Heaps

CSE 332 Spring 2020

Instructor: Hannah C. Tang

Teaching Assistants:

Annie Mao	Howard Xiao	Lucy Jiang
Chris Choi	Jade Watkins	Nathan Lipiarski
Ethan Knutson	Khushi Chaudhari	Richard Jiang

Announcements

- P1: Congrats on completing Checkpoint 1!
  - Next duedate is Tue, Apr 14 or the entire project

- Exs 2-4 due Mon, Apr 13

- Quiz 1 will be released after lecture and is due before lecture on Fri, Apr 10
  - Should take you a few hours to complete
  - Open book, unlimited group size
    - Please submit *one* quiz per group
  - Question clarifications in Piazza, but staff won't clarify concepts

Learning Objectives

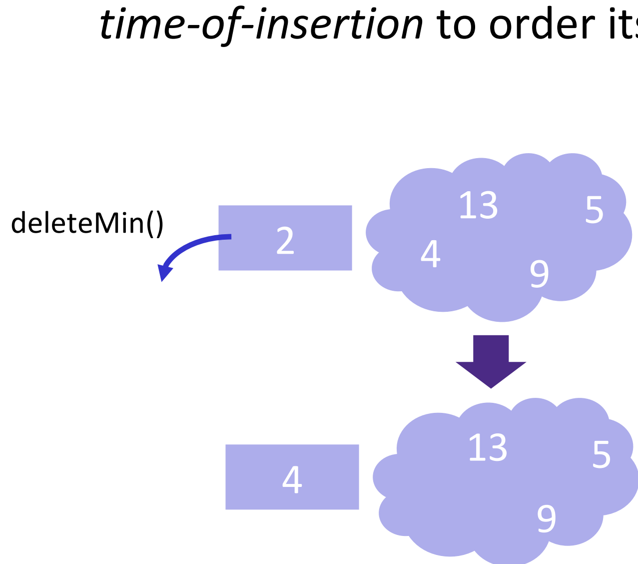
- Describe the *structure* and *order* properties of a binary heap
- Be able to convert between binary heap's two representations
- Implement the `add` and `deleteMin` methods, as well as the `percolateUp` and `percolateDown` helper methods
  - And also understand their expected and worst-case runtimes
- Implement Floyd's buildHeap algorithm, and be able to prove its correctness and efficiency guarantees

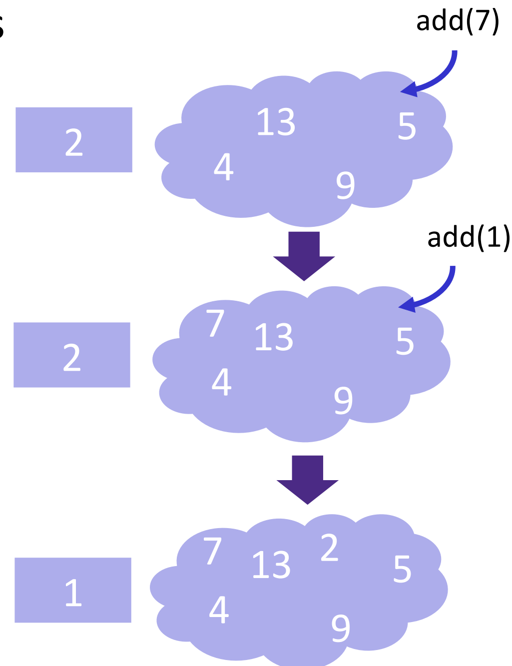
Lecture Outline

- Review
  - **Priority Queue ADT**
  - Tree Terminology and Properties
- Binary Heap
  - Tree Visualization and Operations
  - Array Representation
  - [next lecture] Floyd's `buildHeap` Algorithm

Review: Priority Queue ADT

- In a Priority Queue, the item that matters is the min (or max)
- Unlike FIFO Queues, PQs use *priorities* instead of *time-of-insertion* to order its elements





