Priority Queues and Heaps

CSE 332 Summer 2020

Instructor: Richard Jiang

Teaching Assistants:

Hamsa Shankar Kristin Li Winston Jodjana

Maggie Jiang Hans Zhang Michael Duan

Jeffery Tian Annie Mao

Lecture Q&A: pollev.com/332summer

Lecture clarifications: tinyurl.com/332-06-29A

Announcements

- P1: Congrats on completing Checkpoint 1!
  - Next due date is Tue, July 7th or the entire project
- Ex 2 - 3 on Monday July 6th
  - Start on Ex 3 early!

pollev.com/332summer :: tinyurl.com/332-06-29A

Lecture Outline

- Priority Queue ADT
  - Introduction
  - Application
  - Possible Data Structure
- Tree Terminology Review
- Binary Heap
  - Tree Visualization and Operations
  - Array Representation

pollev.com/332summer :: tinyurl.com/332-06-29A

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*

- Aside: we will use integers as priority and data
  - For simplicity in lecture, we'll suppose data are **`int`**s *and* that 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

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:

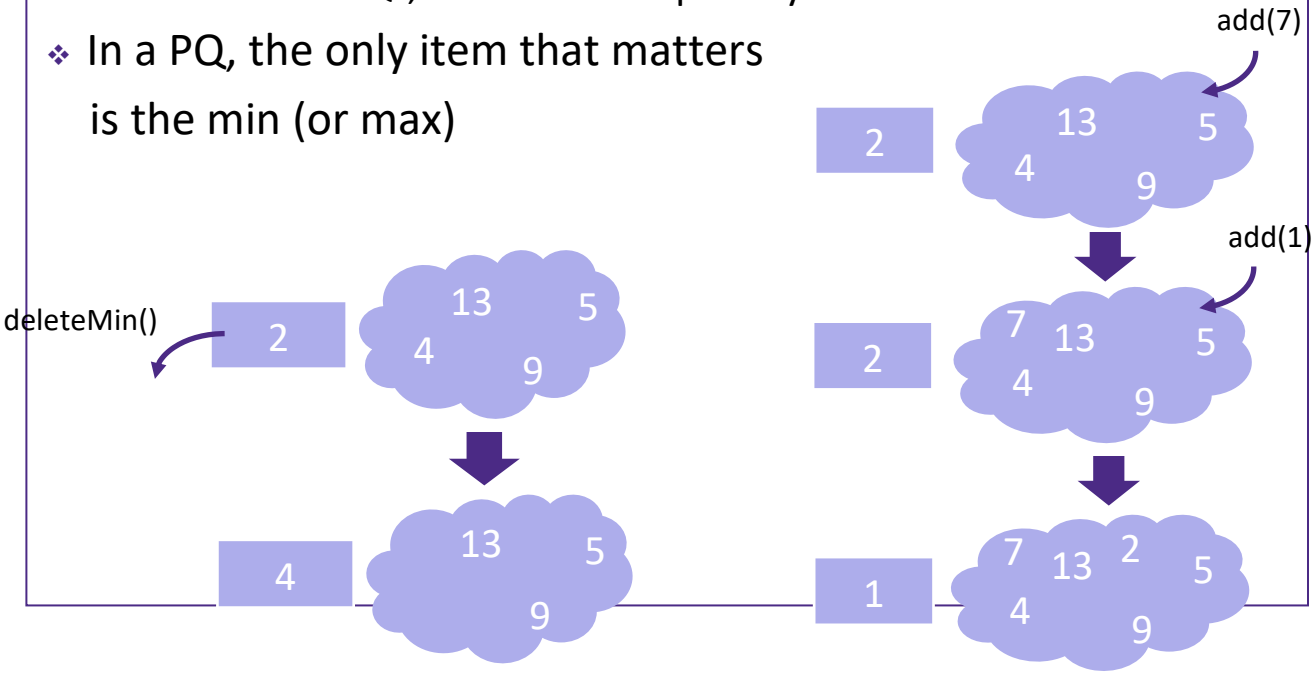
- **`add`**
- **`deleteMin`**

Key property:

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

Priority Queues

- 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 only item that matters is the min (or max)



Priority Queue: Example

add *a* with priority *5*

add *b* with priority *3*

add *c* with priority *4*

w = **deleteMin**

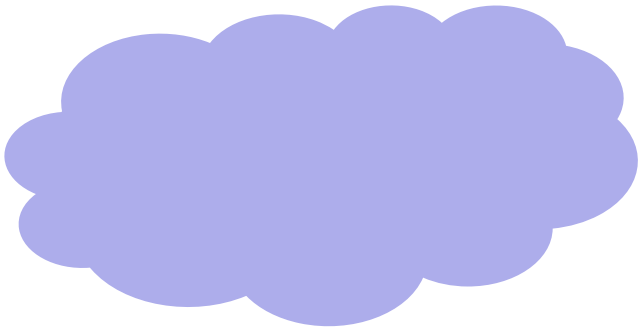
x = **deleteMin**

add *d* with priority *2*

add *e* with priority *6*

y = **deleteMin**

z = **deleteMin**



Analogy: `add` is like `enqueue`, and `deleteMin` is like `dequeue`

Unlike queues, priority queues use *priorities* instead of *time-of-insertion* to order its elements

Lecture Outline

- Priority Queue ADT
  - Introduction
  - Application
  - Possible Data Structure
- Tree Terminology Review
- Binary Heap
  - Tree Visualization and Operations
  - Array Representation

pollev.com/332summer :: tinyurl.com/332-06-29A

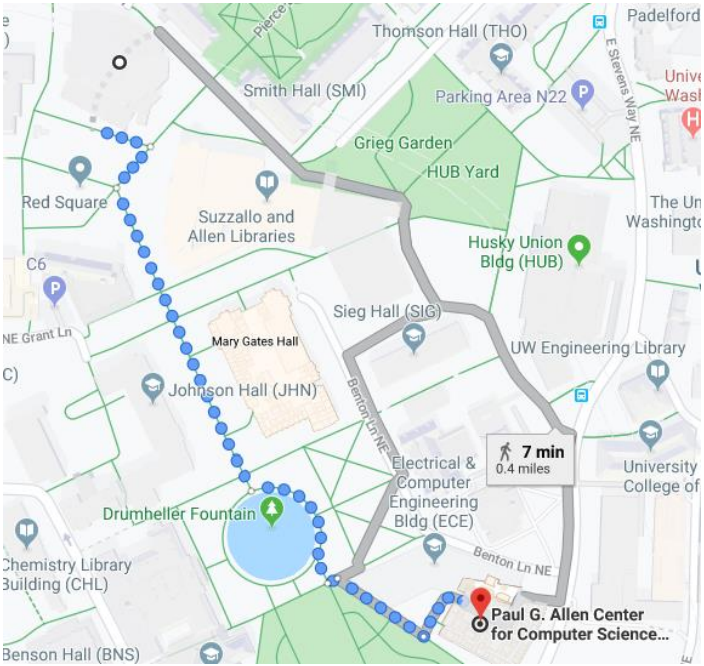
Priority Queue: Applications

- Run multiple programs in the operating system
  - "critical" before "interactive" before "compute-intensive"
- Triage (or treat) hospital patients in order of severity
- Software bugs in industry
- Forward network packets by order of urgency
- Identify most frequently-used symbols for data compression
- Sorting!
  - **`add`** all elements, then repeatedly **`deleteMin`**

Priority Queue: 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)



Lecture Outline

- Priority Queue ADT
  - Introduction
  - Application
  - Possible Data Structure
- Tree Terminology Review
- Binary Heap
  - Tree Visualization and Operations
  - Array Representation

pollev.com/332summer :: tinyurl.com/332-06-29A

Possible Data Structures

	add	deleteMin
Unsorted Array		
Unsorted LinkedList		
Sorted Circular Array		
Sorted Linked List		
Binary Search Tree (BST)		

