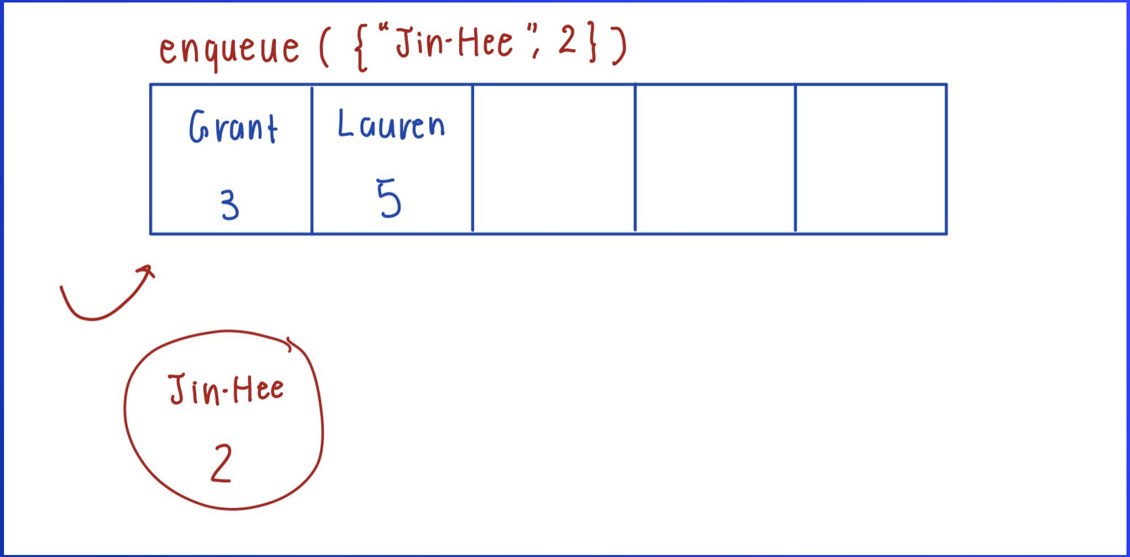
## Great point, char.

We can't just insert at the end, like a regular queue.

Instead, we have to...

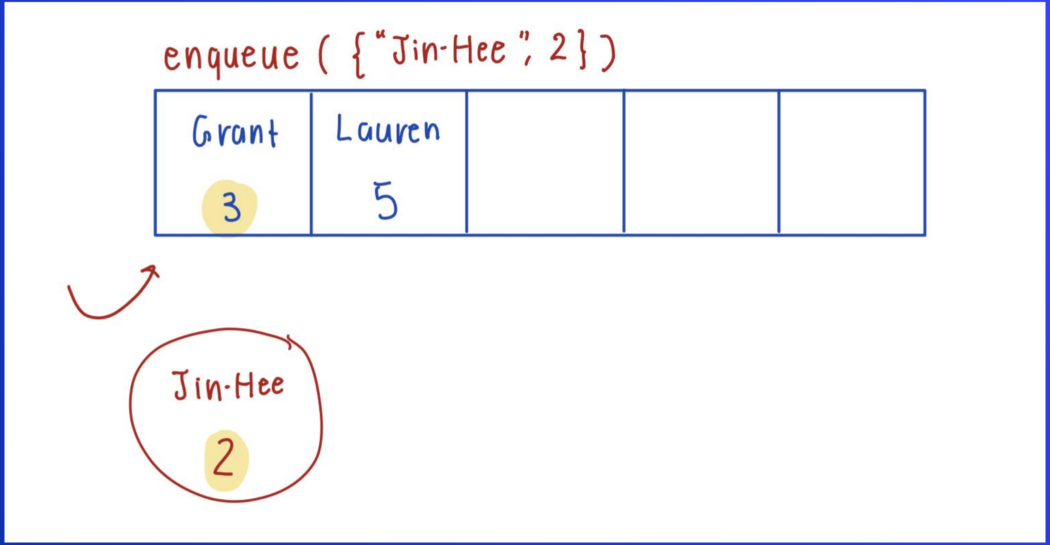
## Steps to Enqueue

1. Loop through the PQ



## Steps to Enqueue

2. Compare the element priorities of the current element and the element you're going to insert

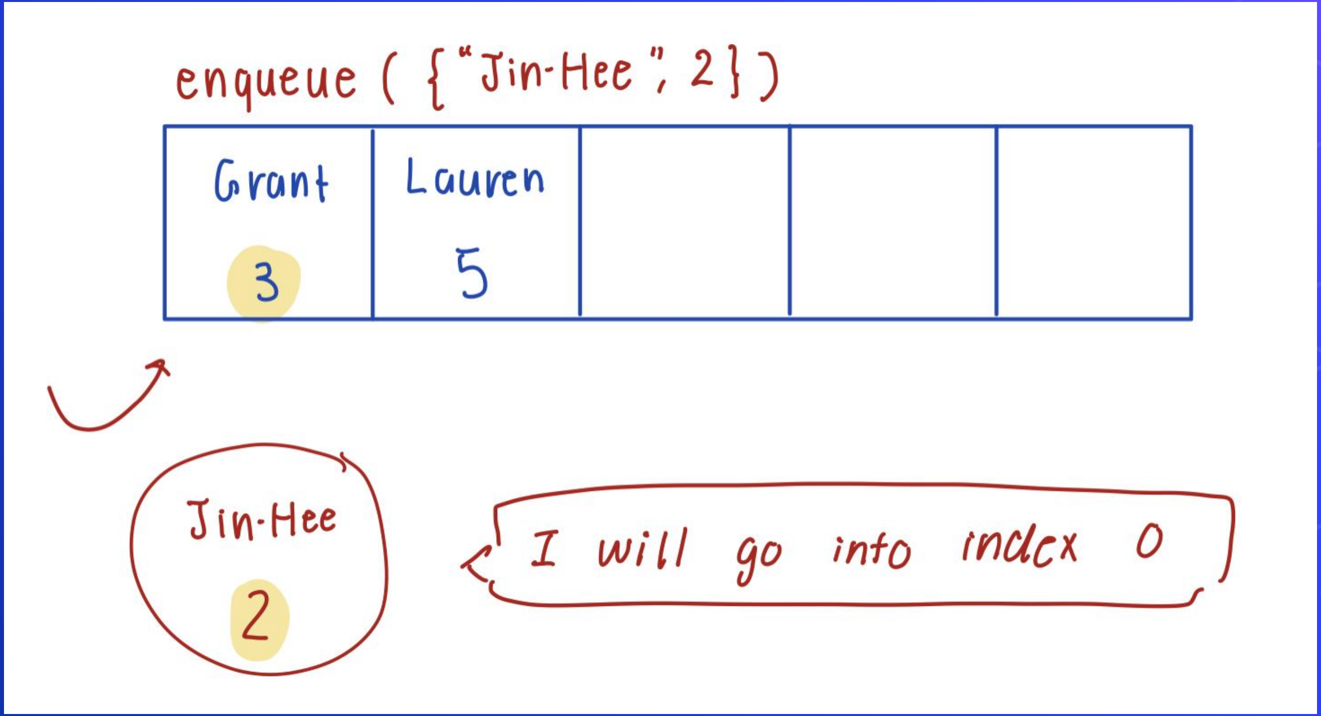


## Steps to Enqueue

3. If you find that your element-to-insert should go right before the current-element-in-PQ, **this is the index in which you'll insert your elem-to-insert!**