Priority Queue: Example

add *a* with priority *5*

add *b* with priority *3*

add *c* with priority *4*

w = **deleteMin**

x = **deleteMin**

add *d* with priority *2*

add *e* with priority *6*

y = **deleteMin**

z = **deleteMin**

after execution:

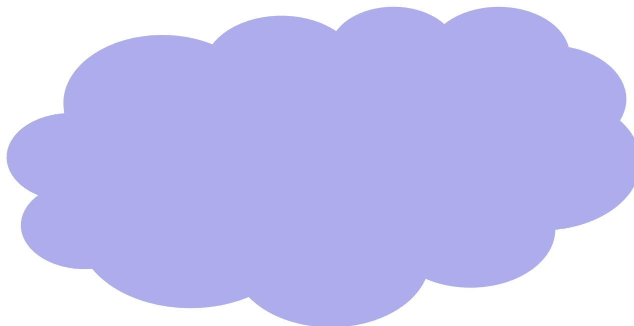
6->e

w = b

x = c

y = d

z = a



Possible Data Structures

	add	deleteMin
Unsorted Array	add elt to end O(1)	search + shift O(n)
Unsorted LinkedList		
Sorted Circular Array		
Sorted Linked List		
Binary Search Tree (BST)	find correct pos O(h)	find min O(h)

But in a BST h is O(n), not O(log n)!

Assumptions: Worst case; Arrays have enough space

Our Eventual Data Structure: The Heap

- **Heap:**
  - `add`: O(log n), worst case
  - `deleteMin`: O(log n), worst case
  - If items added in random order, expected case for `add` is O(1)
  - Very good constant factors

- **Key idea:** Only pay for functionality needed
  - We need something better than scanning unsorted items
  - But we do not need to maintain a full sorted list

- We *visualize* our heap as a tree, so let's review some terminology

Lecture Outline

- Review
  - Priority Queue ADT
  - **Tree Terminology and Properties**

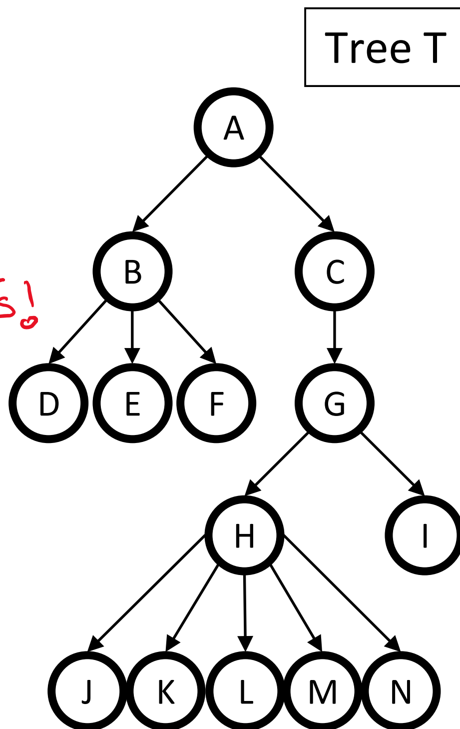
- Binary Heap
  - Tree Visualization and Operations
  - Array Representation

Review: Tree Terminology

- root(T): A
- leaves(T): D–F, J–N, I
- children(B): D- F
- parent(H): G
- siblings(E): D, F
- ancestors(F): F, B, A
- descendents(G): G, H, I, J–N
- subtree(G): G, H, I, J–N
- depth(B): 1
- height(G): 2
- height(T): 5
- degree(B): 3
- branching factor(T): 5 (max), ~2.2 (avg)

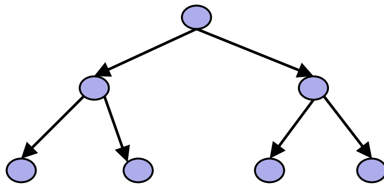
recursive defintions!