Possible Data Structures

	add	deleteMin
Unsorted Array	add at end: O(1)	search: O(N)
Unsorted LinkedList	add at front: O(1)	search: O(N)
Sorted Circular Array	search + shift: O(N)	move front pointer: O(1)
Sorted Linked List	search + insert: O(N)	update front pointer: O(1)
Binary Search Tree (BST)	put in correct place: O(h) = O(N)	remove leftmost: O(h) = O(N)

Assumptions: Worst case; Arrays have enough space

Observations

- If priorities are inserted in *random order*, binary search tree will likely do better than $O(n)$
  - Both **`add`** and **`deleteMin`** become O(log n) expected

- If priorities range between [0, *k*], can use *array of lists*
  - **`add`**: insert to front of list at **`arr[priority]`** - O(1)
  - **`deleteMin`**: remove from lowest non-empty list - O(k)

- Runtime trade-off between **`add`** and **`deleteMin`**
  - if **`add`** is $O(n)$, then **`deleteMin`** is $O(1)$ and vice versa

Priority Queue: Applications

- Run multiple programs in the operating system
  - "critical" before "interactive" before "compute-intensive"
- Triage (or treat) hospital patients in order of severity
- Software bugs in industry
- Forward network packets by order of urgency
- Identify most frequently-used symbols for data compression
- Sorting!
  - **`add`** all elements, then repeatedly **`deleteMin`**

Observations

- If priorities are inserted in *random order*, binary search tree will likely do better than $O(n)$
  - Both **`add`** and **`deleteMin`** become O(log n) expected

- If priorities range between [0, *k*], can use *array of lists*
  - **`add`**: insert to front of list at **`arr[priority]`** - O(1)
  - **`deleteMin`**: remove from lowest non-empty list - O(k)

- Runtime trade-off between **`add`** and **`deleteMin`**
  - if **`add`** is $O(n)$, then **`deleteMin`** is $O(1)$ and vice versa

Our Eventual Data Structure: The Heap

- **Heap:**
  - `add`: O(log n), worst case
  - `deleteMin`: O(log n), worst case

- **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

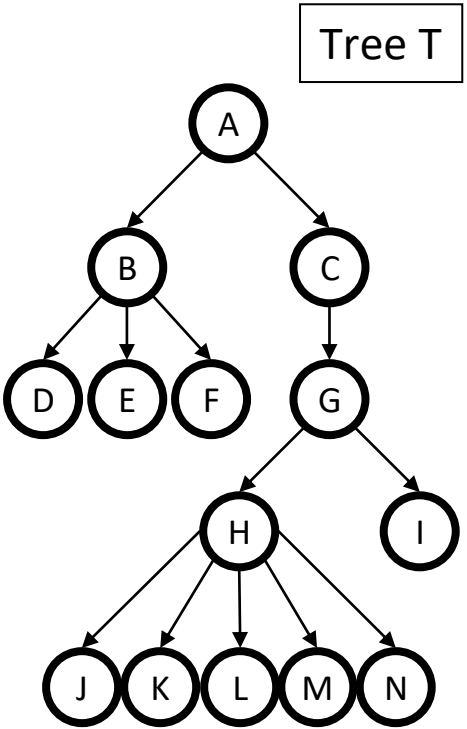
Lecture Outline

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

pollev.com/332summer :: tinyurl.com/332-06-29A

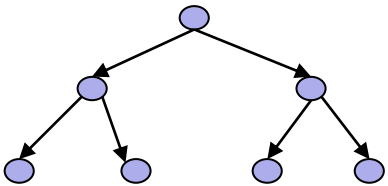
Review: Tree Terminology

- root(T):
- leaves(T):
- children(B):
- parent(H):
- siblings(E):
- ancestors(F):
- descendents(G):
- subtree(G):
- depth(C):
- height(C):
- height(T):
- degree(B):
- branching factor(T):