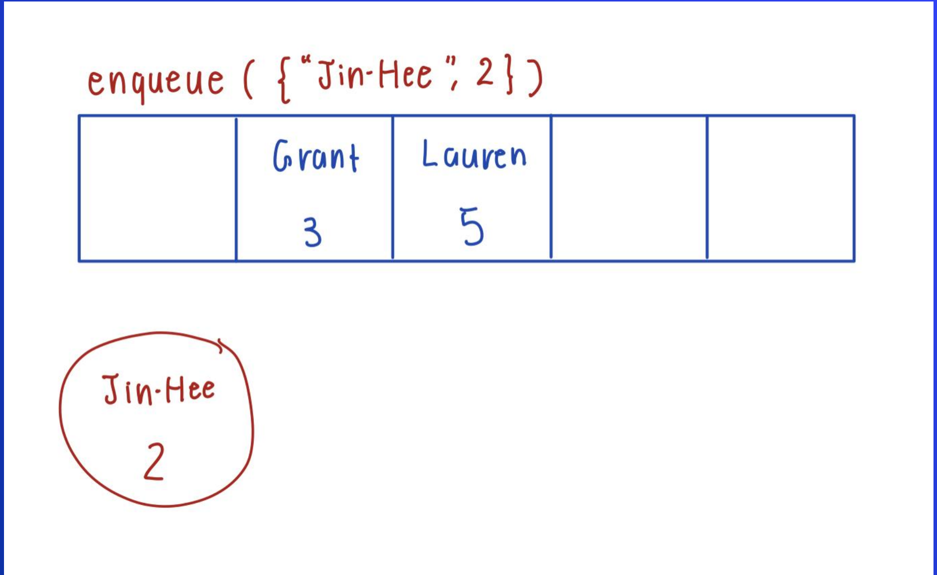
Once you've found this index, there's no need to keep looping.

