Programming Abstractions

CS106B

Cynthia Bailey Lee
Julie Zelenski

Topics:

- Priority Queue ADT
  - Heap data structure implementation
    - What are binary trees?
    - What are heaps?
    - How do we do enqueue/dequeue operations on heaps?

Priority Queue

Emergency Department waiting room operates as a priority queue: patients are sorted according to priority (urgency), not “first come, first serve” (in computer science, “first in, first out” or FIFO).

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
    - Examples:
      - **a class MedicalRecord** that has many fields and is the patient's entire medical history
      - **a string** that is the name of a student waiting in the Lair queue (in a world where Lair is based on urgency of request, rather than FIFO)
      - etc.

6	"SooMin"	13	"Diego"	15	"Muhammad"	22	"Sasha"

Two priority queue implementation options

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

Unsorted array

- Always enqueue new element *at the end of the array*
- Dequeue by searching entire array for highest-priority item, then removing it, and (if needed) scooting elements over to fill in the gap

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

Sorted array

- Always enqueue new elements *where they go* in priority-sorted order, with the highest-priority item at the end of the array
- Dequeue by taking the last element of the array

Priority queue implementations

Unsorted array

Enqueue is FAST

- Just throw it in the array at the back
- **O(1)**

Dequeue/peek is SLOW

- Hard to find item the highest priority item—could be anywhere
- Might need to scoot over elements to fill gap
- **O(N)**

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

This file is licensed under the Creative Commons Attribution-Share Alike 3.0 Unported license. Keyah Cheatum http://commons.wikimedia.org/wiki/File:Messy_Room.JPG

Priority queue implementations

Sorted array

Enqueue is SLOW

- Need to step through the array to find where item goes in priority-sorted order
- If proper place is in the front/middle, need to scoot over other elements to make room
- **O(N)**

Dequeue/peek is FAST

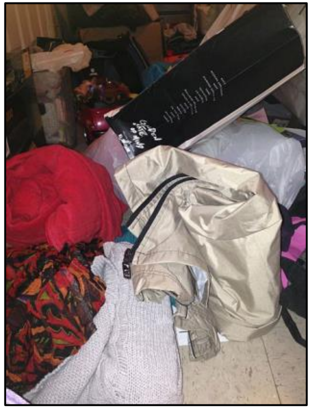
- Easy to find item you are looking for (last in array)
- No need to scoot over elements when removing last
- **O(1)**

Image is in the public domain. http://commons.wikimedia.org/wiki/File:Wall_Closet.jpg

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

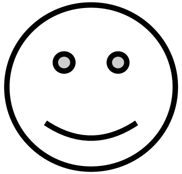
Would be great if we could get the best of both…

