# Programming Abstractions

CS106B

Cynthia Bailey
Chris Gregg

pollev.com/cs106b

# Today's Topics

- First advanced data structure implementation!
  - More practice with classes + dynamic memory
  - Good skills to practice for A5 homework that goes out tomorrow!

- Apply to be a section leader! Applications due Saturday Nov 2.
  - Next quarter too busy to start? Fun fact: you can apply now and interview, and if accepted, defer to Spring!

- **For important announcements, be sure to see the weekly announcements post on the Ed Q&A board! https://edstem.org**
- **Also on Ed: live lecture Q&A with Chris & Jonathan**

## *Previously:*

- Stack implementation using dynamically-allocated array
  - Pointers, `new` and `delete`
  - Array doubling when capacity is exceeded
  - Inserting and deleting elements from an array
- Big-O analysis

# Today's Agenda:

- Priority Queue ADT
- Two "starter" implementations that build on our array skills
  - Sorted array
  - Unsorted array
  - Performance analysis
- Binary heap data structure implementation
  - What are binary heaps?
  - How do we do enqueue in a heap?

# Recall what we saw with `Vector` `insert()`

_elems:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 7 | 8 | 13 | 25 | 29 | 33 | 51 | 89 | 90 | | | |

_size: 10
_capacity: 13

- Because memory is contiguous, all elements must scoot over to make room for an inserted element
  - For example, insert 10

# Recall what we saw with `Vector insert()`

_elems:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 7 | 8 | 13 | 25 | 29 | 33 | 51 | 89 | 90 | | | |

_size: 10
_capacity: 13

- Because memory is contiguous, all elements must scoot over to make room for an inserted element
  - For example, insert 10

# Recall what we saw with `Vector insert()`

`_elems:`

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 7 | 8 | 10 | 13 | 25 | 29 | 33 | 51 | 89 | 90 | | |

`_size: 11`
`_capacity: 13`

- Because memory is contiguous, all elements must scoot over to make room for an inserted element
  - For example, insert 10

# Priority Queue ADT

- Purpose: we need to access items in order of priority
- Requirements
  - The **next item** to access or remove is the **highest-priority item**
  - New items may be added at any time
- Common use case or analogy: Hospital Emergency Department
  - Patients are served in order of urgency of their condition, *not* first-come first serve


Longer wait

Next to be served

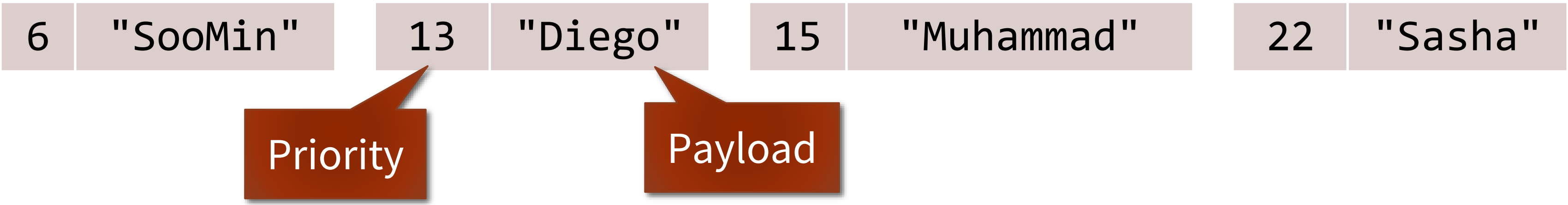
**Low priority** **High priority**

# Today's Agenda:

- Priority Queue ADT
- **Two "starter" implementations** that build on our array skills
  - Sorted array
  - Unsorted array
  - Performance analysis
- Binary heap data structure implementation
  - What are binary heaps?
  - How do we do enqueue in a heap?
  - Homework: dequeue in a heap

# Contents of one element of a Priority Queue

- Individual elements of our priority queue will have two pieces to them:
  - An integer indicating the **priority** of this element
    - We will use smaller number means higher priority, but could be done either way
  - A "**payload**" of whatever the actual element data is
    - Varies based on application, but we'll use **a string** for the patient name



# Two priority queue implementation options

| 0 | | 1 | | 2 | | 3 | | 4 | |
|---|---|---|---|---|---|---|---|---|---|
| 22 | "Sasha" | 6 | "SooMin" | 15 | "Muhammad" | 13 | "Diego" | | |

**Unsorted array**

- **Enqueue:** add new element *at the end of the array*
- **Dequeue:** search for highest-priority item, then remove it

| 0 | | 1 | | 2 | | 3 | | 4 | |
|---|---|---|---|---|---|---|---|---|---|
| 22 | "Sasha" | 15 | "Muhammad" | 13 | "Diego" | 6 | "SooMin" | | |