# Steps to Enqueue



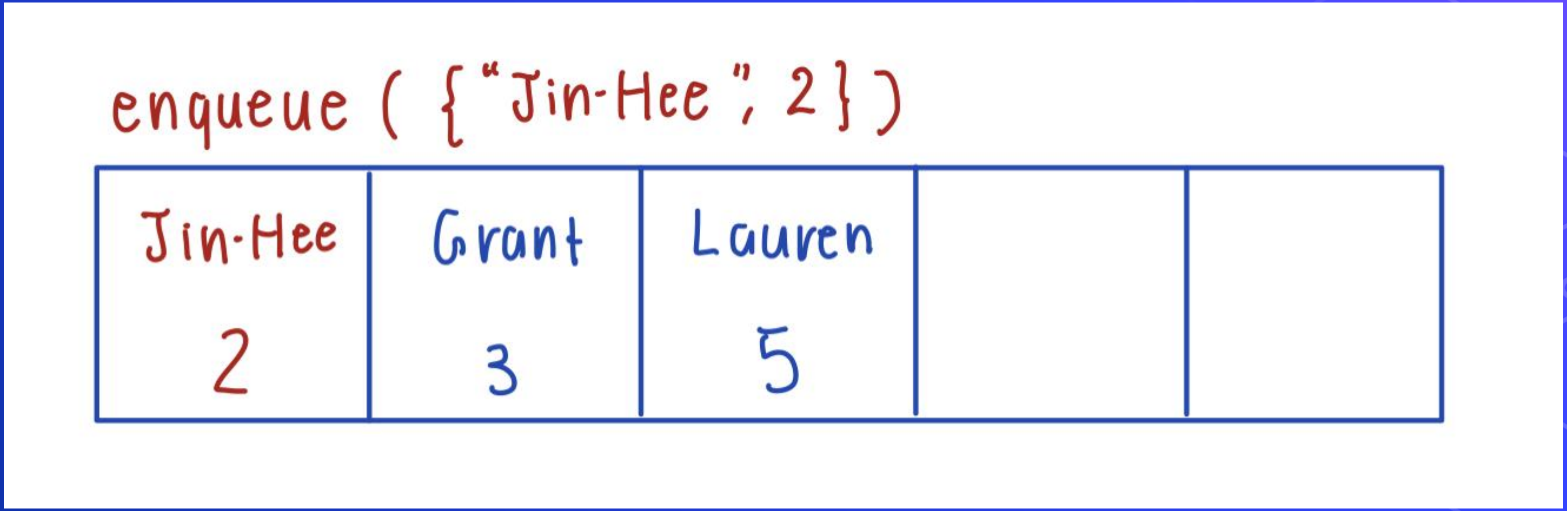
## Steps to Enqueue

4. Scooch all of the necessary elements over by 1 to make space for your new element.



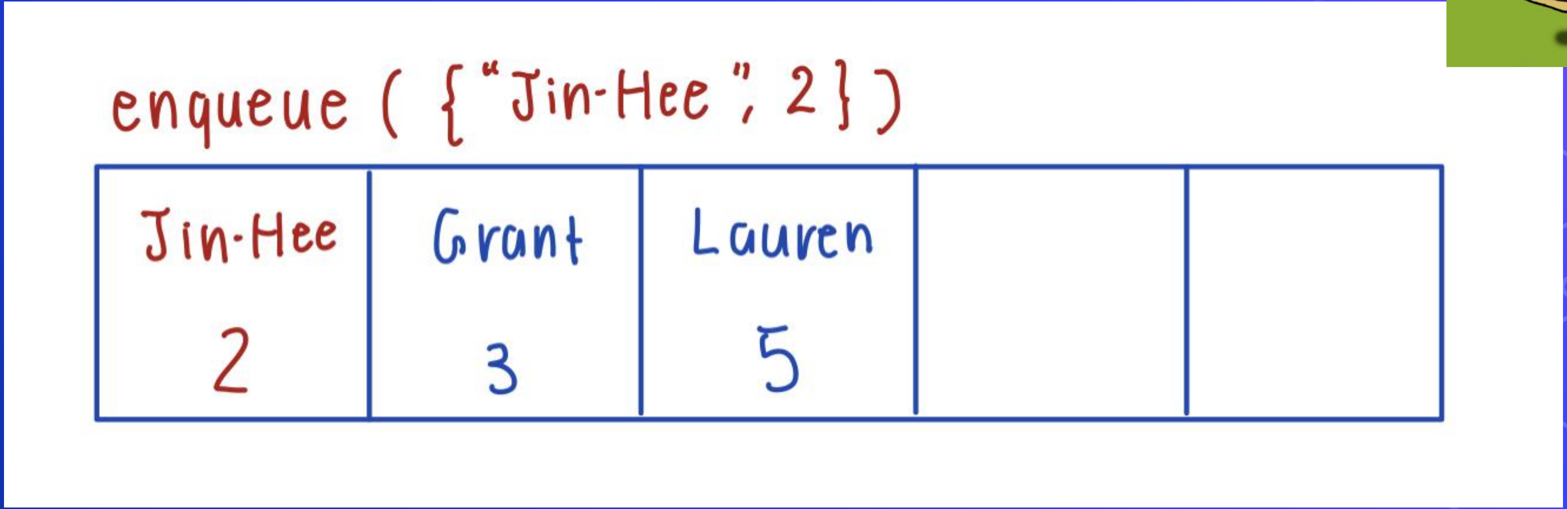
## Steps to Enqueue

5. Add in your element at the index!



## Steps to Enqueue

5. Add in your element at the index!



## TO BE CLEAR:

These steps to enqueue should *always* be how you enqueue (never just add to the back!!!)

The previous example was just drawn out to show us the necessity for the algorithm!

## Things to consider…

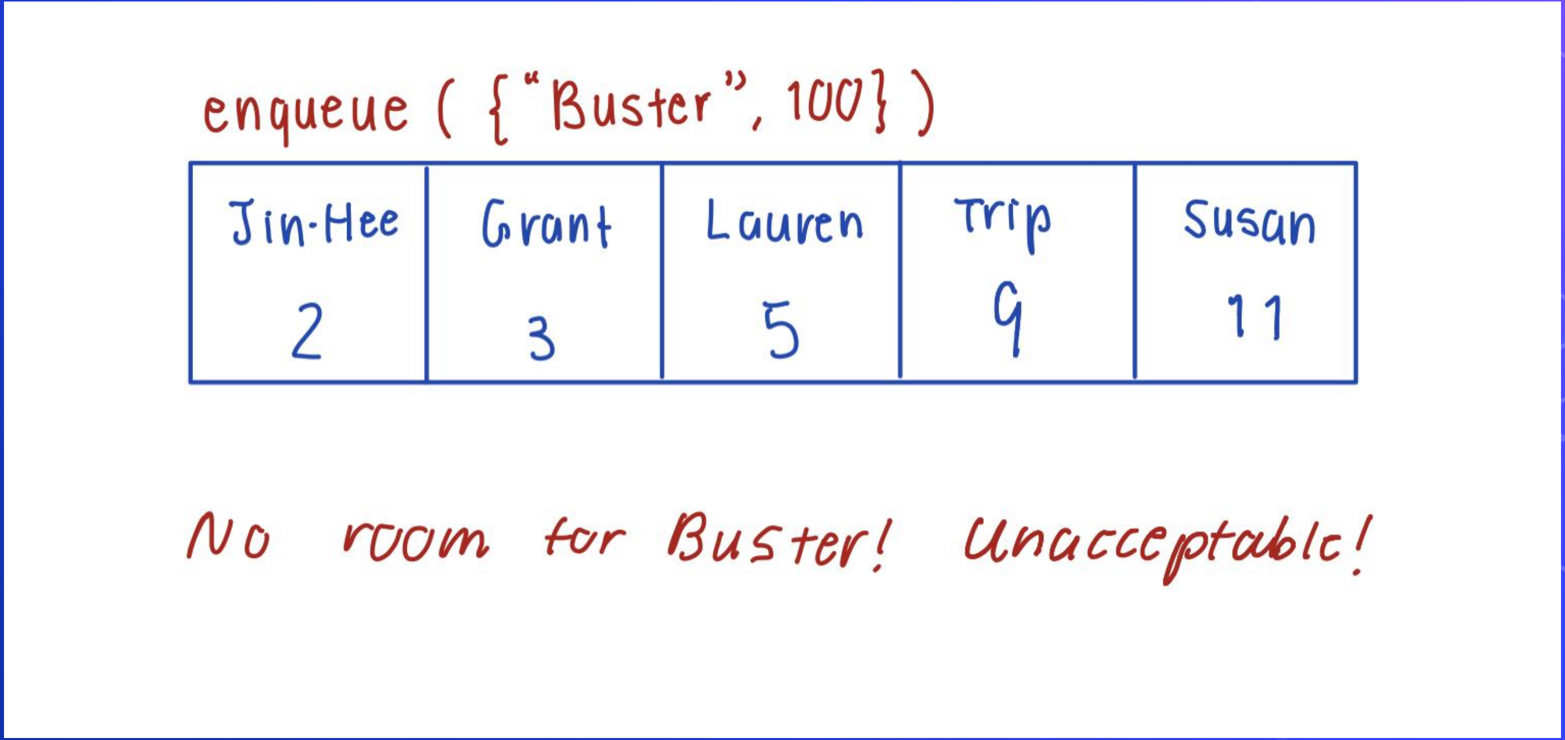
When you enqueue() a new element, do any of your member variables need to be updated?

Can I add as many things as I want? 🤔

## Grow as We Go

Ben Platt - Grow As We Go [Official Video] - YouTube

## Grow as We Go



```
enqueue ( {"Buster", 100} )
```

| Jin-Hee | Grant | Lauren | Trip | Susan |
|---|---|---|---|---|
| 2 | 3 | 5 | 9 | 11 |

No room for Buster! Unacceptable!

## Grow

To handle this, we need the ability to **grow the maximum capacity of our PQ.**

Something to think about: *how can you check if you need to grow? Which member variables do we need to check?*

Let's walk through how to grow our array to twice its size...

Grow

Remember: *we cannot resize dynamically allocated arrays.*

This means we actually need to create a new array if we want one that is 2x bigger.

(Great candidate for a helper function!)