defined by # edges

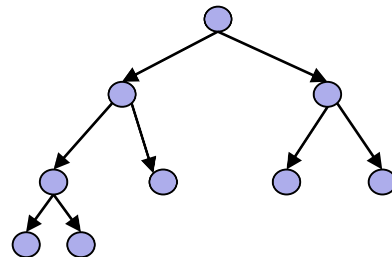


Types of Trees

Binary tree	Every node has ≤ 2 children
N-ary tree	Every node has ≤ n children
Perfect tree	Every row is completely full
Complete tree	All rows except possibly the bottom are completely full. The bottom row is filled from left to right



Perfect Tree

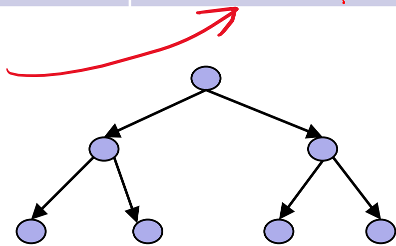


Complete Tree

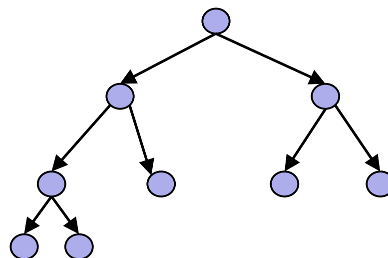
Perfect Tree Properties

Height	Number of Nodes	Number of Leaves
1	3	2
2	7	4
3	15	8
4	31	16
h	$2^{h+1}-1$	2^h

See Weiss chapter 1 for this very important summation!



Perfect Tree



Complete Tree

Lecture Outline

- Review
  - Priority Queue ADT
  - Tree Terminology and Properties
- Binary Heap
  - **Tree Visualization and Operations**
  - Array Representation

Binary (Min-)Heap (1 of 3)

- More commonly known as a *binary heap* or simply a *heap*
  - The "min" refers to the fact that the special priority value is the smallest; a "max heap" tracks the largest priority

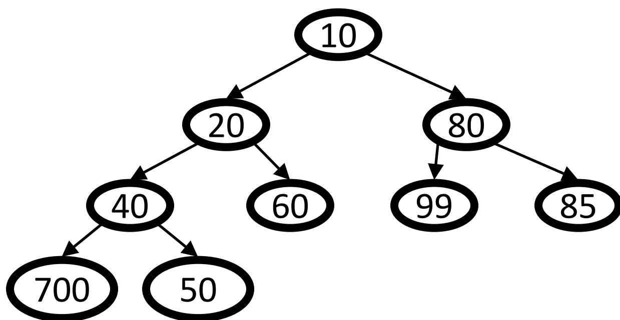
- **Structure Property**: A complete binary tree
- **Order Property**: Every non-root node has a priority value larger than (or possibly equal to) the priority of its parent

How is this different from a binary search tree?

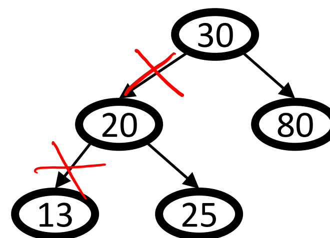
Binary (Min-)Heap (2 of 3)

- More commonly known as a *binary heap* or simply a *heap*
  - The "min" refers to the fact that the special priority value is the smallest; a "max heap" tracks the largest priority

- **Structure Property**: A complete binary tree
- **Order Property**: Every non-root node has a priority value larger than (or possibly equal to) the priority of its parent



A Heap



Not a Heap

Binary (Min-)Heap (3 of 3)

- Where is the minimum priority item?

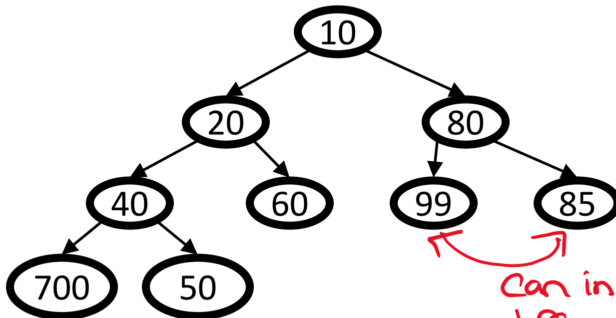
root

- What is the height of a heap with n items?