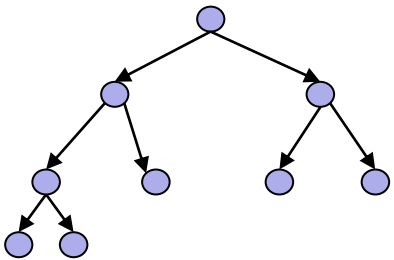


Types of Trees

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



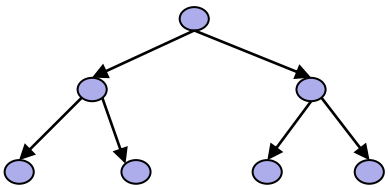
Perfect Tree



Complete Tree

Perfect Tree Properties

Height	Number of Nodes	Number of Leaves
1		
2		
3		
4		
h		



Perfect Tree

Lecture Outline

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

pollev.com/332summer :: tinyurl.com/332-06-29A

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

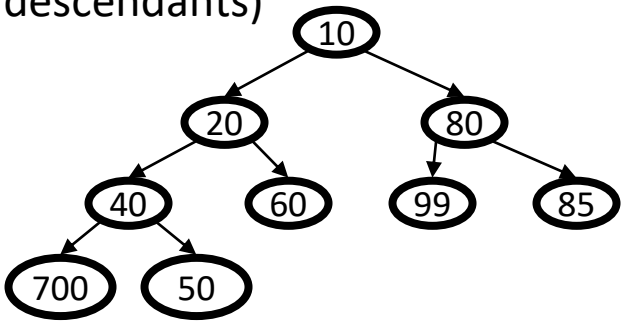
- **Structure Property**: A complete binary tree
- **Order Property**: Every parent node has a priority value smaller than (or possible equal to) the priority of all its children (and descendants)

How is this different from a binary search tree?

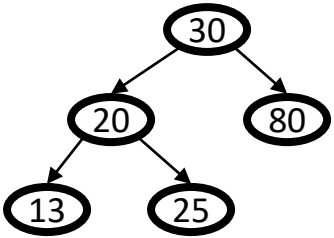
Binary (Min-)Heap (2 of 3)

- 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 parent node has a priority value smaller than (or possible equal to) the priority of all its children (and descendants)



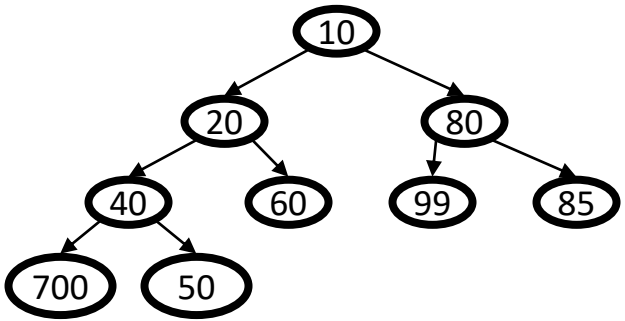
A Heap



Not a Heap

Binary (Min-)Heap (3 of 3)

- Where is the minimum priority item located?
- What is the height of a heap with n items?
- Is this tree unique to this heap?

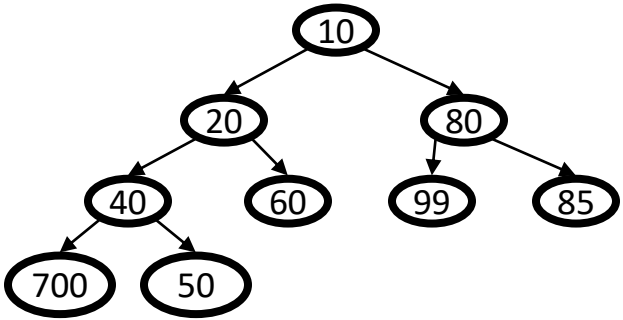


A Heap

Binary Heap Helper Functions

- **`add`**:
  - Put new node in empty leftmost 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*)