## Make sure that you...

- Copy over all of the elements from the original array
- Responsibly clean up any memory that you are no longer using
- Change the values of any  relevant member variables that have changed as a result of this growing

## Questions about PQSortedArray?

## Assignment 4

1. Warmup
2. PQ Sorted Array
3. **PQ Client**
4. PQ Heap

## Client *who*?

A **client** is someone who will actually use the PriorityQueue that you've implemented.

We will be using the PQ that we created to solve different cool problems.

(Psst: we've done this before using ADTs (like in Assign2!), Now, we're just using our *own* ADT, the PQ.)

## PQ Client: Sort Analysis

We are going to analyze the runtime of **`pqSort()`**.

More good news: we've given you the code here, too! (wow!)

```
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();
    }
}
```

## PQ Client: Sort Analysis

We see that the function calls **`enqueue()`** and **`dequeue()`**.

Now, we just need to consider the runtimes of these 2 functions from our PQ and see how it all comes together!

*How many times is enqueue() called? How many times is dequeue() called? What is the runtime for <u>each</u> of those calls? Hmm...*

## PQ Client: Top K

Goal: Return the top K values in a stream of data in descending order of priority.

```
Vector<DataPoint> topK(istream& stream, int k)
```

^ Top values

^ data stream

^ # to return

## Example:

Stream:

{a, 1} -> {n, 5} -> {h, 3} -> {j, 0} -> {p, 9}

K = 3

1) Top 3 values:
    a) {n, 5}, {h, 3}, {p, 9}
2) Descending **integer** order of priority:
    a) {p, 9}, {n, 5}, {h, 3}

## Stream of data

This is just a way to work with a single element at a time - to avoid storing everything at once.

How to iterate through a stream:

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

## K and Stream size

- Return the most elements up to and including K that you can
- If Stream size >= K
    - Return top K values
- If Stream size < K
    - Return top “Stream size” values
- You can assume K > 0

## Size Constraint O(k)

- Tip: Use runtime and space constraints to help you design your algorithm
- O(k) size => **we can never store more than k elements at a time**
- Check out the handout out for some time constraint tips

## How to keep size O(k):

Let’s say k = 3

So far our largest K values are {3, 10, 5}

If we see a 2 - how do we update our largest 3 values?

Do nothing!

If we see a 11, how do we update our largest 3 values?

We should replace 3 with 11

## Size Constraint O(k)

- Use the intuition on the previous slide to keep within the space constraint
- There's a reason we use a PQ here
    - Take advantage of what we dequeue

## Big O analysis

- The runtime will depend on n and k

1) First give it your best guess!
2) Test it with time trials
    a) First only change n - what effect does it have?
    b) Next only change k - how does the runtime change?

## Questions about PQ Client?

## Assignment 4

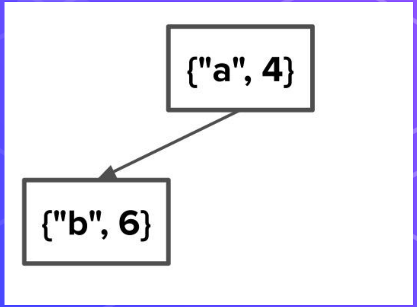
1. Warmup
2. PQ Sorted Array
3. PQ Client
4. **PQ Heap**

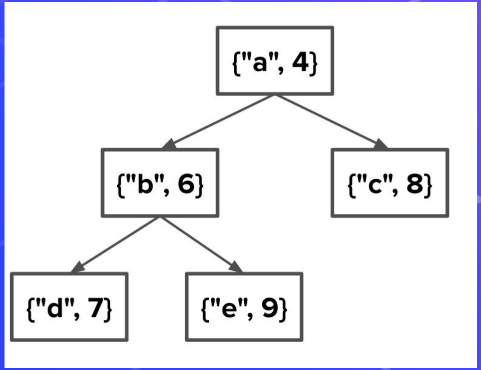
## PQ Heap

- PQSortedArray is kinda slow
- PQHeap is quicker!
- The trick: we store the data better
    - It'll still be in an array but we have a different representation

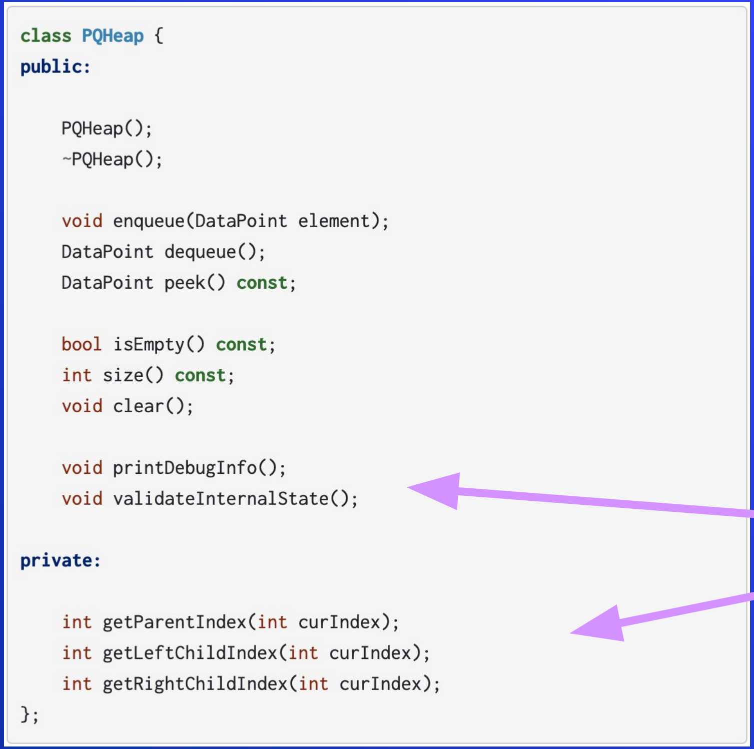
## Min Binary Heap Rules

- Every parent has a higher priority (lower integer value) than its children
- Binary: Two children per parent
- Complete: we fill the heap from left to right, above rows are completely filled





# PQ Heap: Intro to the PQueue Heap Class



```
class PQHeap {
public:

    PQHeap();
    ~PQHeap();

    void enqueue(DataPoint element);
    DataPoint dequeue();
    DataPoint peek() const;

    bool isEmpty() const;
    int size() const;
    void clear();

    void printDebugInfo();
    void validateInternalState();

private:

    int getParentIndex(int curIndex);
    int getLeftChildIndex(int curIndex);
    int getRightChildIndex(int curIndex);
};
```