**Sorted array**

- **Enqueue:** add new element *where it goes* in priority-sorted order
- **Dequeue:** take the last element of the array

## Unsorted array

**Enqueue**

Add new element

*at the end of the array*

**Dequeue**

Search for highest-priority item, then remove it

## Sorted array

**Enqueue**

Add new element *where it goes* in priority-sorted order

**Dequeue**

Take the last element of the array

# Click to mark ALL the operations you think are FAST: O(1)

## Unsorted array

**Enqueue**

Add new element
*at the end of the array*

**Dequeue**

Search entire array for highest-priority item, then remove it

## Sorted array

**Enqueue**

Always add new elements *where they go* in priority-sorted order

**Dequeue**

Take the last element of the array

## Unsorted array

**Enqueue**

Add new element *at the end of the array*

O(1)

**Dequeue**

Search for highest-priority item, then remove it

O(N)—search whole array, *and* items may have to scoot over

## Sorted array

**Enqueue**

Add new element *where it goes* in priority-sorted order

O(N)—items may have to scoot over

**Dequeue**

Take the last element of the array

O(1)

## Unsorted array

**Enqueue**

Add new element *at the end of the array*

O(1)

**Dequeue**

Search for highest-priority item, then remove it

O(N)—search whole array, *and* items may have to scoot over

## Sorted array

**Enqueue**

Add new element *where it goes* in priority-sorted order

O(N)—items may have to scoot over

**Dequeue**

Take the last element of the array

O(1)

# Entirely unsorted is too chaotic, but entirely sorted is too difficult to maintain

O(1)

Is there a “just-right” compromise with the **best of both**?

O(1)