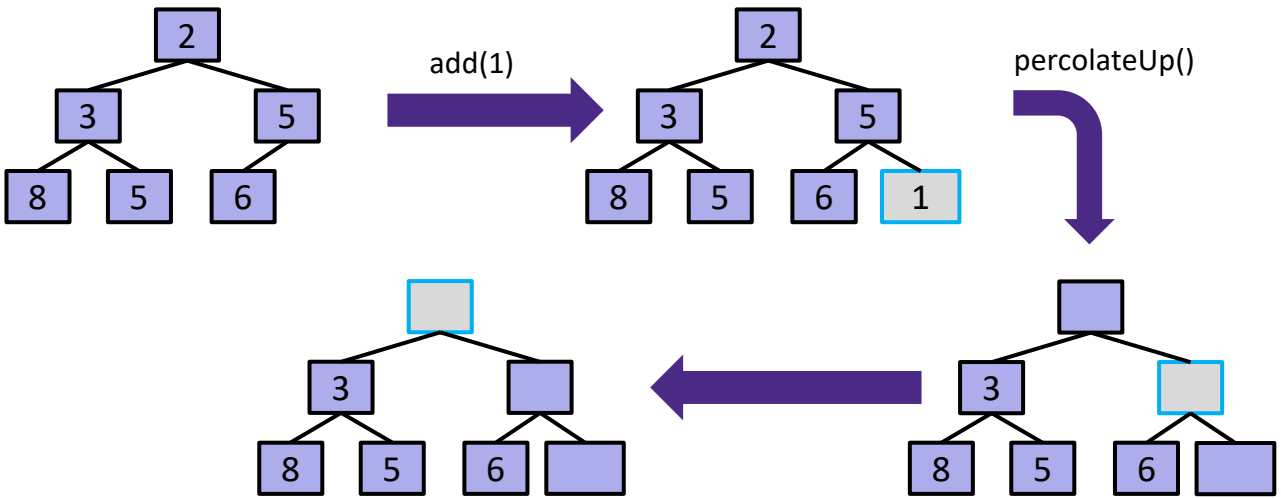


Overall strategy:

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

Binary Heap: add()

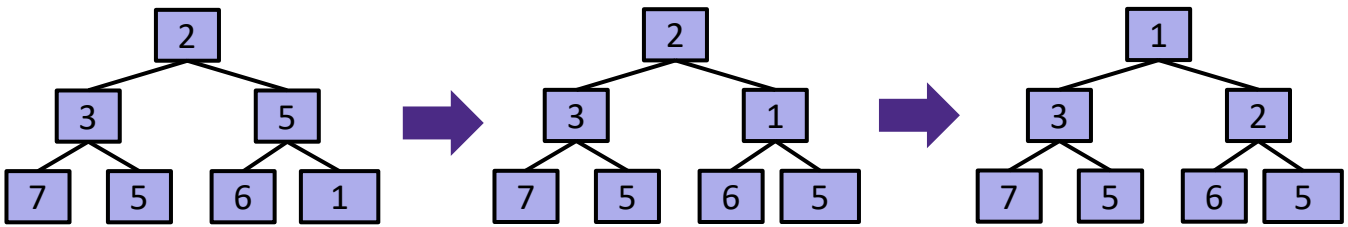
- Put new node in rightmost position of the last row
- "Percolate up" to correct layer



percolateUp()

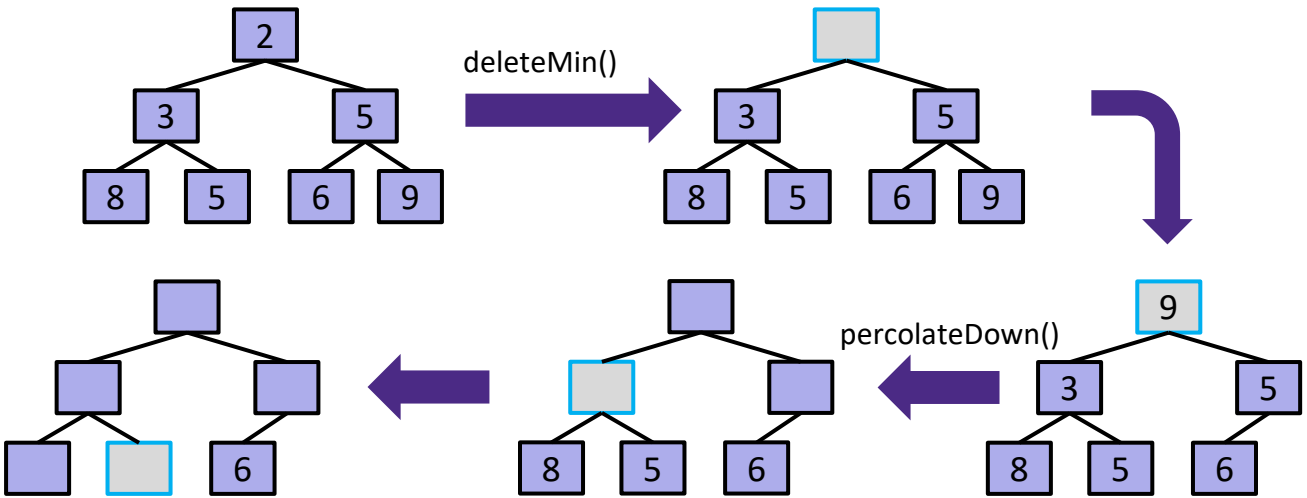
- percolateUp():
  - Put new item in new location
  - If parent larger, swap with parent, and continue
  - Done if parent $\leq$ item or reached root