Same interface (client side) as PQSortedArray

You don't **have to** use these, but it'll make your life easier

## PQ Heap: Intro to the PQueue Heap Class

- Since we are using a min heap - the highest priority, the lowest integer value, will be our root
- We’re storing the min heap in an array- let’s see how:

# PQ Heap: Intro to the PQueue Heap Class

Example:

{ “a”, 0 }

{ “b”, 1 }

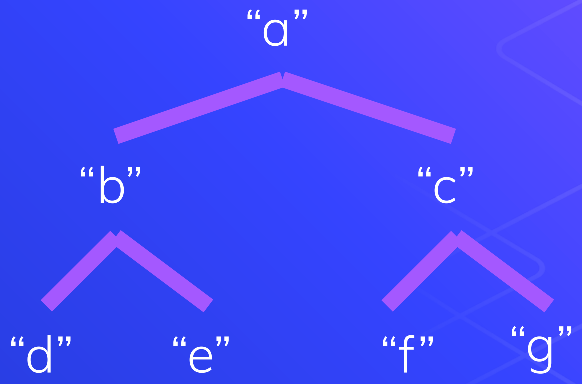
{ “c”, 2 }

{ “d”, 3 }

{ “e”, 4 }

{ “f”, 5 }

{ “g”, 6 }



[ { “a”, 0 }, { “b”, 1 },  “c”, 2 }, { “d”, 3 }, { “e”, 4 }, { “f”, 5 }, { “g”, 6 } ]

## PQ Heap: Intro to the PQueue Heap Class

For the purpose of examples, we will be drawing these binary heaps with the priority numbers rather than elem strings.

## PQ Heap: Validate Internal State

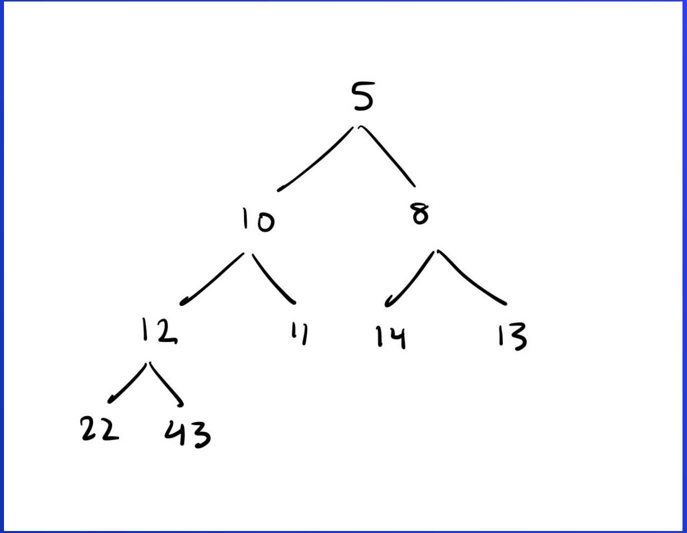
- This is a way to check if your heap is valid
- Optional but highly recommended!! It's hard to otherwise track down which action broke heap ordering
- Think about which approach is easier:
    - Loop through each child, compare to parent
    - Loop through each parent, compare to children

## PQ Heap: Enqueue

1) Add elem to the end of array, updating internal information (like your elem count), resizing if you need!
2) "Bubble up" to rearrange elements as needed
3) Validate internal state to check as you go!

## PQ Heap: Enqueue

{ 5, 10, 8, 12, 11, 14, 13, 22, 43 }

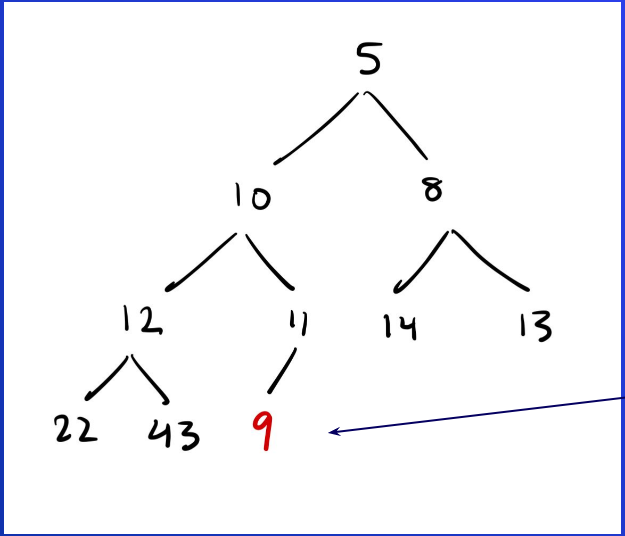


Let's enqueue an elem with a priority **9**.

We have to add it to the first available slot

## PQ Heap: Enqueue

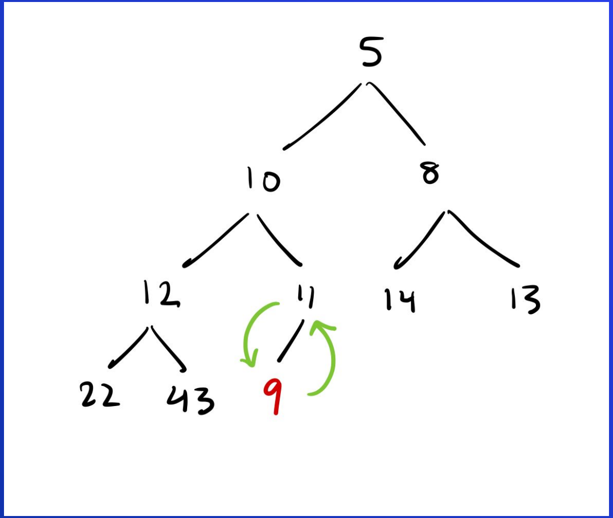
{ 5, 10, 8, 12, 11, 14, 13, 22, 43, **9** }



Here, we added it to our internal array

# PQ Heap: Enqueue -- Bubble Up

{ 5, 10, 8, 12, **11**, 14, 13, 22, 43, **9** }

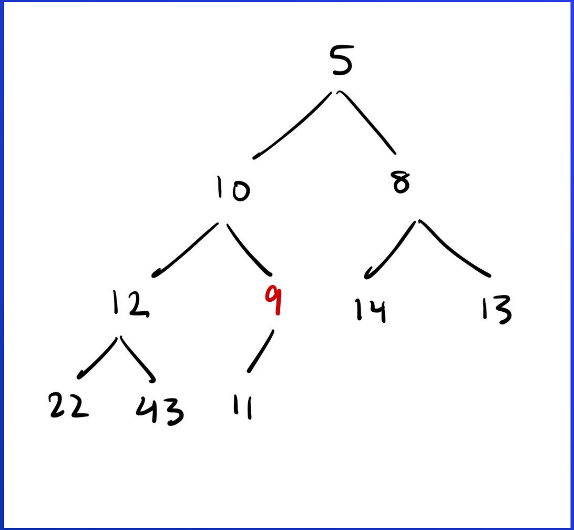


Now, we start the process of bubbling up:

- Compare the priority of the value you just added with the priority of its parent index
- Here, the parent is a *larger priority*, so we **swap** the elems

## PQ Heap: Enqueue -- Bubble Up

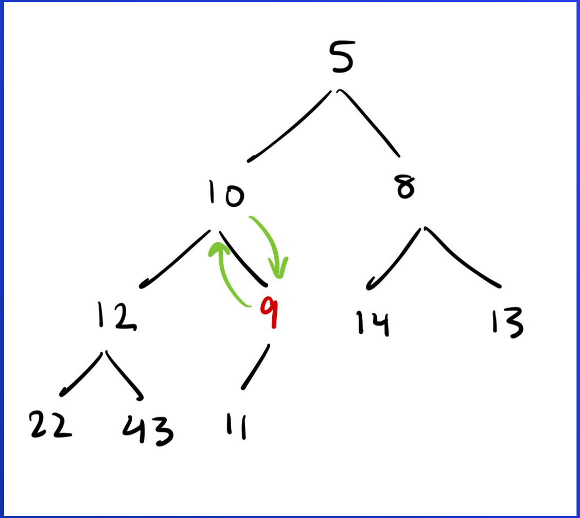
{ 5, 10, 8, 12, 9, 14, 13, 22, 43, 11 }



We continue this process ***until we either reach the top (index 0 of the array) or are no longer smaller than our parent index***

## PQ Heap: Enqueue -- Bubble Up

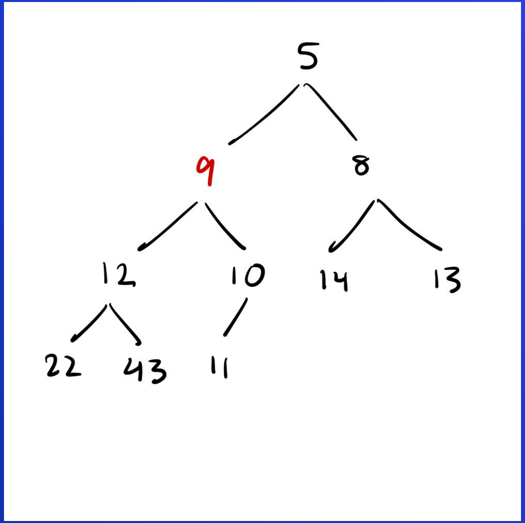
{ 5, **10**, 8, 12, **9**, 14, 13, 22, 43, 11 }



We continue this process ***until we either reach the top (index 0 of the array) or are no longer smaller than our parent index***

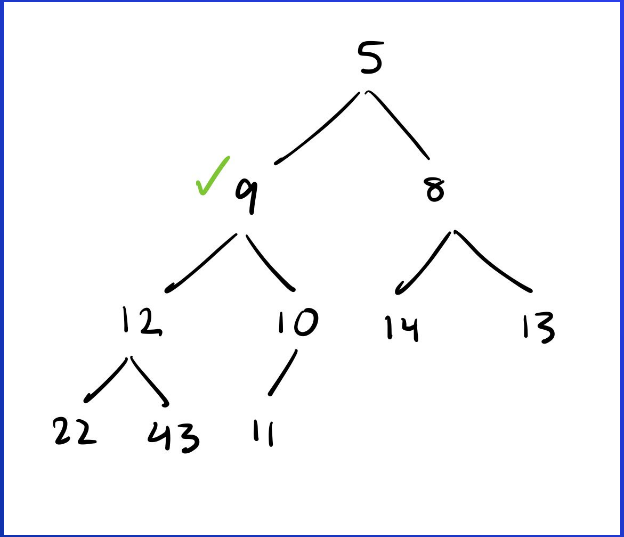
## PQ Heap: Enqueue -- Bubble Up

{ 5, **9**, 8, 12, 10, 14, 13, 22, 43, 11 }



## PQ Heap: Enqueue -- Bubble Up

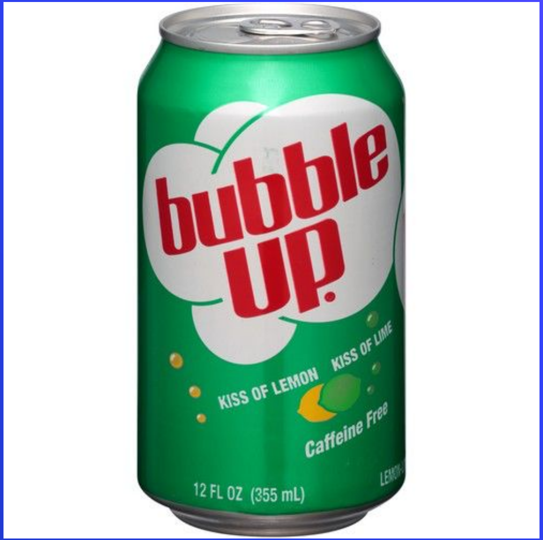
{ 5, 9, 8, 12, 10, 14, 13, 22, 43, 11 }



We are done! Priority 9 >= 5 so it can stay!

Observe the numeric representation of our array above

## Questions about Enqueue/Bubble Up?



Apparently it’s a soda???

## PQ Heap: Recommended helper(s)

- Get parent index
- Get left child index
- Get right child index
- Swap

**Although our Heap is easiest to comprehend drawn out, we are actually dealing with an array of dataPoints. Making helper functions to calculate indexes of parents/children as well as swapping two elements will be SUPER helpful.**

## PQ Heap: Recommended helper(s)

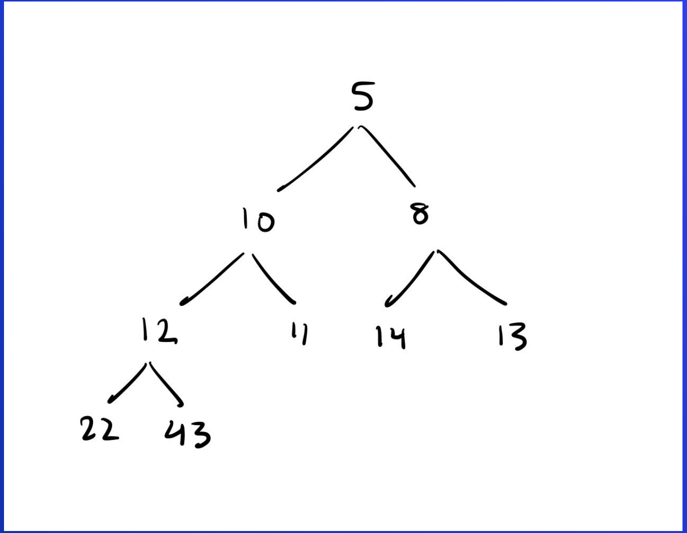
- Resize function

Also recommend making a function that dumps the elements in your current array into a **new** one with **double the size** if you reach your capacity!