## Entirely unsorted is too chaotic, but entirely sorted is too difficult to maintain

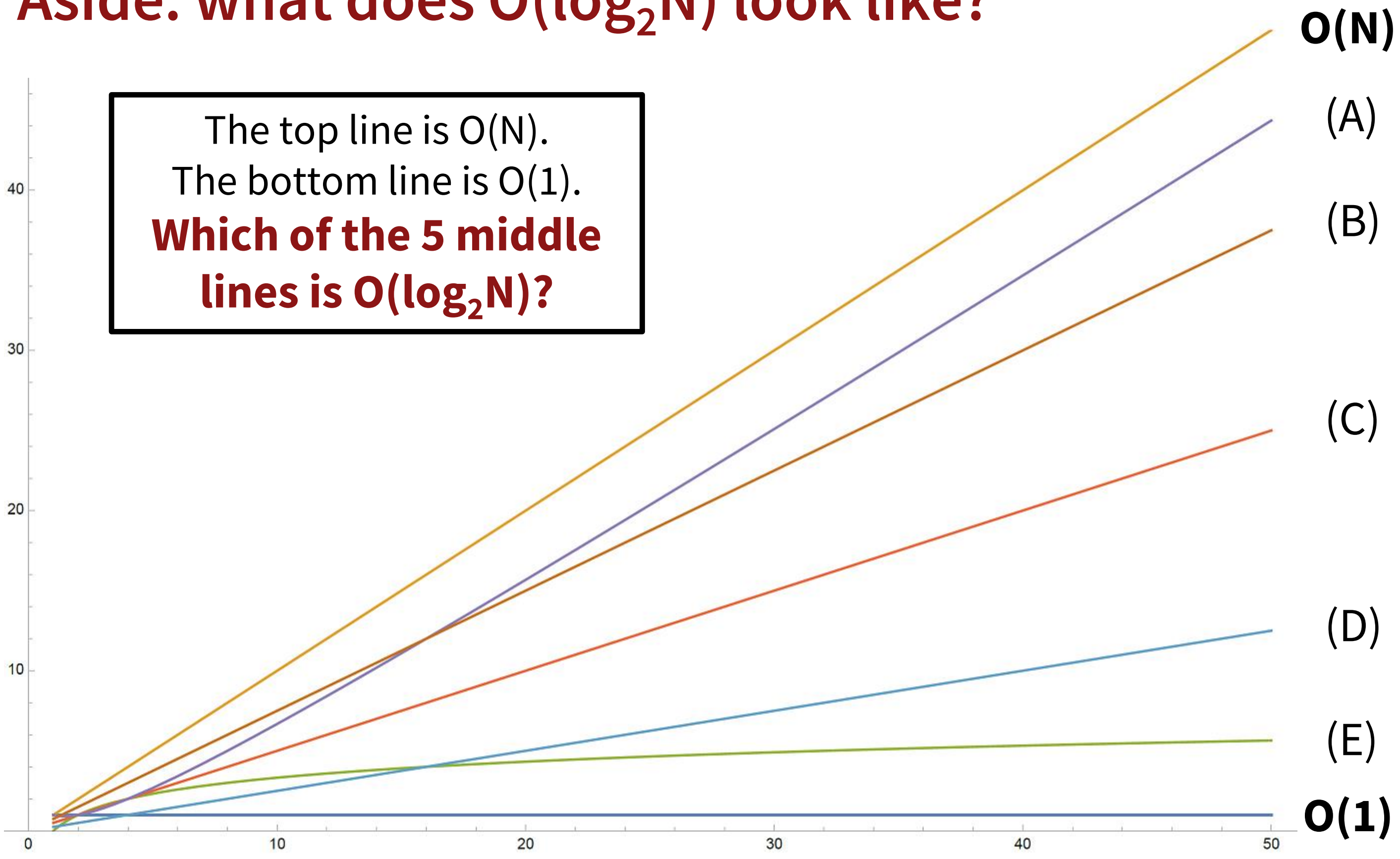
$O(\log_2 N)$

Is there a "just-right" compromise with the **best of both**?

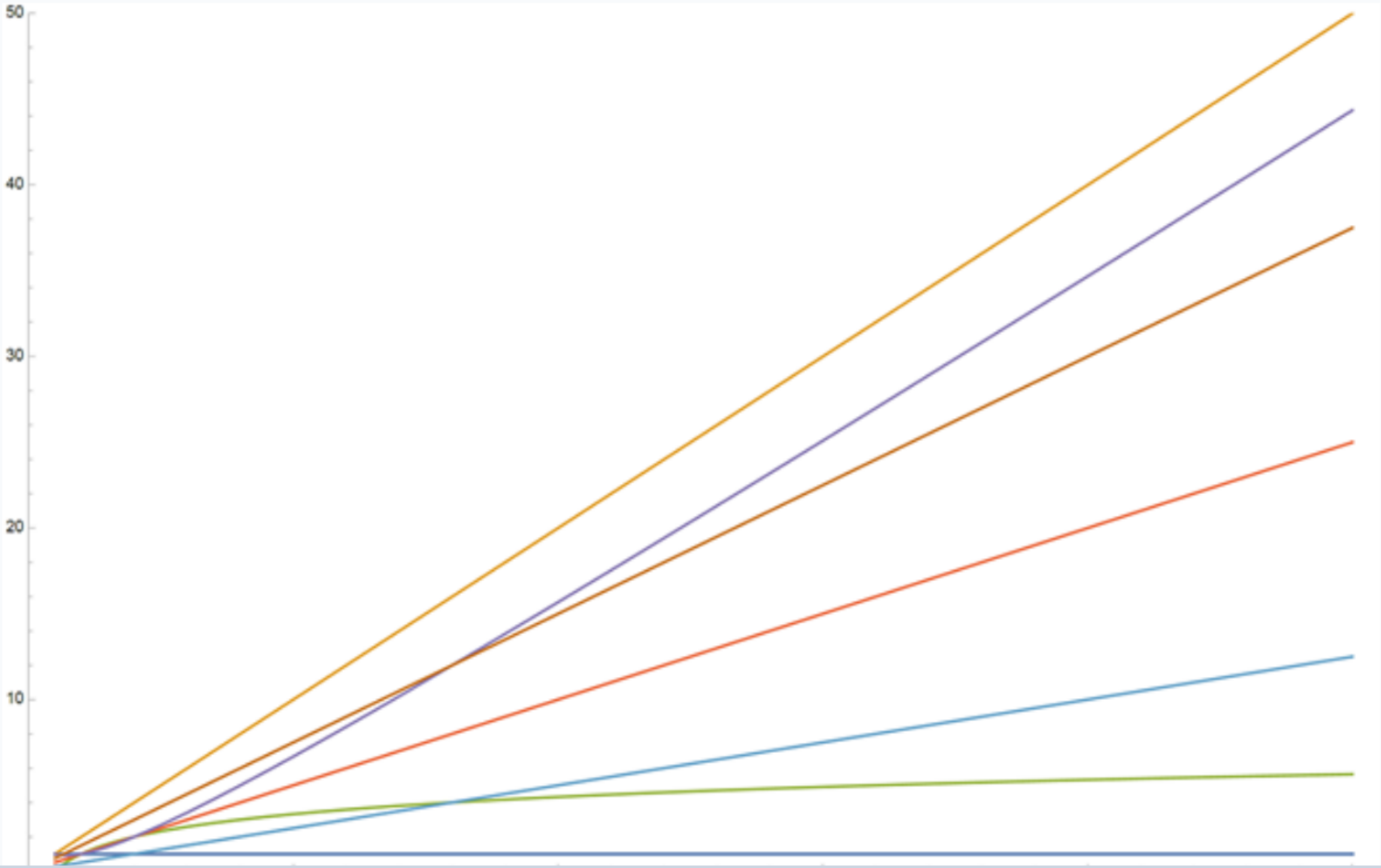
$O(\log_2 N)$

# Aside: what does $O(\log_2 N)$ look like?

The top line is O(N).
The bottom line is O(1).
**Which of the 5 middle lines is $O(\log_2 N)$?**