$\lfloor \log n \rfloor$ ← this is a complete tree, not a perfect tree

- Is this tree unique to this heap?

No!

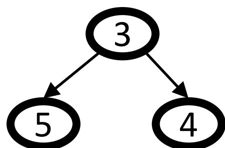


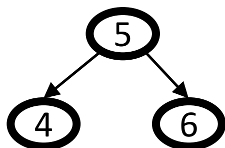
A Heap

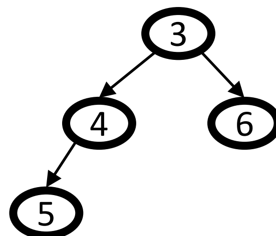
can invert these values to yield a different tree but same heap!

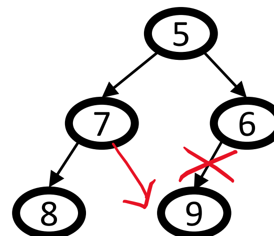
pollev.com/cse332

- Which of these are valid binary min-heaps?









A. Yes, no, yes, yes
B. Yes, yes, yes, yes
C. Yes, no, no, yes
D. Yes, no, yes, no
E. No, no, yes, no
F. I'm not sure …

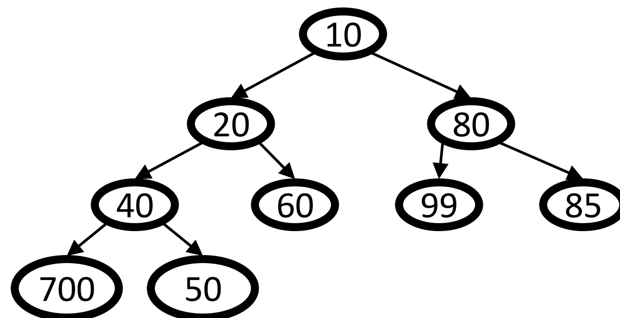
Priority Queue ADT

Priority Queue ADT. A collection storing a set of elements and their priority.

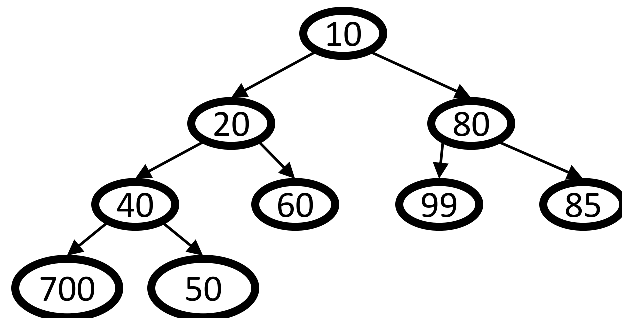
- A PQ has a size defined as the number of elements in the set
- You can add elements (and their priorities)
- You cannot access or remove arbitrary elements, only the element with the min priority

Primary Operations:

- `add`
- `deleteMin`



Binary Heap Helper Functions



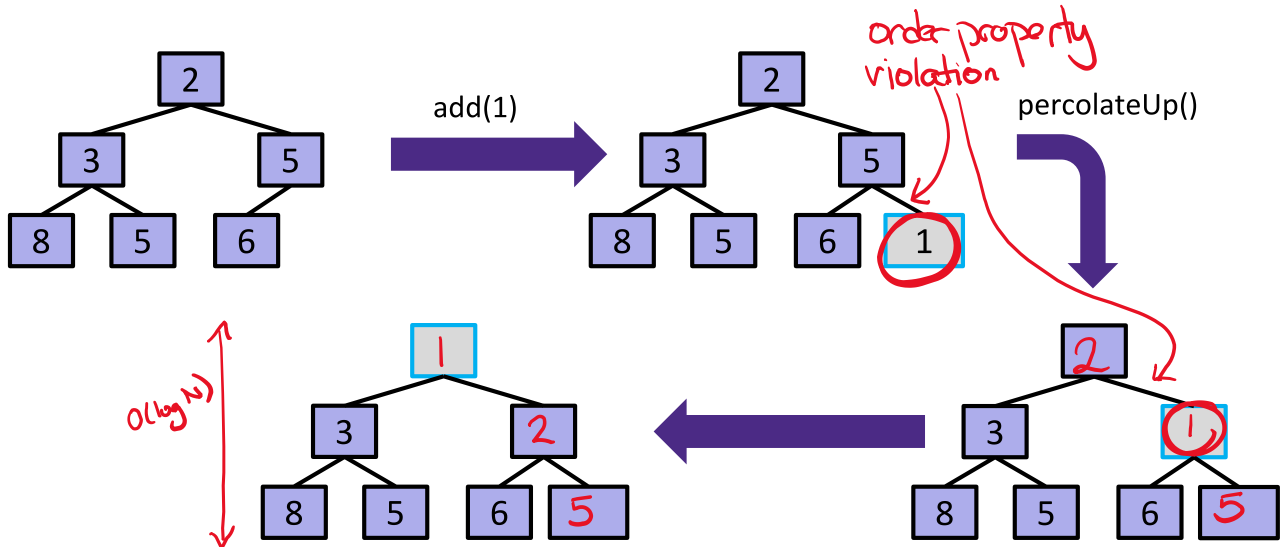
- `add`:
  - Put new node in rightmost position of the last row (*restore structure property*)
  - "Percolate up" to correct layer (*restore order property*)
- `deleteMin`:
  - `answer = root.item`
  - Move rightmost node in last row to root (*restore structure property*)
  - "Percolate down" to correct layer (*restore order property*)

Overall strategy:

- *Preserve complete tree **structure property***
  - *... which may break heap order property*
- *Percolate to restore **heap order property***

Binary Heap: add()

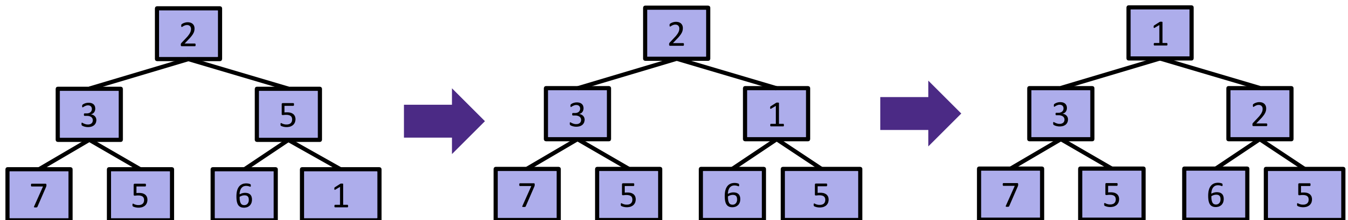
- Put new node in rightmost position of the last row
- “Percolate up” to correct layer



percolateUp()