Fast enqueue *and* fast dequeue/peek



+

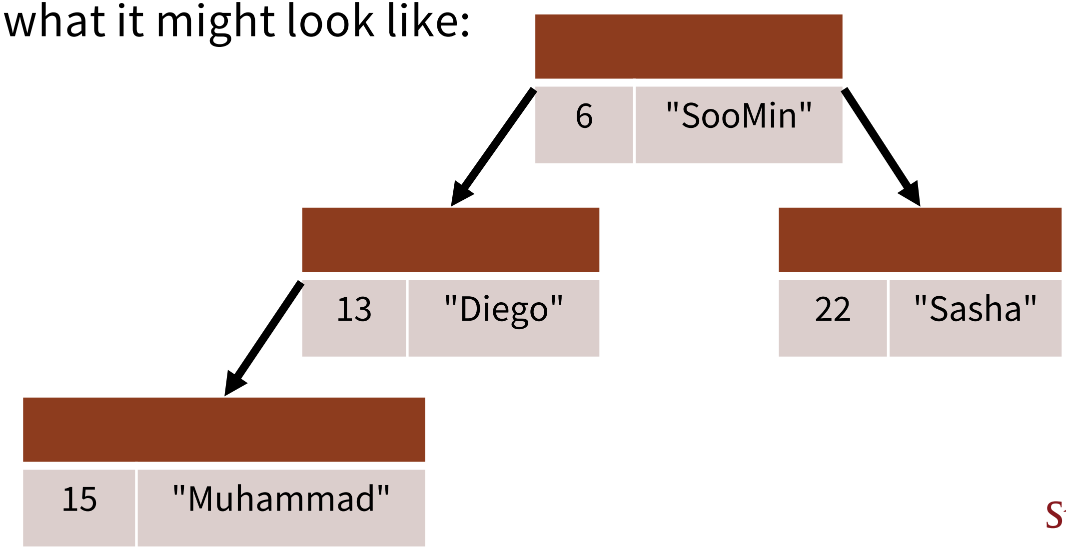
=



Fast enqueue Fast dequeue/peek

Binary heap for our priority queue

- Instead of storing our priority queue nodes entirely sorted or entirely unsorted, we will store them *partially-sorted*.
- The partial sorting will still be stored in an array, but it's best to imagine it as what we call a "tree" in computer science (computer science trees are upside-down for some reason ¯_(ツ)_/¯)
- Here's what it might look like:



Binary trees

Before we delve into how to construct a binary heap, let's take a step back and introduce computer science binary trees generally

A binary tree

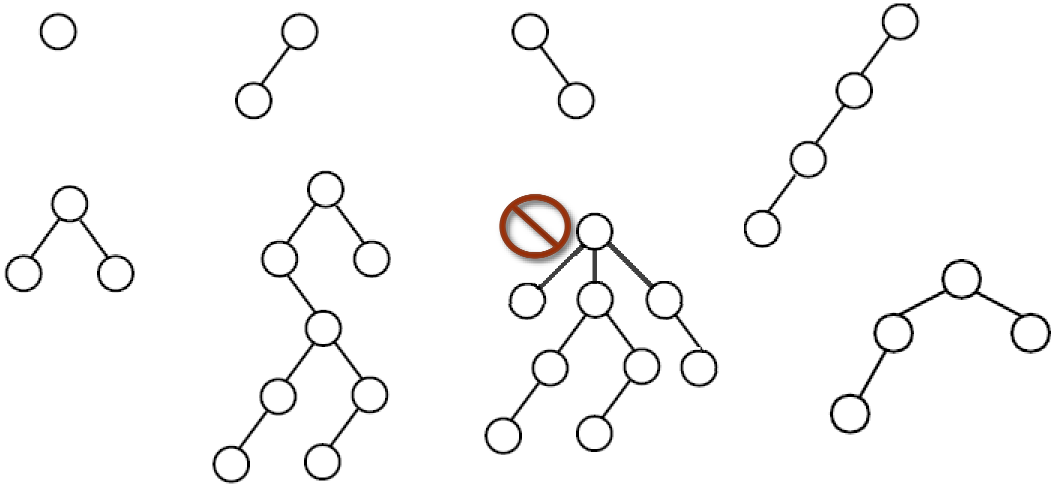
“In computer science, a **binary tree** is a tree data structure in which each node has at most two child nodes, usually distinguished as "left" and "right."”

(Thanks, Wikipedia!)

“In computer science, a **binary tree** is a tree data structure in which each node has at most two child nodes, usually distinguished as "left" and "right.""

(Thanks, Wikipedia!)

How many of these are valid binary trees?



Heaps!

Binary Heaps*

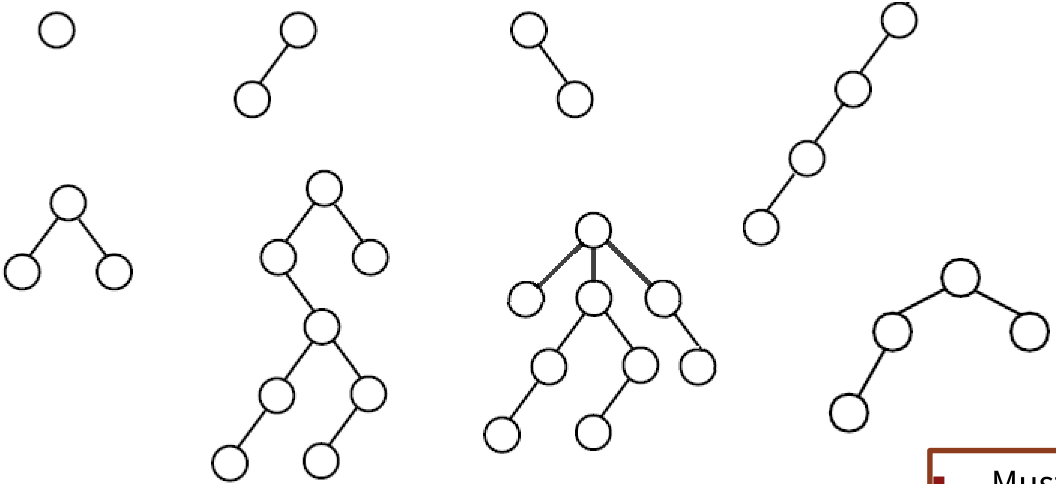
Binary heaps are **one kind** of binary tree

