Priority Queue ADT; Heaps

CSE 332 Summer 2021

Instructor: Kristofer Wong

Teaching Assistants:

Alena Dickmann	Arya GJ	Finn Johnson
Joon Chong	Kimi Locke	Peyton Rapo
Rahul Misal	Winston Jodjana	

Announcements

- Check Ed post for ex2 clarification

Lecture Outline

- **Priority Queue ADT**

- Binary Heap
  - Tree Visualization and Operations
  - Array Representation
  - BuildHeap

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`**

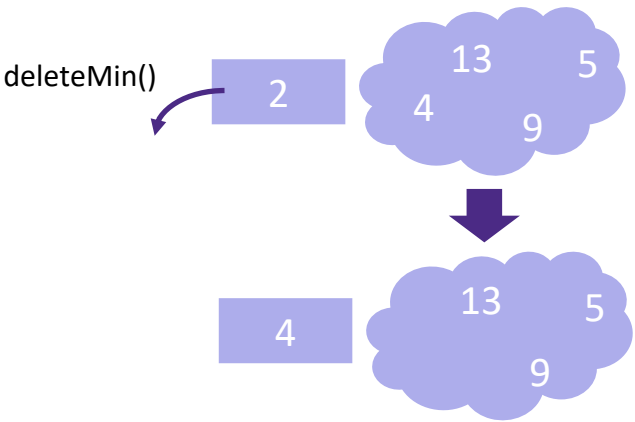
Key property:

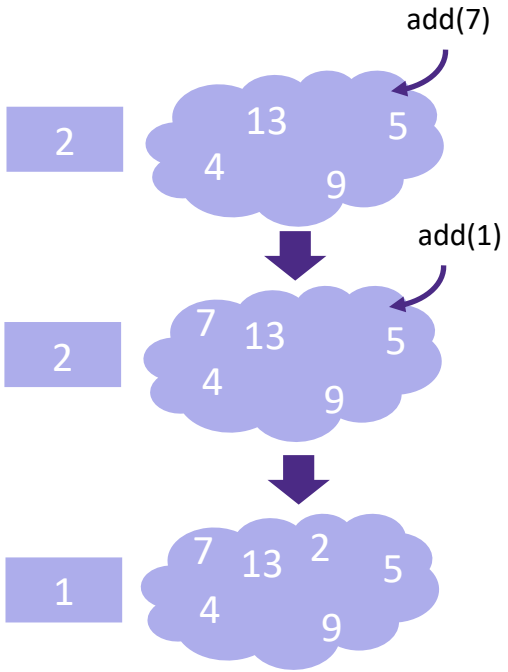
- **`deleteMin`** removes and returns the "most important" item (lowest priority value)
- Can resolve ties arbitrarily

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 only item that matters is the min (or max)