# Which plot line represents $O(log_2 N)$?

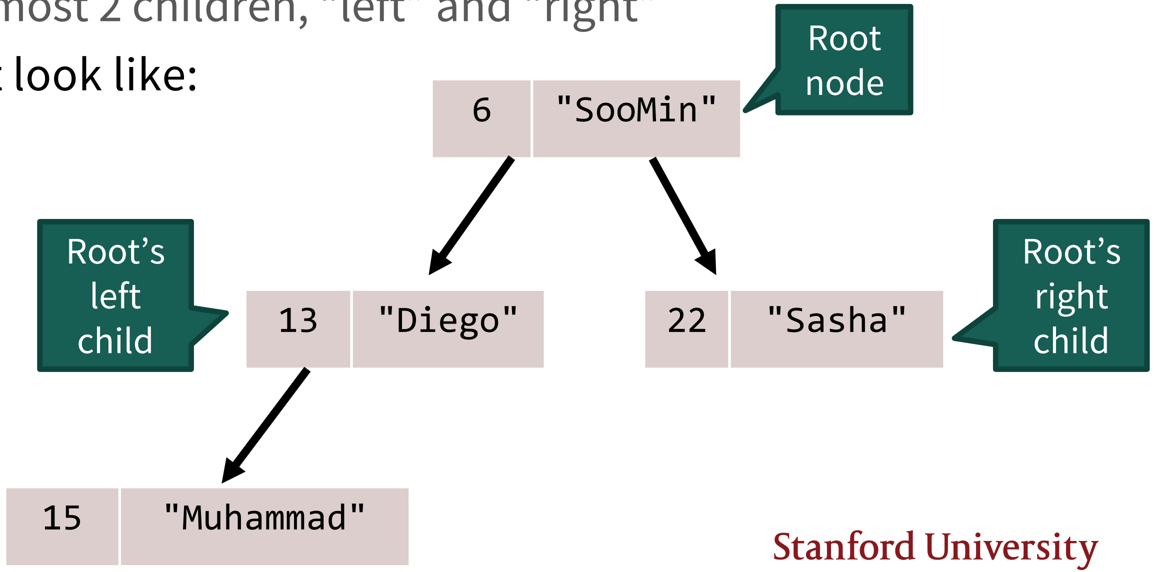


# Today's Agenda:

- Priority Queue ADT
- Two "starter" implementations that build on our array skills
  - Sorted array
  - Unsorted array
  - Performance analysis
- **Binary heap** data structure implementation
  - What are binary heaps?
  - How do we do enqueue in a heap?
  - Homework: dequeue in a heap

# Binary heap for our priority queue

- Binary heaps store things *partially-sorted*.
- The partial sorting will still be stored in an array, but it's best to imagine it as what we call a binary "tree"
  - One root node at the top
  - Each node has at most 2 children, "left" and "right"
- Here's what it might look like:



# Binary Heaps

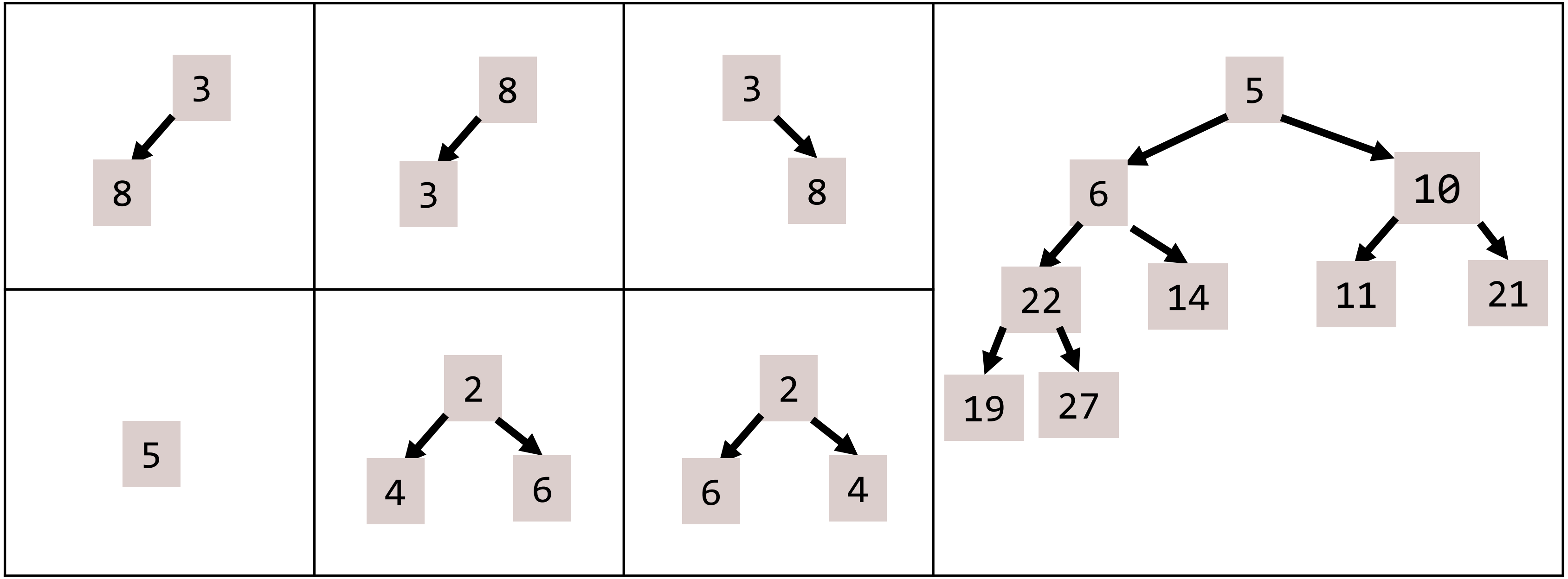
Binary heaps have a few special restrictions, in addition to being a binary tree:

- Must be **complete**
  - No “gaps”—nodes are filled in left-to-right on each level (row) of the tree
- Ordering of priorities must obey **heap property**
  - A parent’s priority is always ≤ both its children’s priority (min-heap)

# How many of these are valid binary heaps?

- Must be a valid **binary tree**
- Must be **complete**
- Ordering of priorities must obey **heap property**

*For the next few slides, we'll focus on the priority, so for simplicity we'll leave the payload off the diagrams.*

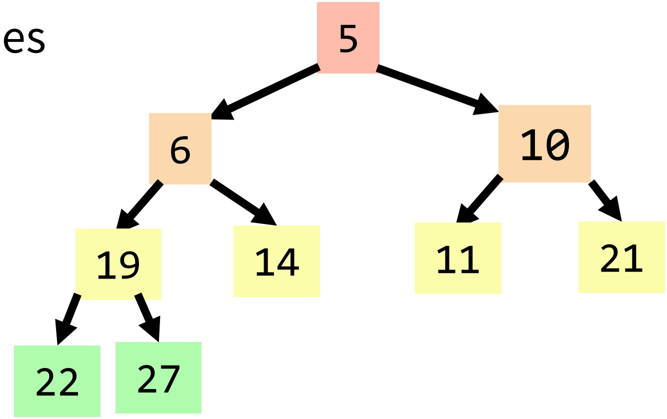


# How many of these are valid binary heaps?

- 0 -2
- 3
- 4
- 5
- 6-7

# Implementing binary heap in an array

- Because of the special constraint that they must be **complete**, binary heaps fit nicely into an **array**
- We fill the array by reading out the tree nodes top to bottom, left to right

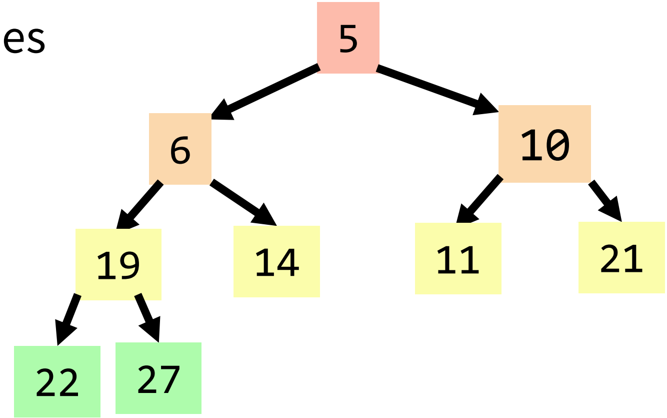



```
_size     = 9
_capacity = 15
_elements =
```

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 6 | 10 | 19 | 14 | 11 | 21 | 22 | 27 | ? | ? | ? | ? | ? | ? |

# Implementing binary heap in an array

- Because of the special constraint that they must be **complete**, binary heaps fit nicely into an **array**
- We fill the array by reading out the tree nodes top to bottom, left to right