They have a few special restrictions, in addition to the usual binary tree:

- Must be **complete**
  - No "gaps"—nodes are filled in left-to-right on each level (row) of the tree
- Ordering of data must obey **heap property**
  - Min-heap version: a parent's priority is always ≤ both its children's priority
  - Max-heap version: a parent's priority is always ≥ both its children's priority

** There are other kinds of heaps as well. For example, binomial heap is an extra-fun one!*

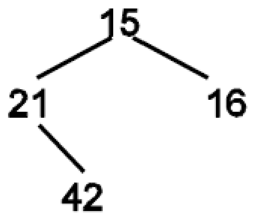
How many of these could be valid binary heaps?

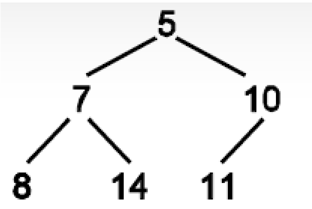


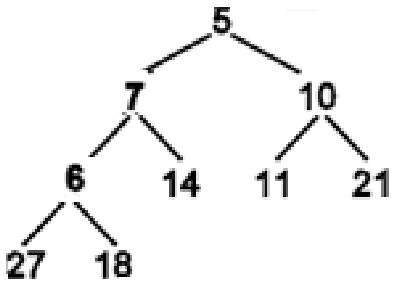
A. 0-1
B. 2
C. 3
D. 4
E. 5-8

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

How many of these are valid min-binary-heaps?





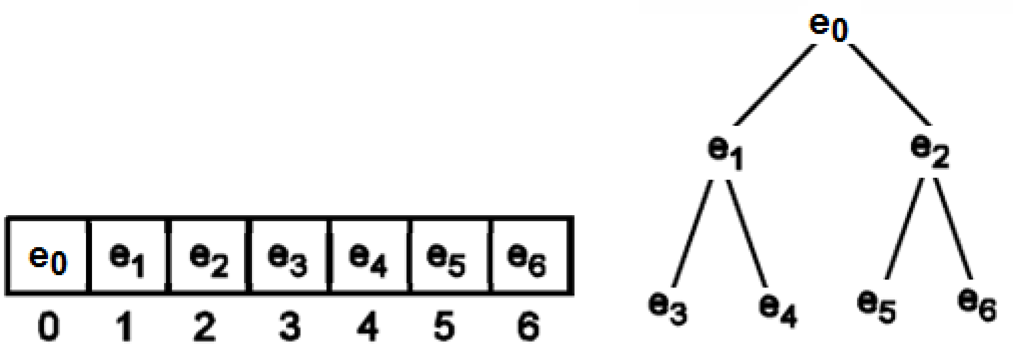


- Must be a valid **binary tree**
- Must be **complete**
- Ordering of data must obey **heap property**

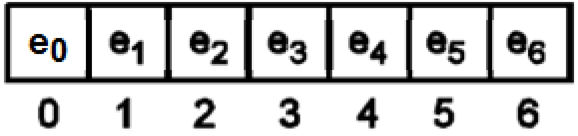
Binary heap in an array

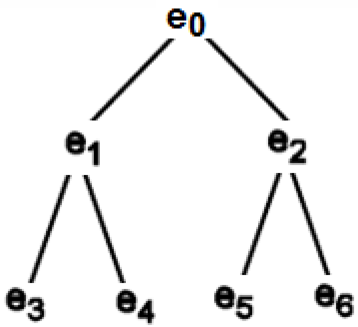
Binary heap in an array

- Because of the special constraint that they must be **complete**, binary heaps fit nicely into an **array**
  - *As we'll see in later lectures, this is <u>not</u> true of some other kinds of tree data structures, and we'll use a different approach for those*



