Priority Queue ADT; Heaps

CSE 332 Spring 2021

Instructor: Hannah C. Tang

Teaching Assistants:

Aayushi Modi	Khushi Chaudhari	Patrick Murphy
Aashna Sheth	Kris Wong	Richard Jiang
Frederick Huyan	Logan Milandin	Winston Jodjana
Hamsa Shankar	Nachiket Karmarkar	

gradescope

gradescope.com/courses/256241

- What is the difference between a *binary tree* and a *binary **search** tree*?

Announcements

- P1: Congrats on completing Checkpoint 1!
  - (if you didn't fill out the survey, you still can until tomorrow night (PDT)

- Reminder that we will NOT answer concept questions in office hours after the quiz is released on Tuesday
  - Get your questions in now!

Lecture Outline

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

ADTs So Far (1 of 3)

List ADT. A collection storing an ordered sequence of elements.

- Each element is accessible by a zero-based index
- A list has a size defined as the number of elements in the list
- Elements can be added to the front, back, *or any index in the list*
- Optionally, elements can be removed from the front, back, *or any index in the list*

ADTs So Far (2 of 3)

Stack ADT. A collection storing an ordered sequence of elements.

- A stack has a size defined as the number of elements in the stack
- Elements can only be added and removed from the top ("LIFO")

Queue ADT. A collection storing an ordered sequence of elements.

- A queue has a size defined as the number of elements in the queue
- Elements can only be added to one end and removed from the other ("FIFO")

ADTs So Far (3 of 3)

Set ADT. A collection of values.

- A set has a size defined as the number of elements in the set
- You can add and remove values, but the contained values are unique
- Each value is accessible via a "get" operation

Dictionary ADT. A collection of keys, each associated with a value.

- A dictionary has a size defined as the number of elements in the dictionary
- You can add and remove (key, value) pairs, but the keys are unique
- Each value is accessible by its key via a "find" or "contains" operation

A Scenario

- What is the difference between waiting for service at a pharmacy versus an ER?
  - Pharmacies usually follow the rule “First Come, First Served”
  - Emergency Rooms assign priorities based on each individual's need

A New ADT: Priority Queue

- See Weiss Chapter 6
- A **priority queue** holds *compare-able data*
  - Unlike lists, stacks, and queues, we need to *compare items*
    - Given *x* and *y:* is *x* less than, equal to, or greater than *y?*
    - Much of this course will require comparable items: e.g. sorting
  - Typically two fields: the *priority* and the *data*
- For simplicity in lecture, we'll suppose data are **`int`**s *and* that the same **`int`** value is also the priority
  - **`int`** priorities are common, but really just need `Comparable`
  - Not having "other data" is very rare
    - Example: print job has a priority *and* the file to print

Priority Queue ADT: Intro

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`**

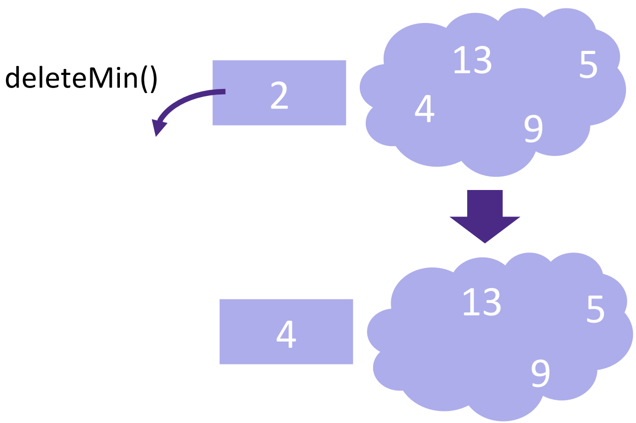
Key property:

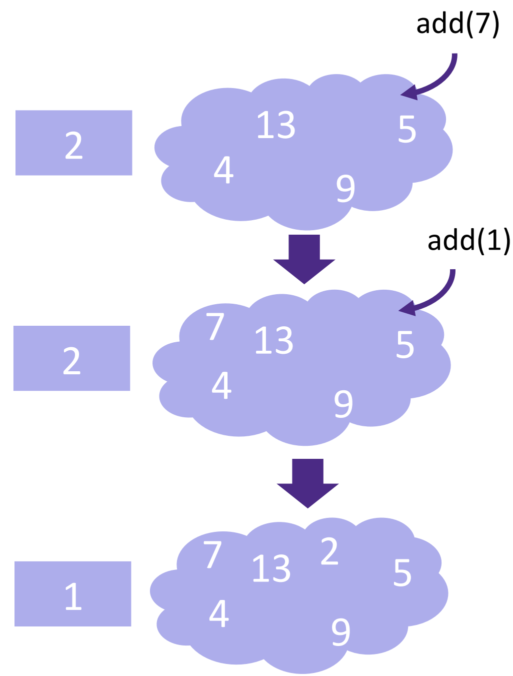
- **`deleteMin`** removes and returns the “most important” item (lowest priority value)
- Can resolve ties arbitrarily

Priority Queue ADT: Functionality

- In lecture, we will study **min priority queues** but you may also see **max priority queues**
  - Same as minPQs, but invert the priority

- In a PQ, the <u>only</u> item that matters is the min (or max)





Priority Queue ADT: 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*

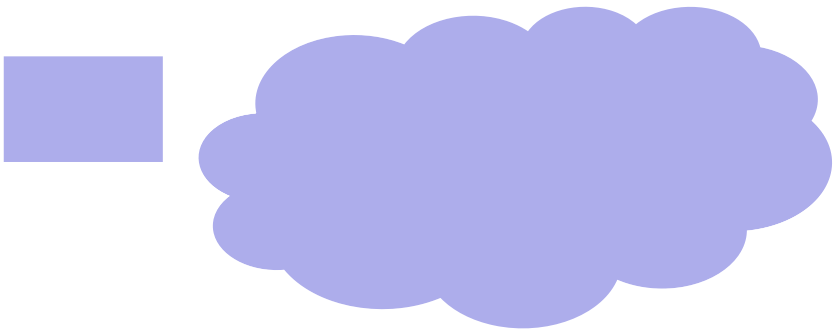
`add` *e* with priority *6*

y = **`deleteMin`**

z = **`deleteMin`**

```
after execution:
6->e

w = b
x = c
y = d
z = a
```



Priority Queue ADT: 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

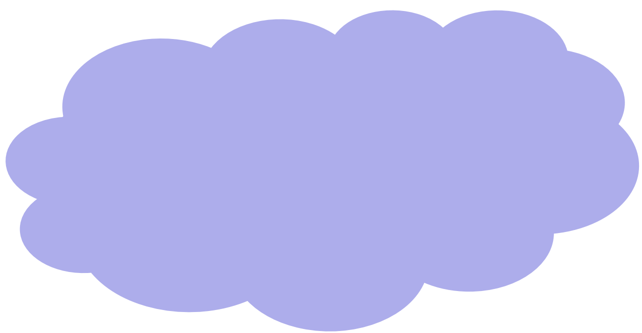
w = b

x = c

y = d

z = a

6



gradescope

gradescope.com/courses/256241

- How do Priority Queues differ from Queues? How can you implement a Queue using a Priority Queue?