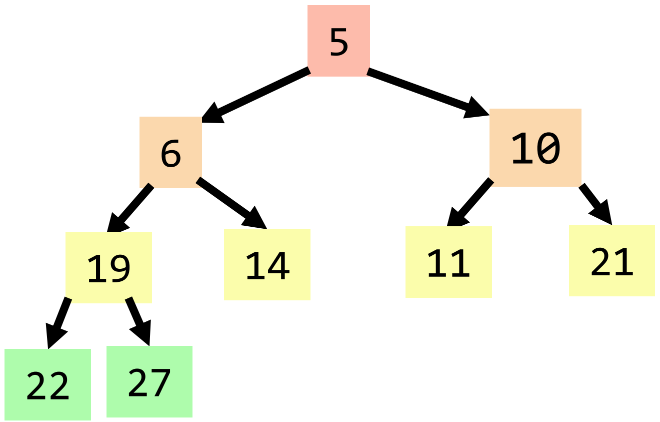

Let's hop into the code now!

```
_size     = 9
_capacity = 15
_elements =
```

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 6 | 10 | 19 | 14 | 11 | 21 | 22 | 27 | ? | ? | ? | ? | ? | ? |

# Navigating in the heap array

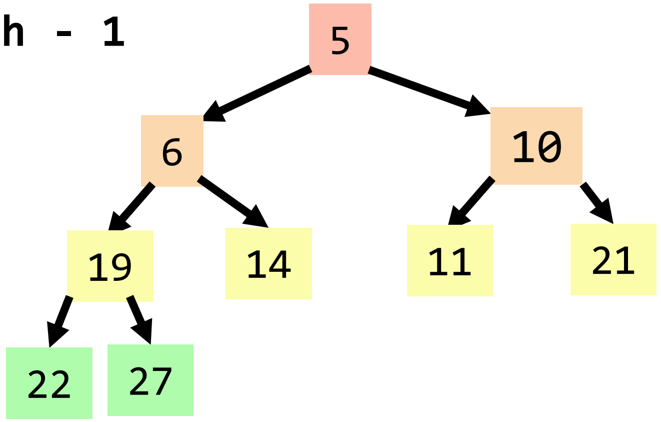
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 5 | 6 | 10 | 19 | 14 | 11 | 21 | 22 | 27 |




- The parent of the node found in array **index** `i` is found where?
  - A. In array index `i / 2`
  - B. In array index `i - 2`
  - C. In array index `(i - 1) / 2`
  - D. In array index `2i`
  - E. Somewhere else
  - › *For now, assume that the node in array index* `i` *has a parent, i.e.,* `i > 0`
  - › *Extra time? Think about a formula for the index of the left and right child of index* `i`

# Fact summary: Binary heap in an array

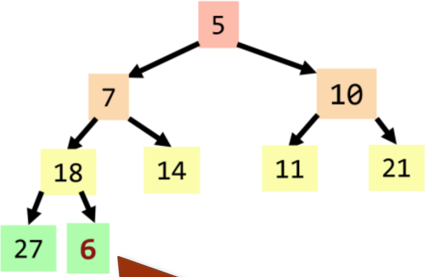
- For tree of height **h**, required array length is **`2^h - 1`**
- For a node in array index **`i`**:
  - Parent is at array index: **`(i – 1)/2`**
  - Left child is at array index: **`2i + 1`**
  - Right child is at array index: **`2i + 2`**
  - *These all assume the parent/child exists*




| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 5 | 6 | 10 | 19 | 14 | 11 | 21 | 22 | 27 |

**Take a photo of this slide for reference!**

# Binary heap enqueue algorithm (append + “bubble up”)

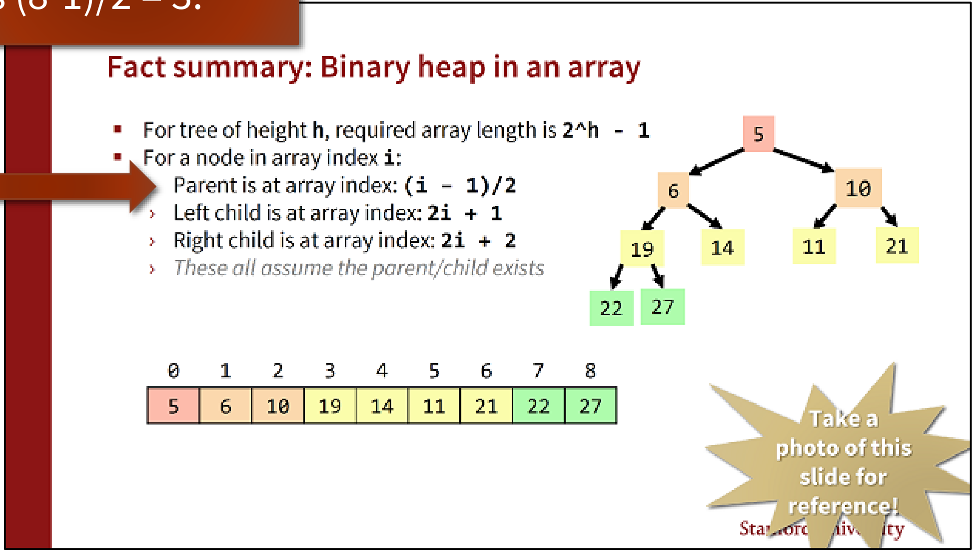



_size = 9, _capacity = 15

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | … | 14 |
|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 7 | 10 | 18 | 14 | 11 | 21 | 27 | 6 | … | ? |