You will want to call this in your enqueue function, if you are trying to enqueue an element and are full!

# PQ Heap: Dequeue

{ 5, 10, 8, 12, 11, 14, 13, 22, 43 }



Now, let's work through the process of dequeuing from our PQ

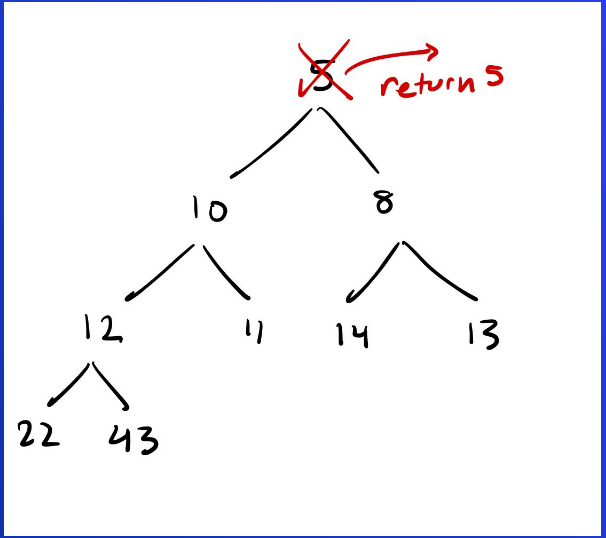
[Remember that just like regular Queues, this will give us the ***front*** elem (5)]

## PQ Heap: Dequeue

1) Swap first elem (to be returned) and last element of array
2) “Bubble down” to rearrange elements as needed
3) Return your elem!

## PQ Heap: Dequeue -- Bubble Down

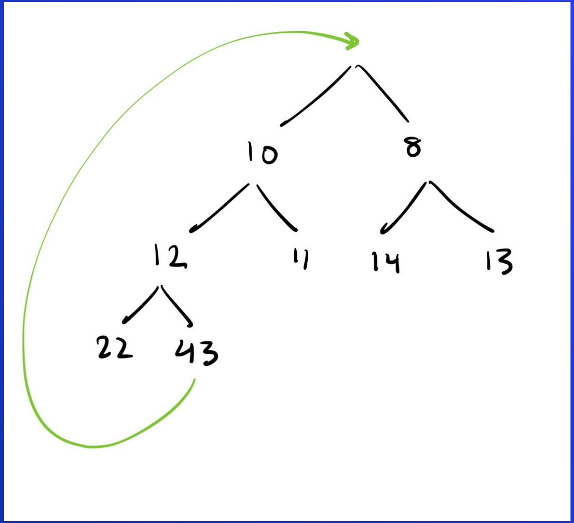
{ ~~5~~, 10, 8, 12, 11, 14, 13, 22, 43 }



Hint: save this value as a variable to return at the end of your function

## PQ Heap: Dequeue -- Bubble Down

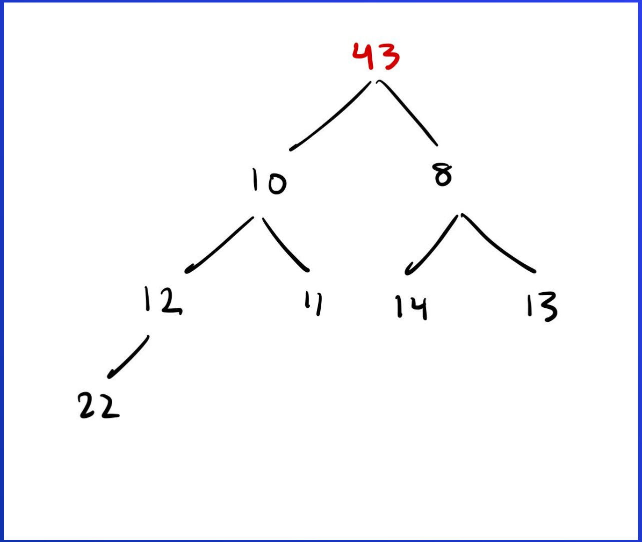
{ **43**, 10, 8, 12, 11, 14, 13, 22, ~~5~~ }



Swap last elem with first elem

## PQ Heap: Dequeue -- Bubble Down

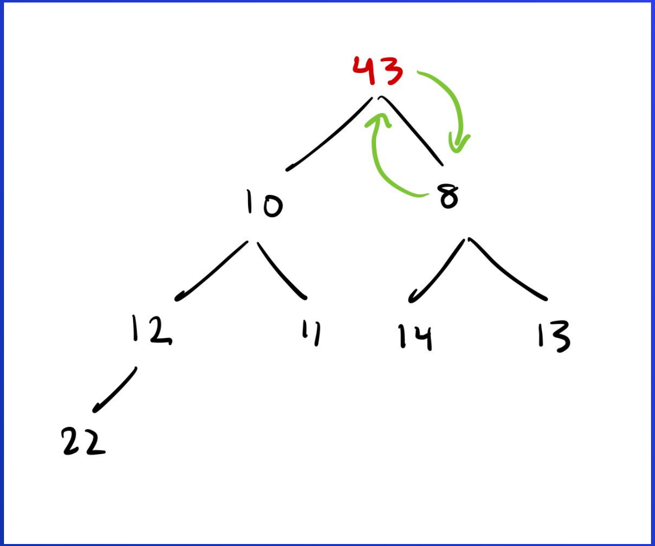
{ **43**, 10, 8, 12, 11, 14, 13, 22, ~~5~~ }



Note how now **43** is at the front of my array (top of my binary heap)