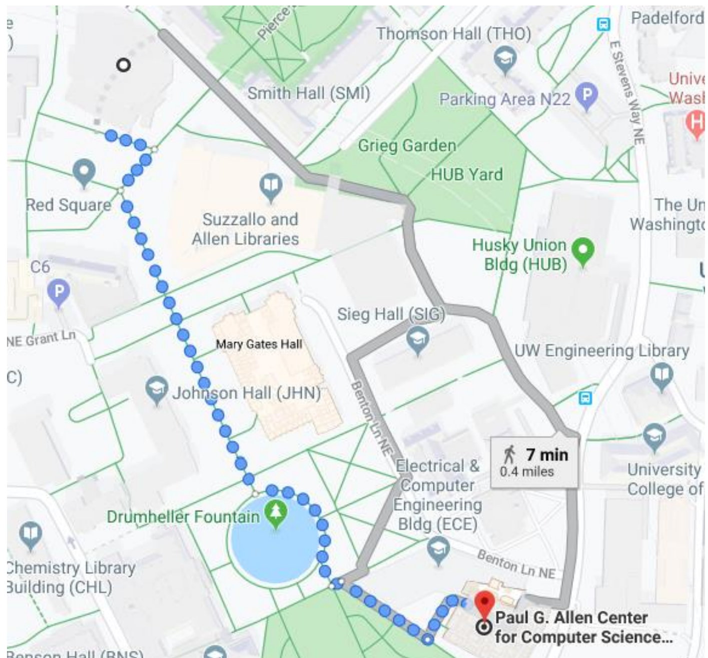
Priority Queue ADT: Applications

- Run multiple programs in the operating system
  - “critical” before “interactive” before “compute-intensive”
- Triage (or treat) hospital patients in order of severity
- Order print jobs (by increasing length?)
- Forward network packets by order of urgency
- Identify most frequently-used symbols for data compression
- Sorting!
  - **`add`** all elements, then repeatedly **`deleteMin`**

Priority Queue ADT: More Applications

- Used heavily in **greedy algorithms**, where each phase of the algorithm picks the locally optimum solution

- Example: route finding
  - Represent a map as a series of *segments*
  - At each intersection, ask which segment gets you closest to the destination (ie, has max priority or min distance)



Priority Queue ADT: Possible Data Structures

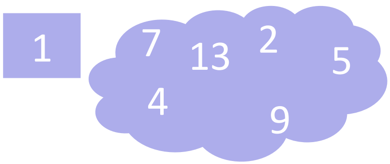
	add	deleteMin
Unsorted Array	O(1)	O(N)
Unsorted Singly-linked Linked List		
Sorted Circular Array	O(N)	O(1)
Sorted Doubly-linked Linked List		
Binary Search Tree (BST)		

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



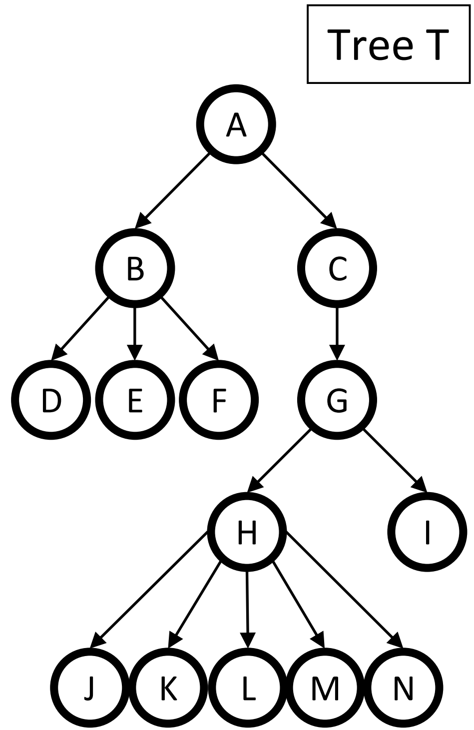
- We *visualize* our heap as a tree, so let's review some terminology

Lecture Outline

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

Review: Tree Terminology

- root(T):
- leaves(T):
- children(B):
- parent(H):
- siblings(E):
- ancestors(F): B, A
- descendants(G):
- subtree(G):
- depth(B):
- height(G):
- height(T): 4
- degree(B):
- branching factor(T):



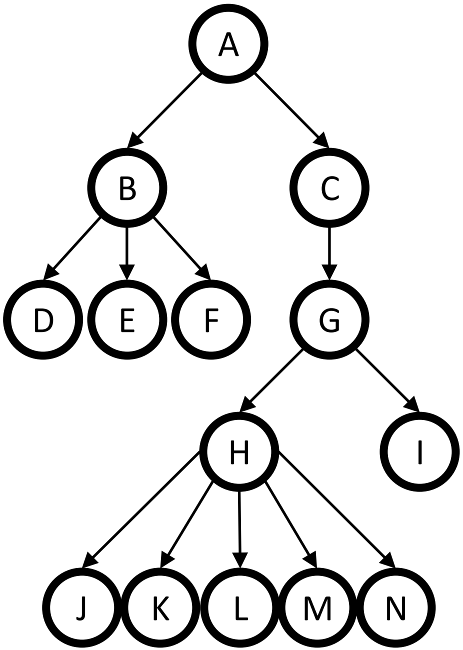
gradescope

gradescope.com/courses/256241

- siblings(E): D, F
- height(T): 4
- branching factor(T):