We can tell by looking at this tree visualization that the 6 doesn't go here—but remember in the code all you have is the array. How do we tell there?

Parent of index 8 is (8-1)/2 = 3.

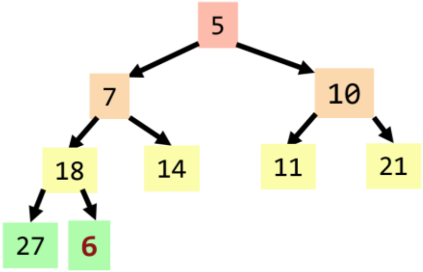


## Fact summary: Binary heap in an array

- For tree of height h, required array length is 2^h - 1
- For a node in array index i:
  - Parent is at array index: (i - 1)/2
  - Left child is at array index: 2i + 1
  - Right child is at array index: 2i + 2
  - *These all assume the parent/child exists*

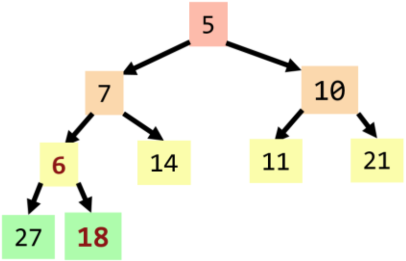
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 5 | 6 | 10 | 19 | 14 | 11 | 21 | 22 | 27 |

# Binary heap enqueue algorithm (append + "bubble up")



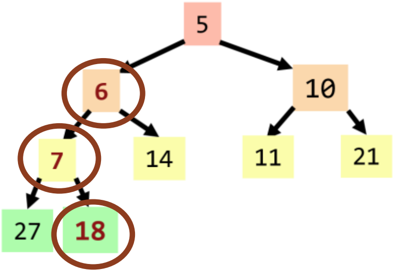

`_size = 9, _capacity = 15`

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | … | 14 |
|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 7 | 10 | 18 | 14 | 11 | 21 | 27 | 6 | … | ? |




`_size = 9, _capacity = 15`

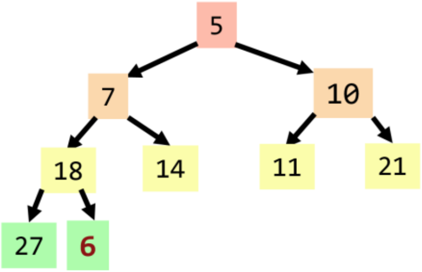
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | … | 14 |
|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 7 | 10 | 6 | 14 | 11 | 21 | 27 | 18 | … | ? |




`_size = 9, _capacity = 15`

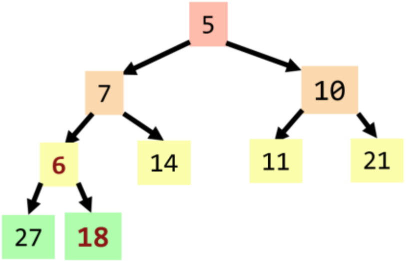
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | … | 14 |
|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 6 | 10 | 7 | 14 | 11 | 21 | 27 | 18 | … | ? |

# Binary heap enqueue algorithm (append + “bubble up”)



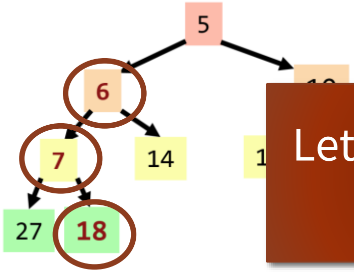

_size = 9, _capacity = 15

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | … | 14 |
|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 7 | 10 | 18 | 14 | 11 | 21 | 27 | 6 | … | ? |




_size = 9, _capacity = 15

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | … | 14 |
|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 7 | 10 | 6 | 14 | 11 | 21 | 27 | 18 | … | ? |