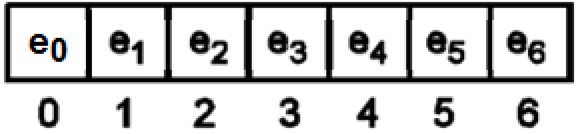
Heap in an array

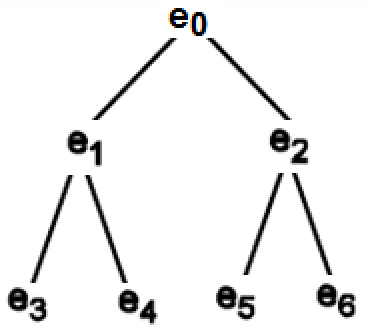




- Q: The parent of the node found in array index **`i`** is found where?
  - A. In array index **`i - 2`**
  - B. In array index **`i / 2`**
  - C. In array index **`(i - 1)/2`**
  - D. In array index **`2i`**
  - E. Somewhere else
  - › For now, assume that the node in array index **`i`** has a parent.
  - › In your code, of course you'll want to be careful not to go up past the top of the tree.

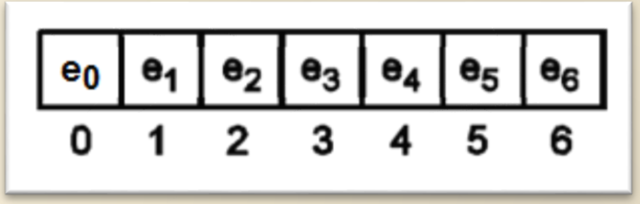
Heap in an array

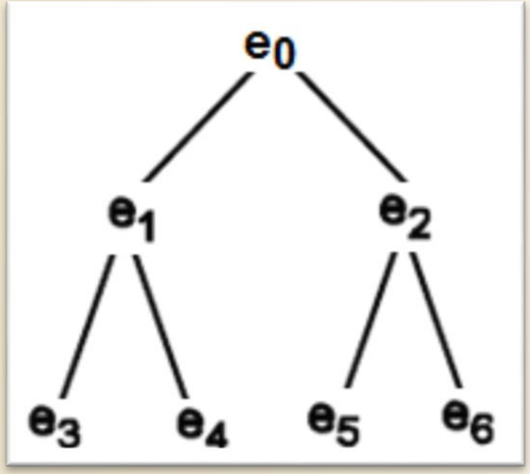




- Q: Write an expression for the array index where we find the right child of the node in array index **`i`**.
  - For now, assume that the node in array index **`i`** has a right child.
  - In your code, of course you'll want to be careful not to go past the ends of the tree.

Fact summary: Binary heap in an array



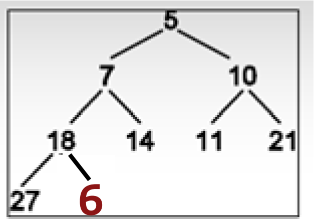


- For tree of height **`h`**, array length is **`2^h-1`**
- For a node in array index **`i`**:
  - Parent is at array index: **`(i – 1)/2`**
  - Left child is at array index: **`2i + 1`**
  - Right child is at array index: **`2i + 2`**

Bookmark this slide!

Binary heap enqueue and dequeue

Binary heap enqueue example (insert 6 + "bubble up")