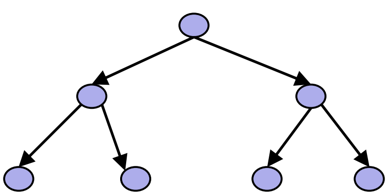
avg/mean: $\frac{13}{6}$

max: 5

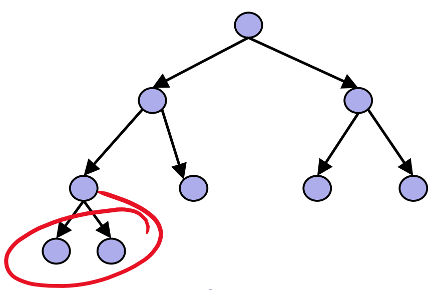


Types of Trees

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



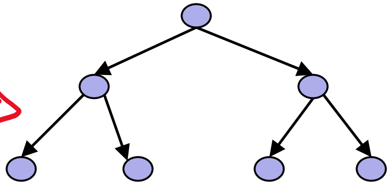
Perfect Tree



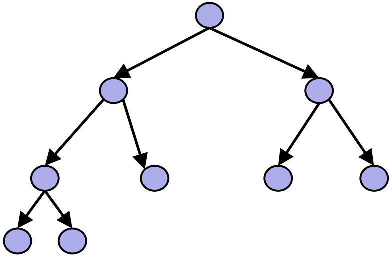
Complete Tree

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



Perfect Tree



Complete Tree

Lecture Outline

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

Our Data Structure: 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

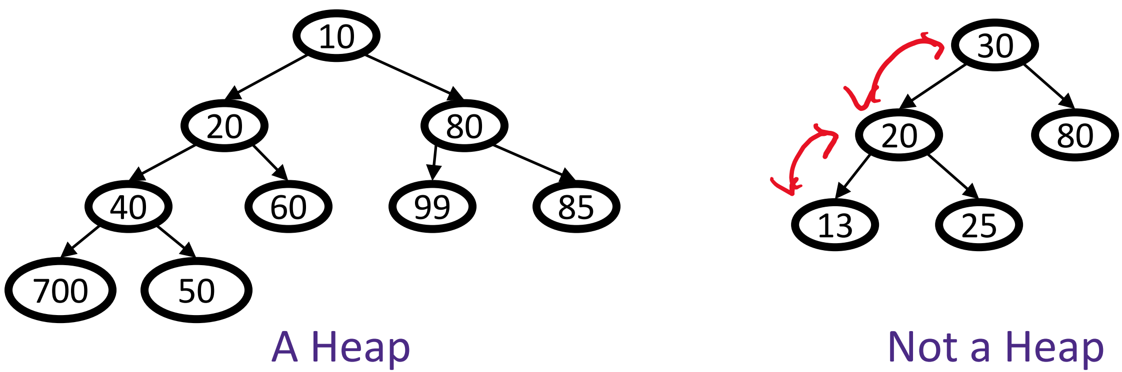
- **Structure Property**: A complete binary tree
- **Order Property**: Every non-root node has a priority value larger than (or possibly equal to) the priority of its parent

How is this different from a binary search tree?

Our Data Structure: 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

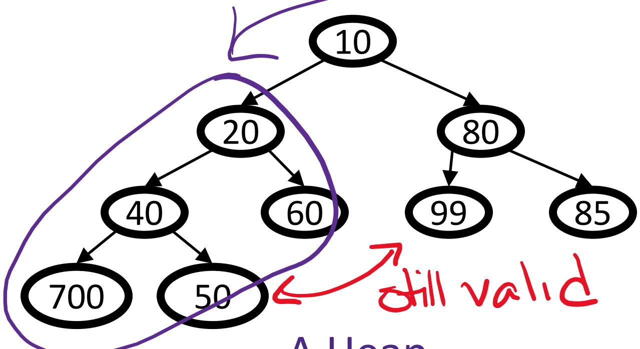


Our Data Structure: Binary (Min-)Heap (3 of 3)

- Where is the minimum priority item? root!
- What is the height of a heap with n items? $\lfloor \log_2 n \rfloor$
- Is this tree unique to this heap?