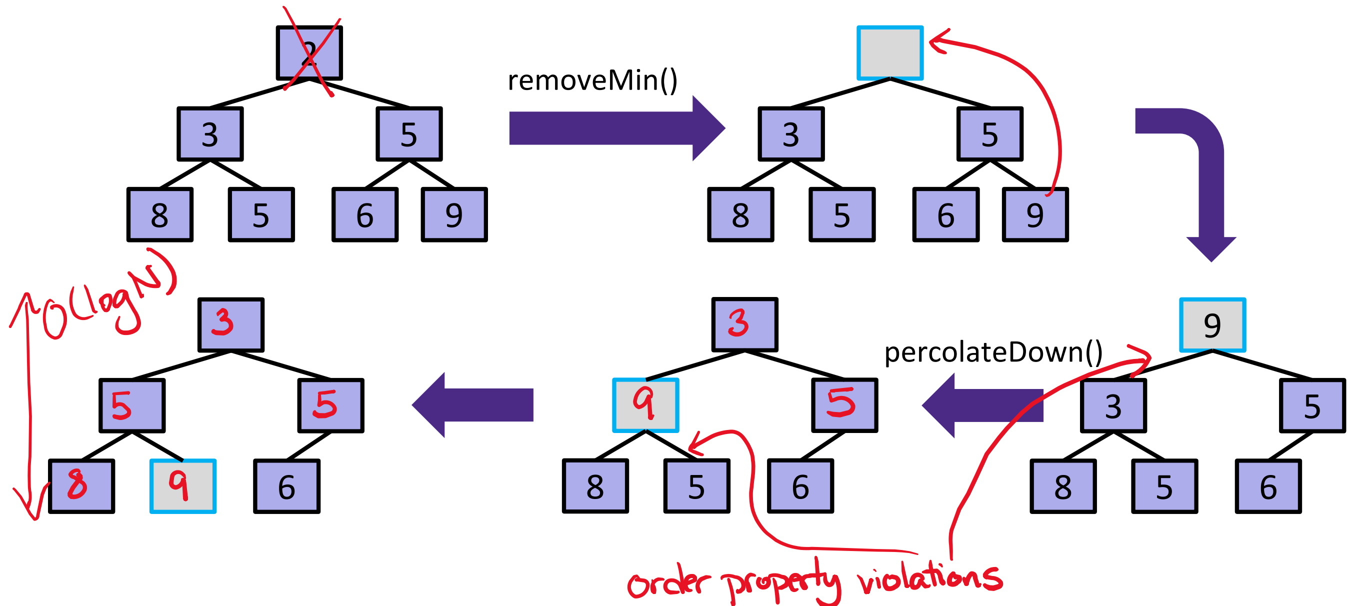
- percolateUp():
  - Put new item in new location
  - If parent larger, swap with parent, and continue
  - Done if parent ≤ item or reached root

- Why does this work? What is the run time? O(log N) (but hang tight for a cool analysis)



Binary Heaps: removeMin()

- Move rightmost node in last row to root
- “Percolate down” to correct layer

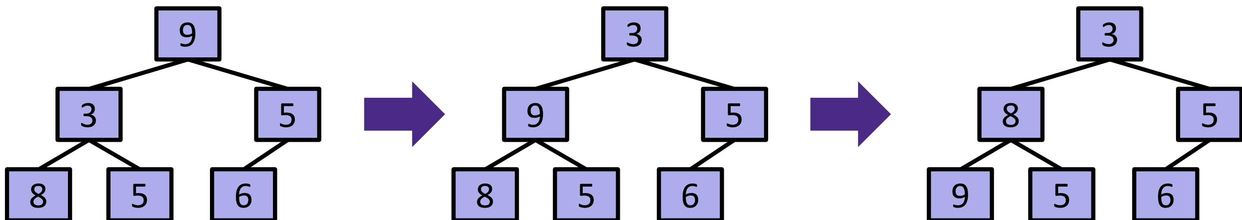


percolateDown()

- percolateDown:
  - Keep comparing with both children
  - Move *smaller* child up and go down one level
  - Done if both children are ≥ item or reached a leaf node

- Why does this work? What is the run time?

O(log N)



Lecture Outline

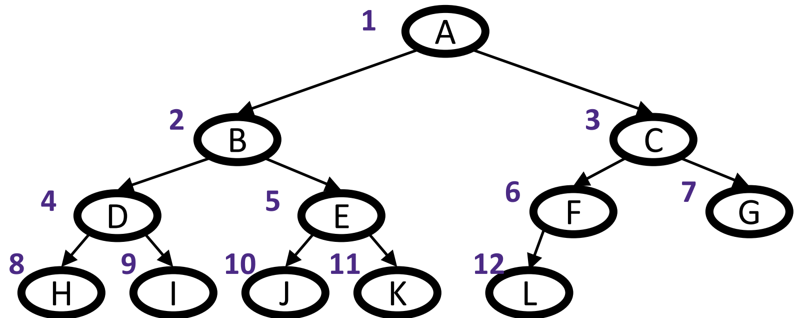
- Review
  - Priority Queue ADT
  - Tree Terminology and Properties
- Binary Heap
  - Tree Visualization and Operations
  - **Array Representation**

A Clever Trick for Storing the Heap…

- All complete trees of size n contain the same edges
  - So why are we even representing the edges?
  - We should only pay for the functionality we need!!