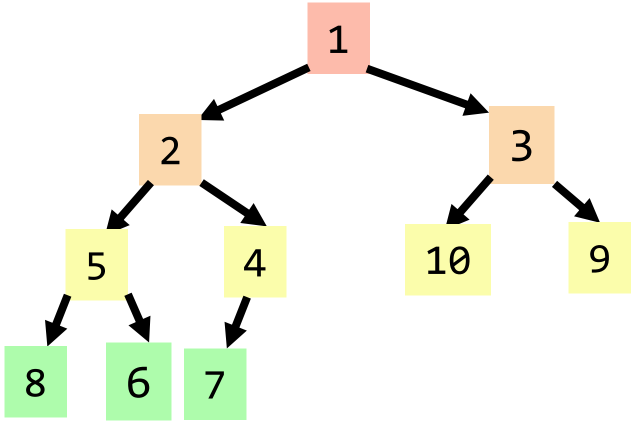

_size = 9, _capacity = 15

| 2 | 3 | 4 | 5 | 6 | 7 | 8 | … | 14 |
|---|---|---|---|---|---|---|---|---|
| 10 | 7 | 14 | 11 | 21 | 27 | 18 | … | ? |

Let’s hop into the code now!

# Checking our test case

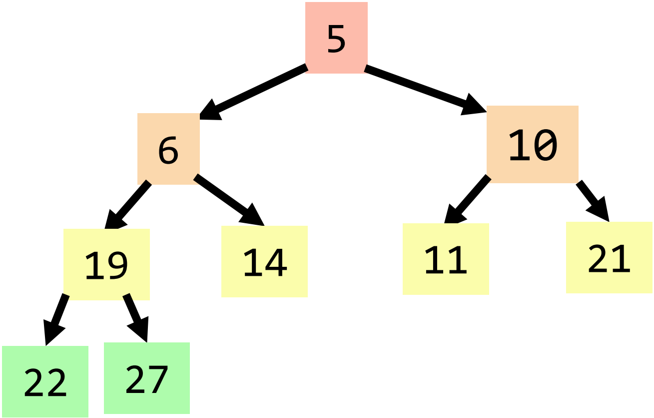
Inserted values: {5, 8, 9, 7, 1, 10, 3, 4, 6, 2}




`_size = 10, _capacity = 10`

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 5 | 4 | 10 | 9 | 8 | 6 | 7 |

# Dequeue algorithm




- Remove the highest-priority item
- Move the "last" element (array-index-wise) into its place
- "Bubble down" swaps until it is correctly placed
  - Important: of the two children, swap with the higher priority (smaller number) child

## Entirely unsorted is too chaotic, but entirely sorted is too difficult to maintain

$O(\log_2 N)$

Is there a "just-right" compromise with the **best of both**?

$O(\log_2 N)$

# Dequeue and “trickle-down” algorithm summary

**1. Remove the min element (the one in the root node—index 0) and that’s the value you’re going to return**

- There’s now a “gap”—so the heap no longer follows the structural requirement that it be “complete”

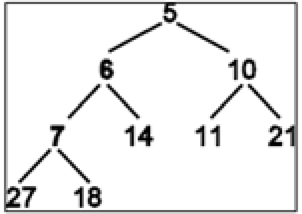
**2. Promote the *last* element into the root node (index 0) position**

- We have now immediately restored the “complete” property, but…
- …we have likely broken the “heap ordering” property!

**3. “Trickle down” the new root element until the heap ordering property is restored**

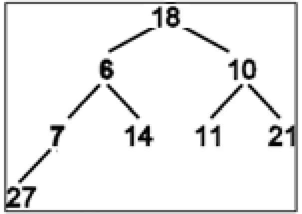
- Pick the smaller value of the left and right children of this element, and swap downward with that smaller one (i.e., you might trickle-down left, and you might trickle-down right, depending on which is smaller!)
- Repeat step 3 as needed (until it is smaller than both left and right children)

# Binary heap dequeue (delete min + “trickle down”)



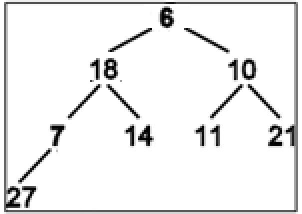

Size=9, Capacity=15

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | … | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 6 | 10 | 7 | 14 | 11 | 21 | 27 | 18 | ? | … | ? |



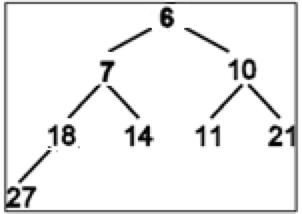

Size=8, Capacity=15

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | … | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 18 | 6 | 10 | 7 | 14 | 11 | 21 | 27 | 18 | ? | … | ? |




Size=8, Capacity=15

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | … | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 6 | 18 | 10 | 7 | 14 | 11 | 21 | 27 | 18 | ? | … | ? |




Size=8, Capacity=15

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | … | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 6 | 7 | 10 | 18 | 14 | 11 | 21 | 27 | 18 | ? | … | ? |