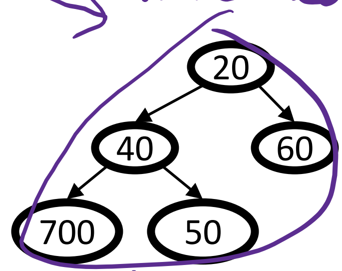
No!

subheap still a valid heap



still valid

A Heap

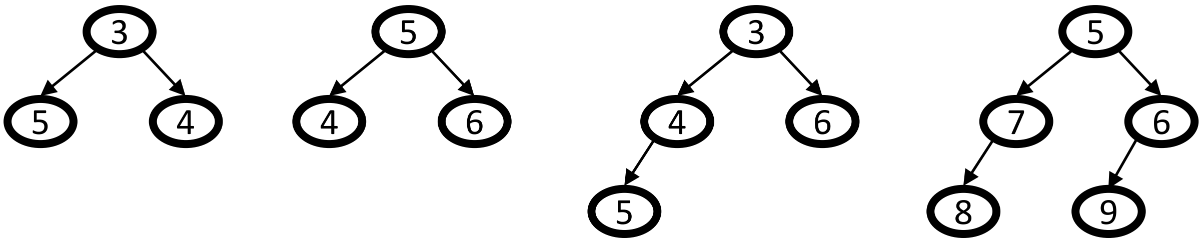


Also a Heap

gradescope

gradescope.com/courses/256241

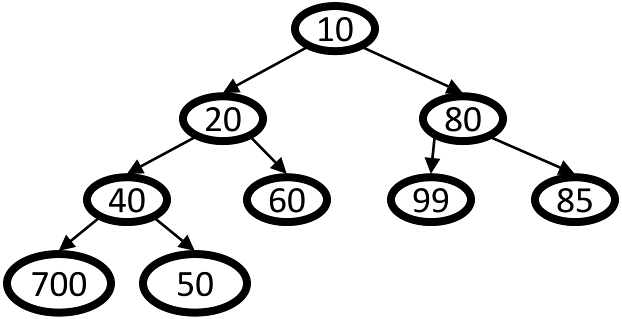
- Are these valid binary min-heaps?



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 …

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*)



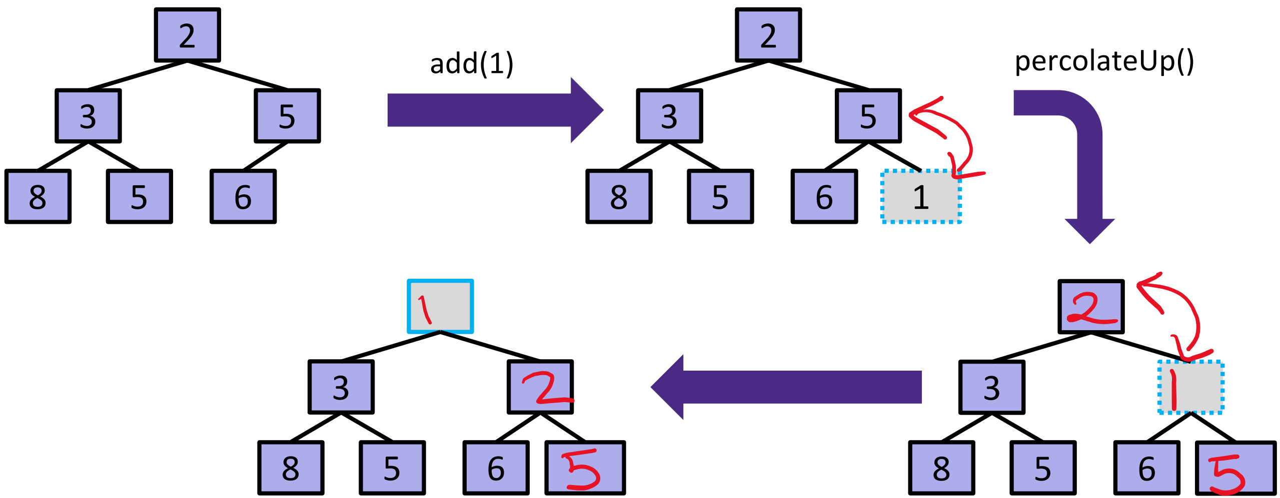
- **`deleteMin`**:
  - `answer = root.item`
  - Move rightmost node in last row to root (*restore structure property*)
  - "Percolate down" to correct layer (*restore order property*)