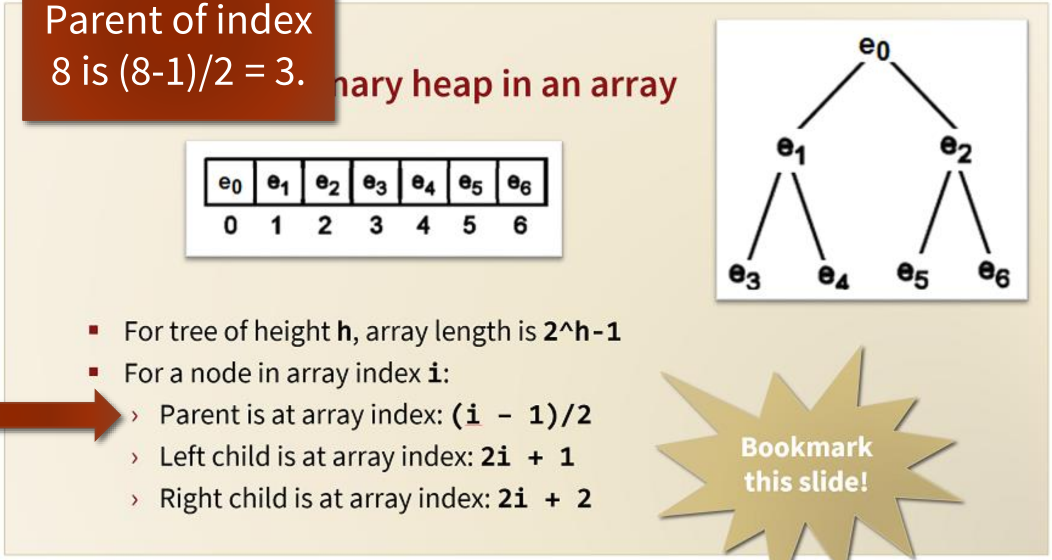


We can tell by looking at this tree visualization that the 6 doesn't go here—but remember in the code all you have is the array. How do we tell there?

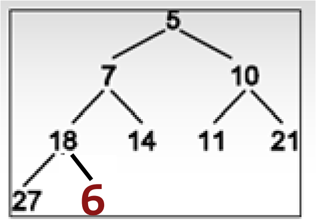
Size=9, Capacity=15

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

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

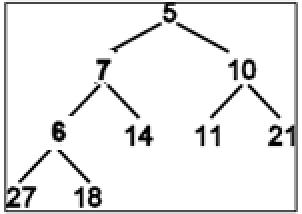


Binary heap enqueue example (insert 6 + “bubble up”)



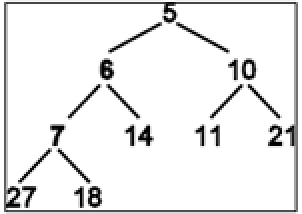
Size=8, Capacity=15

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



Size=9, Capacity=15

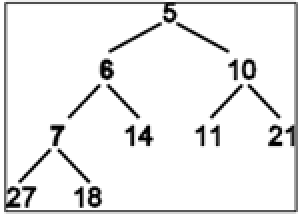
0	1	2	3	4	5	6	7	8	9	…	14
5	7	10	6	14	11	21	27	18	?	…	?



Size=9, Capacity=15

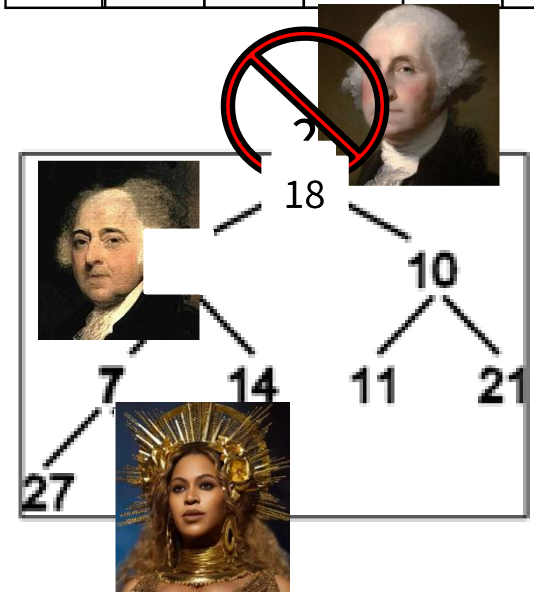
0	1	2	3	4	5	6	7	8	9	…	14
5	6	10	7	14	11	21	27	18	?	…	?

Binary heap dequeue (delete min)



Size=9, Capacity=15

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



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”

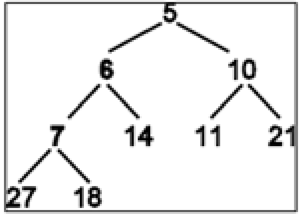
2. Promote the *last* element into the root node (index 0) position

- 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

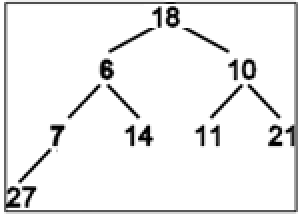
- Pick the <u>smaller</u> value of the left and right children of this element, and swap downward with that smaller one (i.e., you might trickle-down left, and you might trickle-down right, depending on which is smaller!)
- Repeat step 3 as needed (until it is smaller than <u>both</u> left and right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