# PQ Heap: Dequeue -- Bubble Down

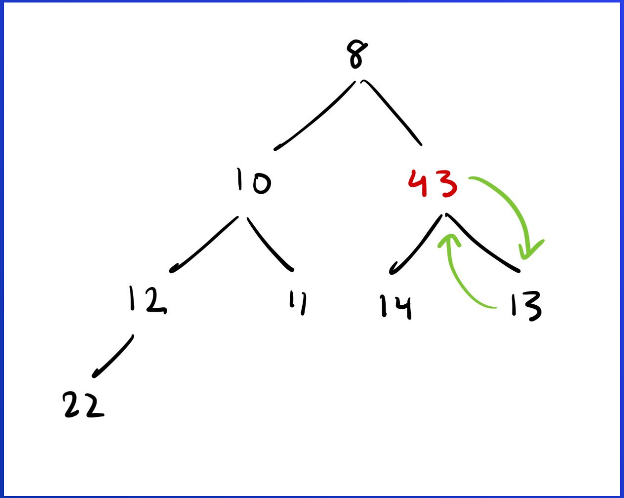
{ **43**, 10, 8, 12, 11, 14, 13, 22, ~~5~~ }



Now we compare the priority of that top element to the priorities of its 2 children, and **swap it with the smaller child**

## PQ Heap: Dequeue -- Bubble Down

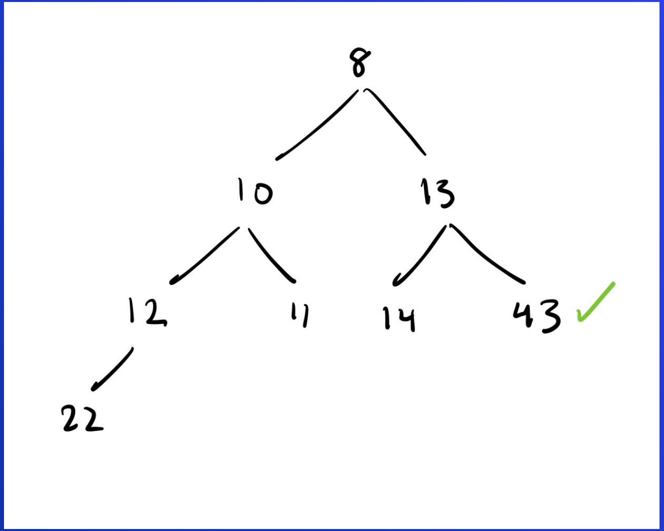
{ 8, 10, **43**, 12, 11, 14, 13, 22, ~~5~~ }



We repeat this process **until we have reached the end of the array** or until our value's priority is no longer larger than its children's priorities

# PQ Heap: Dequeue -- Bubble Down

{ 8, 10, 13, 12, 11, 14, **43**, 22, ~~5~~ }



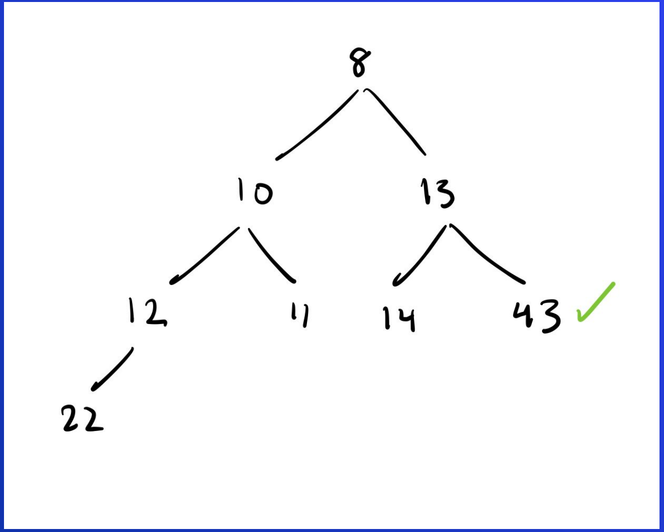
We are done here! In this case, our elem with priority 43 has reached the bottom...

## Questions on Dequeue/Bubble Down?

No soda for bubble down :(

## Quick note:

{ 8, 10, 13, 12, 11, 14, 43, 22, ~~5~~ }



Technically the elem with priority 5 (that we returned) is still at the end of our array. This is okay! When you dequeue make sure you decrement the number of elements in your array that you are internally keeping track of!

## PQ Heap: Biggest Tips

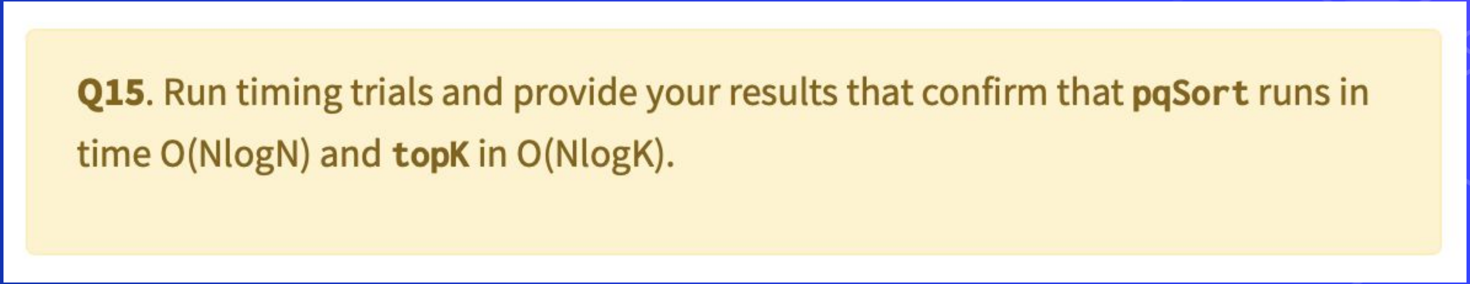
- TEST AS YOU GO
- Array is 0-Indexed
- Note that you are sorting based on PRIORITY (hence the name of our queue)
  - Use dataPoint.priority to see this
  - Lowest priority number is higher priority → confusing... top of binary heap is LOWEST number.

## TIPS: Debugging Memory Leaks/Errors

- Essentially for every **new** you need to **delete**
- Don't free your heap more than once
- Don't try to access memory you don't have access to! (Seg faults!)
- We will deal with this much more in future assignments! Stay tuned! If you are curious about dealing with memory and pointers like this, take CS107!!

## Break the speed barrier

Jump back to your role as client and open up **pqclient.cpp**. Edit the functions **pqSort** and **topK** to use **PQHeap** in place of PQSortedArray, and observe the insane time differences!



**Q15**. Run timing trials and provide your results that confirm that **pqSort** runs in time O(NlogN) and **topK** in O(NlogK).

## Data Analysis Demos and Ethics!

For the final part of the assignment, you will be asked to use your code to analyze real life data and answer embedded ethics questions!

## Final questions?

Go forth!
Heap that Queue!