Overall strategy:

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

Binary Heap: add()

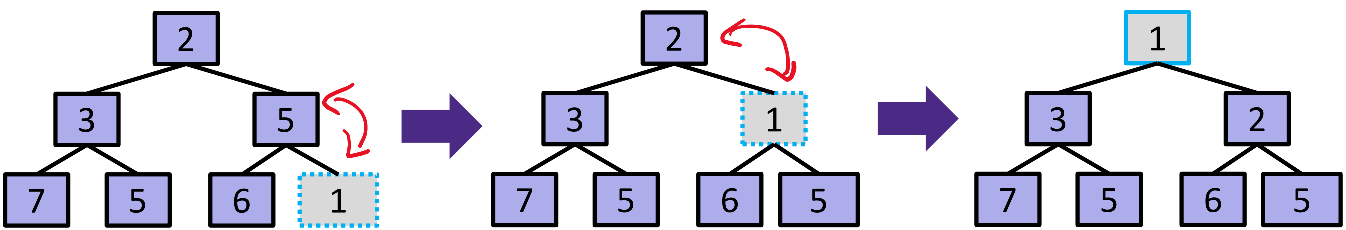
- Put new node in rightmost position of the last row
- "Percolate up" to correct layer



percolateUp() Helper Function

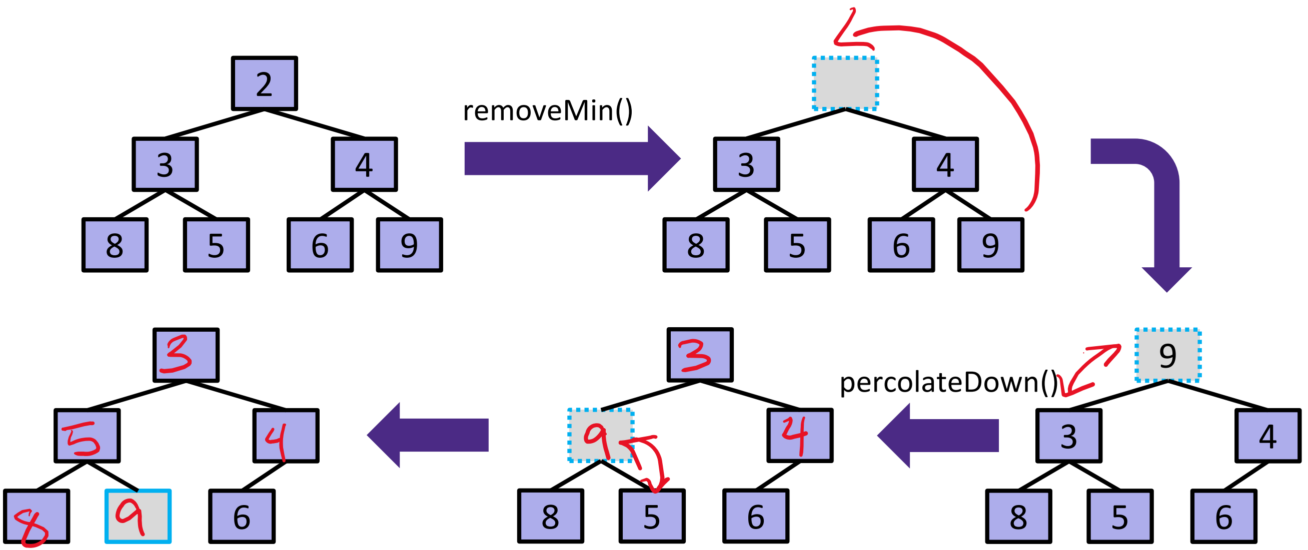
- percolateUp():
  - Put new item in new location
  - If parent larger, swap with parent, and continue
  - Done when parent ≤ item or reached root