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



Binary Heaps: deleteMin()

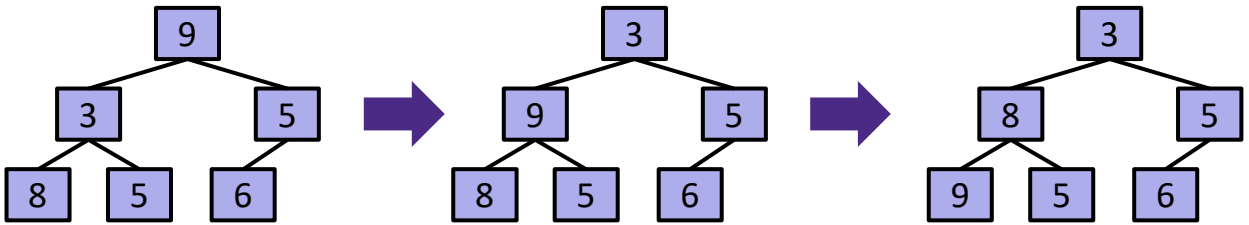
- Move rightmost node in last row to root
- "Percolate down" to correct layer



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?



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!!

Lecture Outline

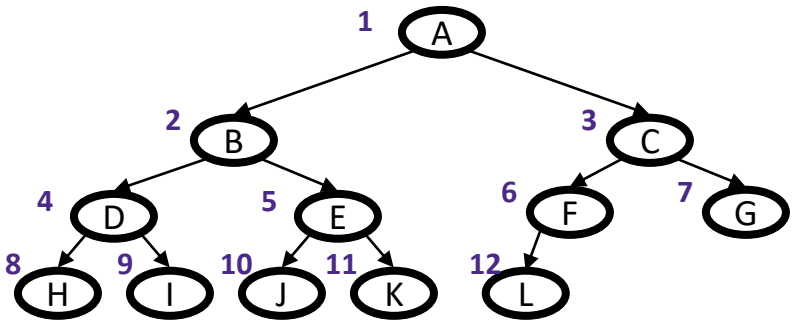
- Priority Queue ADT
- Tree Terminology Review
- Binary Heap
  - Tree Visualization and Operations
  - Array Representation

pollev.com/332summer :: tinyurl.com/332-06-29A

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:
  - right child:
  - parent:

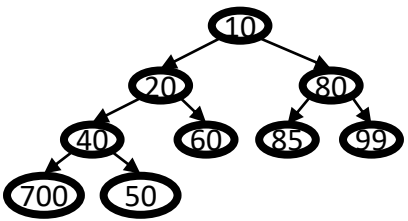


	A	B	C	D	E	F	G	H	I	J	K	L	
0	1	2	3	4	5	6	7	8	9	10	11	12	13

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:

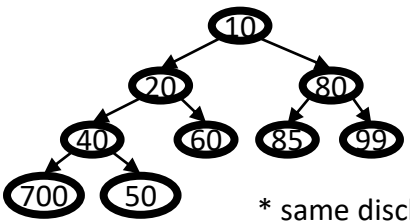
- This pseudocode uses ints. In real use, you will have nodes with priorities and values
- Exercises and P1 start counting from 0