Array Representation of a Binary Heap

- We skip index 0 to make the math simpler, though it's a good place to store the current size of the heap
  - Note: Exercises and P1 start counting from 0



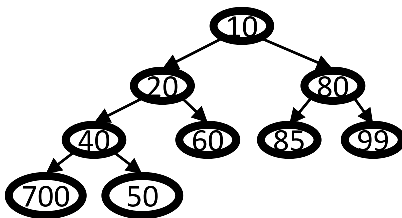
- From node i:
  - left child: $2i$
  - right child: $2i+1$
  - parent: $\lfloor \frac{i}{2} \rfloor$

	A	B	C	D	E	F	G	H	I	J	K	L	
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Pseudocode: add()

```
void insert(int val) {
   if (size == arr.length-1)
     resize();
   size++;
   i = percolateUp(size,val);
   arr[i] = val;
}
```

```
int percolateUp(int hole,
                int val) {
  while (hole > 1 &&
         val < arr[hole/2]) {
    arr[hole] = arr[hole/2];
    hole = hole / 2;
  }
  return hole;
}
```



Disclaimers:

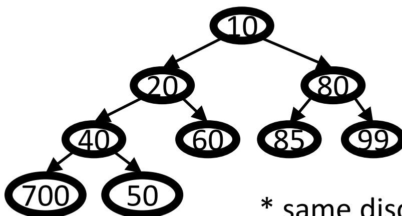
- This pseudocode uses ints. In real use, you will have nodes with priorities and values
- Exercises and P1 start counting from 0

0	1	2	3	4	5	6	7	8	9	10	11	12	13
	10	20	80	40	60	85	99	700	50				

Pseudocode: deleteMin()

```
int deleteMin() {
  if(isEmpty()) throw …
  ans = arr[1];
  hole = percolateDown(
           1, arr[size]);
  arr[hole] = arr[size];
  size--;
  return ans;
}
```

```
int percolateDown(int hole,
                  int val) {
 while (2*hole <= size) {
   left  = 2*hole;
   right = left + 1;
   if (arr[left] < arr[right]
      || right > size)
     target = left;
   else
     target = right;
   if (arr[target] < val) {
     arr[hole] = arr[target];
     hole = target;
   } else
     break;
 }
 return hole;
}
```