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



Binary Heap: removeMin()

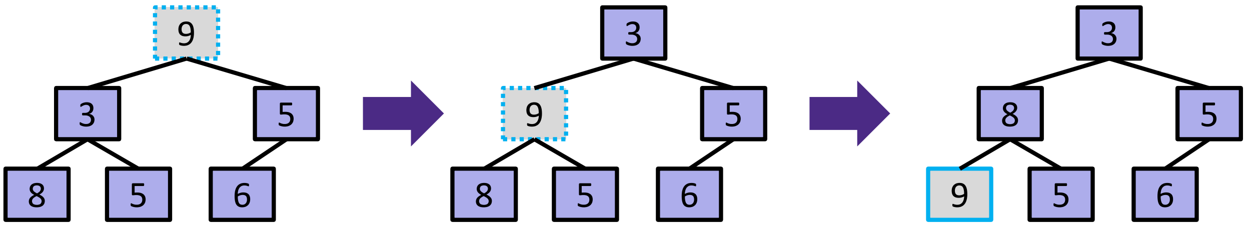
- Move rightmost node in last row to the root
- "Percolate down" to correct layer



percolateDown() Helper Function

- 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?



Lecture Outline

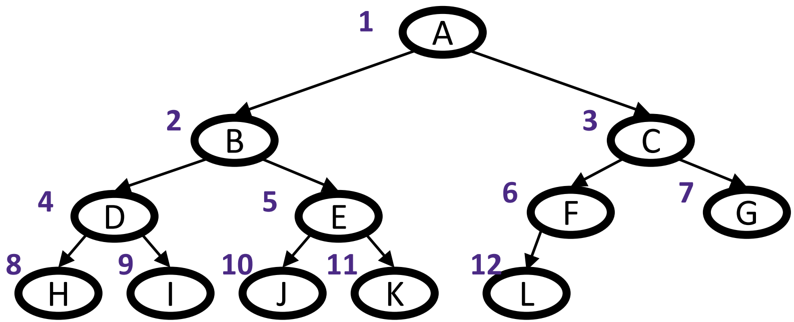
- Priority Queue ADT
- Tree Terminology and Properties
- Binary Heap
  - Tree Visualization and Operations
  - **Array Representation**

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

- In lecture and in Weiss, skip index 0 to make the math simpler
  - Though, it's a good place to store the current size of the heap
  - P1 doesn't skip; starts counting from 0



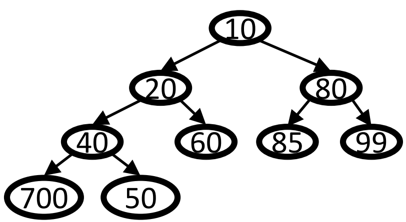
- From node i:
  - left child: $2i$
  - right child: $2i+1$
  - parent: $\left\lfloor \frac{i}{2} \right\rfloor$

	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;
}
```



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

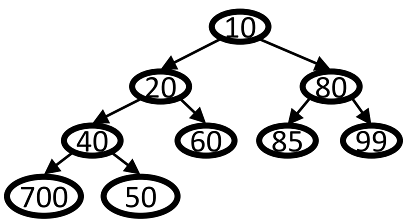
Disclaimers:

- This pseudocode uses ints. In real use, you will have nodes with priorities and values
- P1 doesn't skip; starts counting from 0

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;
}
```



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

gradescope

gradescope.com/courses/256241

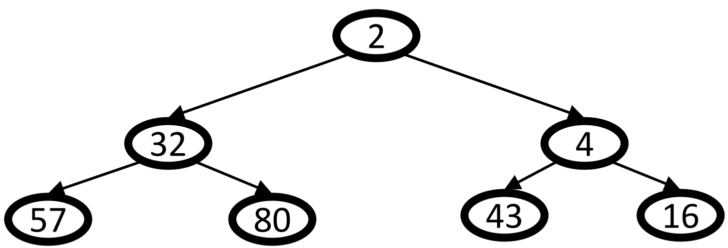
1. add: 16, 32, 4, 57, 80, 43, 2
2. deleteMin

0	1	2	3	4	5	6	7

Activity Answer: After add()s

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



Activity Answer: After deleteMin()

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