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

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;
}
```

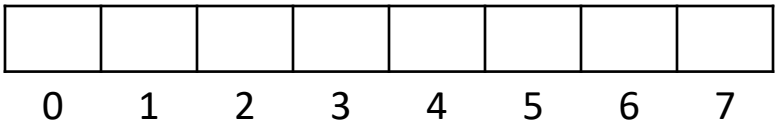


* same disclaimers apply

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

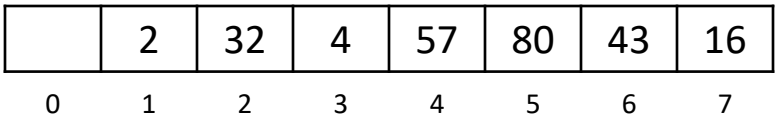
Example

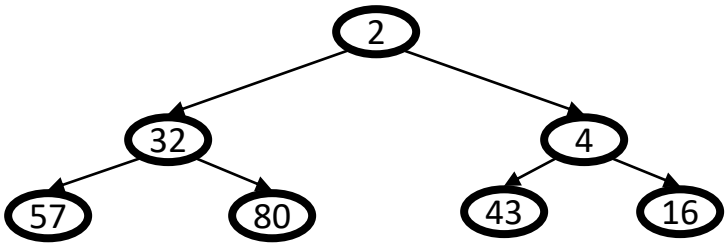
1. add: 16, 32, 4, 57, 80, 43, 2
2. deleteMin



Example: After insertion

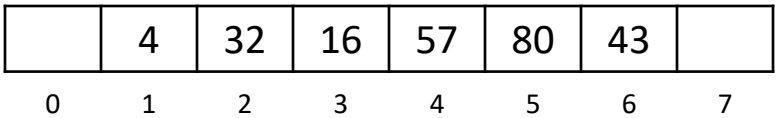
1. add: 16, 32, 4, 57, 80, 43, 2
2. deleteMin

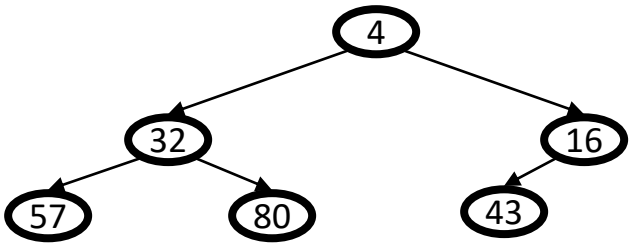




Example: After deletion

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





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)

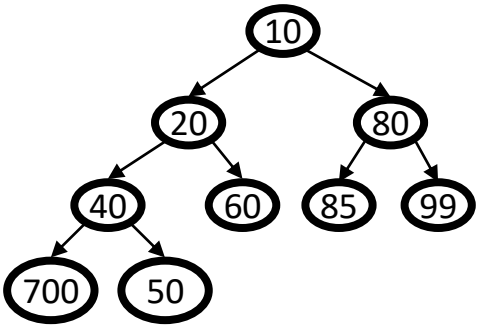
- Yes, `add`'s worst case is O(log n)
  - It all depends on the order the items are inserted
  - What is the worst case order?

- 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

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

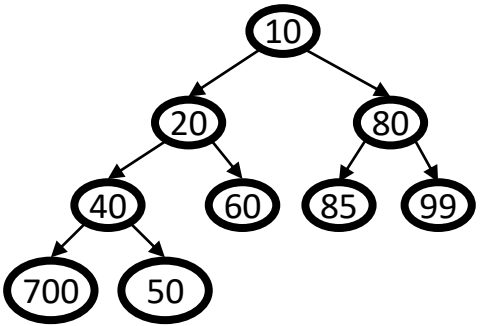
- Yes, `add`'s worst case is O(log n)
  - It all depends on the order the items are inserted
  - What is the worst case order?

- 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 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 priority 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 value; 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