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



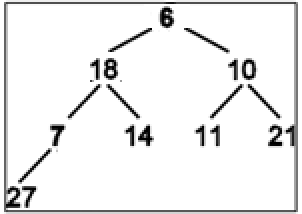
Size=9, Capacity=15

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



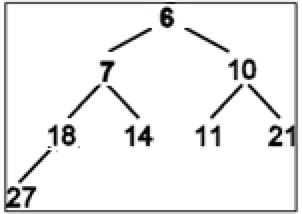
Size=**8**, Capacity=15

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

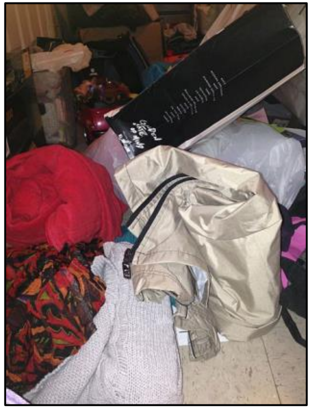
0	1	2	3	4	5	6	7	8	9	…	14
6	**7**	10	**18**	14	11	21	27	18	?	…	?

Summary analysis

Comparing our priority queue options

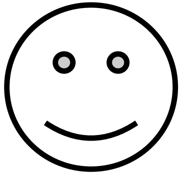
Would be great if we could get the best of both…

Fast enqueue *and* fast dequeue/peek



+

=



Fast enqueue

Fast dequeue/peek

Review: priority queue implementation options performance

Unsorted array

- Enqueue new element in back: O(1)
- Dequeue by searching list and scooting over: O(N)

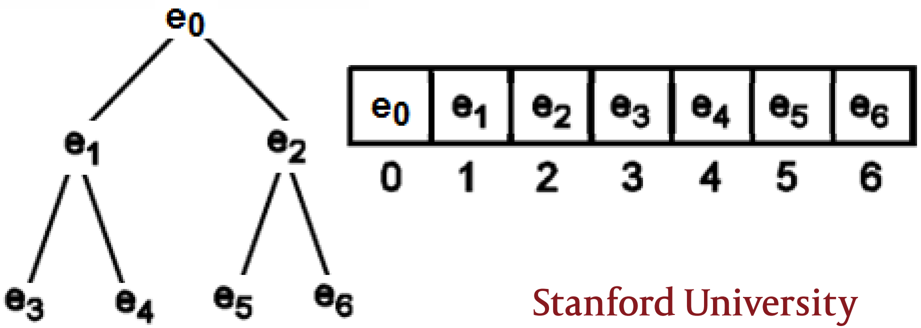
Sorted array

- Always enqueue in sorted order: O(N)
- Dequeue from back: O(1)

Binary heap

- Enqueue + “bubble up”: O(logN)
- Dequeue + “trickle down”: O(logN)

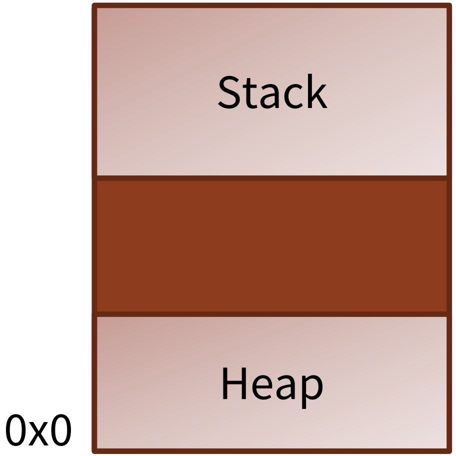
☺



Final aside on terminology

Aside: Binary Heap, not to be confused with Heap memory!

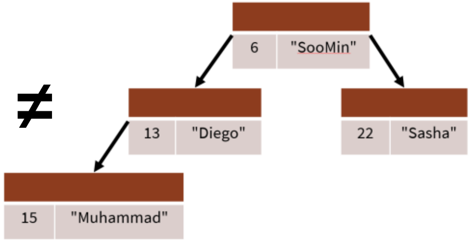
- The Stack section of memory **<u>is</u>** a Stack like the ADT
- The Heap section of memory **<u>has nothing to do</u>** with the Heap structure.



=

Stack ADT

≠



Heap data structure

- Probably just happened to reuse the same word ☹

Stanford University

Source: http://www.flickr.com/photos/35237093334@N01/409465578/
Author: http://www.flickr.com/people/35237093334@N01 Peter Kazanjy]