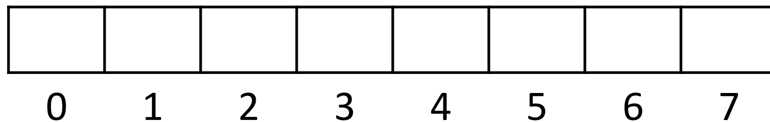


* same disclaimers apply

	10	20	80	40	60	85	99	700	50				
0	1	2	3	4	5	6	7	8	9	10	11	12	13

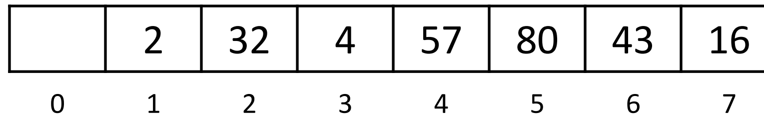
Example

1. add: 16, 32, 4, 57, 80, 43, 2
2. deleteMin

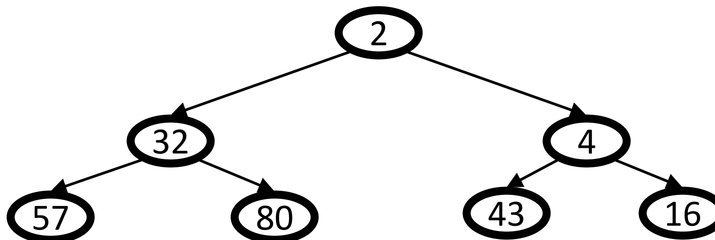


Example: After insertion

1. add: 16, 32, 4, 57, 80, 43, 2
2. deleteMin

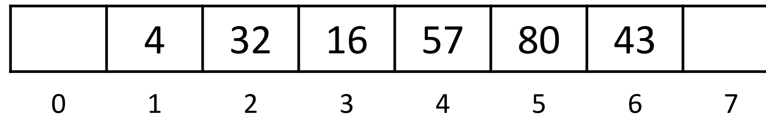


	2	32	4	57	80	43	16
0	1	2	3	4	5	6	7

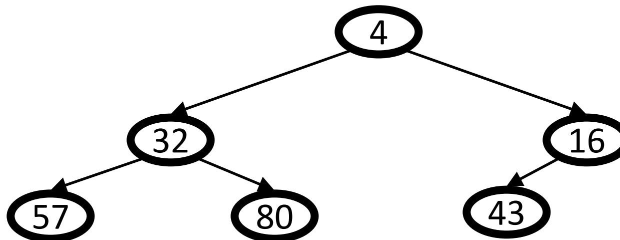


Example: After deletion

1. add: 16, 32, 4, 57, 80, 43, 2
2. deleteMin



	4	32	16	57	80	43	
0	1	2	3	4	5	6	7



Evaluating the Array Implementation

- Advantages:
  - Minimal amount of wasted space:
    - Only index 0 and any unused space on right in the array
    - No "holes" due to complete tree property
    - No wasted space representing tree edges
  - Fast lookups:
    - Benefit of array lookup speed
    - Multiplying / dividing by 2 is extremely fast (see CSE 351 and bit-shifting)
    - Last used position is easily found by using the PQueue's size for the index

- Disadvantages:
  - If the array gets too full, needs to be resized
  - If the array is too empty, wastes space and needs to be resized

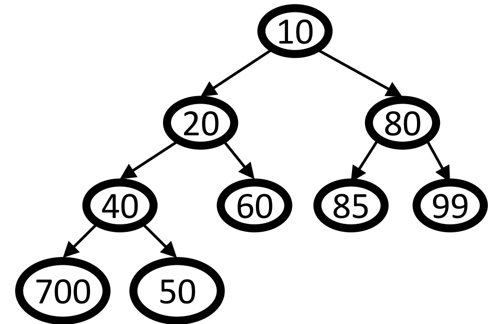
- ***Advantages outweigh Disadvantages: This is how it is done!***

O(1) average-case add()?! (1 of 2)

- Yes, `add`'s worst case is O(log n)
  - It all depends on the order the items are inserted
  - What is the worst case order?
- Empirical studies of randomly ordered inputs shows:
  - Average 2.607 comparisons per insert (# of percolation passes)
  - An element usually moves up 1.607 levels
- If we define "average" as *a single operation with a random input occurring after a sequence of similarly randomized operations*:
  - `add`'s *average case* is O(1)
  - `deleteMin`'s average case is still O(log n)
    - Moving a leaf to the root usually requires re-percolating that item back to the bottom

O(1) average-case add()?! (2 of 2)

- In a complete binary tree, each row has 2x nodes of its parent row
  - Bottom level has ~1/2 of all nodes
  - Second to bottom has ~1/4 of all nodes
  - …



- Intuition:
  - When inserting a *random* priority, likely not to have highest nor lowest priority; somewhere in middle
  - Given a random distribution of priorities in the heap:
    - Bottom level should have the upper ½ of priorities
    - Second to bottom, next ¼
    - …
  - Expect to only percolate up 1-2 levels