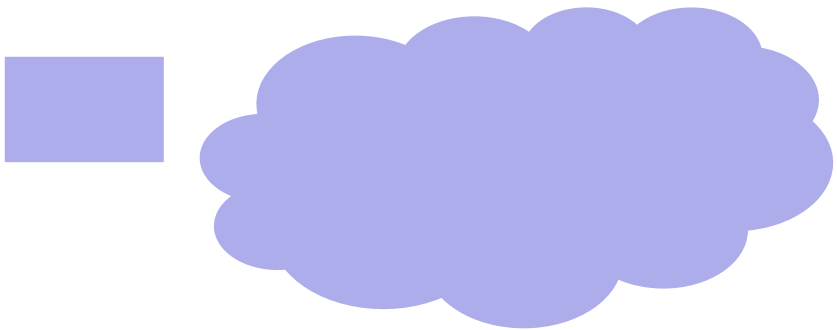


Priority Queue ADT: Example

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
```



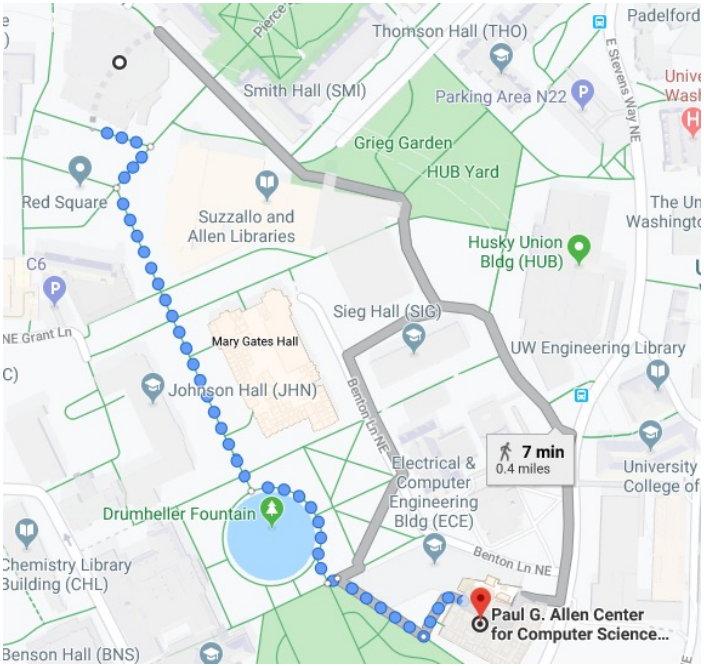
Priority Queue ADT: Applications

- 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		
Unsorted Singly-linked Linked List		
Sorted Circular Array		
Sorted Doubly-linked Linked List		
Binary Search Tree (BST)		

Assumptions: Worst case; Arrays have enough space

New 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

Lecture Outline

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

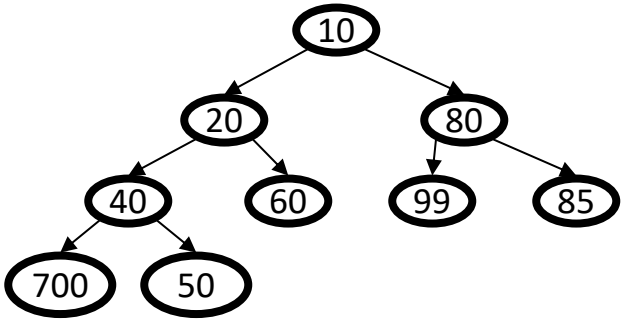
Binary Heap

- Most commonly known as a *binary heap* or simply a *heap*
  - For this lecture, we'll discuss Min-Heaps
  - This just means we'll prioritize keys with smaller values
  - Max-Heaps on the other hand track the largest priority
    - You can apply everything we talk about today to max heaps
- **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
  - Opposite in Max-Heap

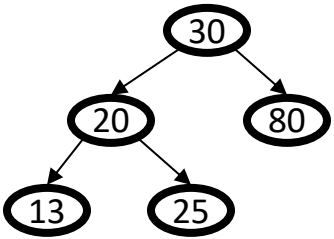
Our Data Structure: Binary Min-Heap

- 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



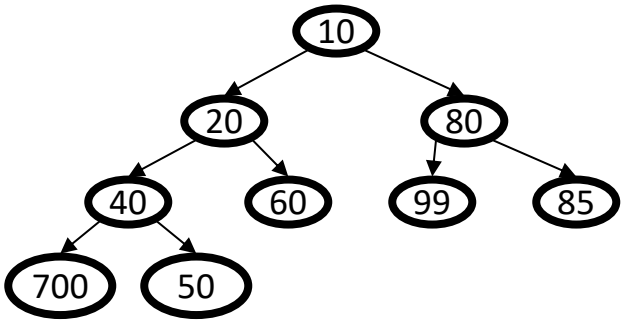
A Heap



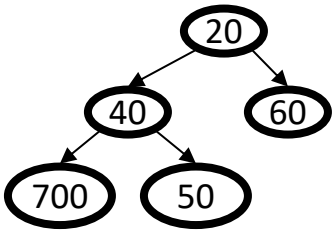
Not a Heap

Our Data Structure: Binary Min-Heap

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



A Heap

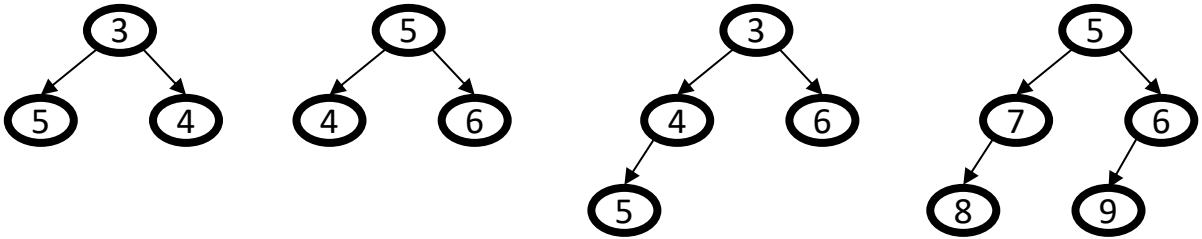


Also a Heap

gradescope

gradescope.com/courses/275833

- Are these valid binary min-heaps?

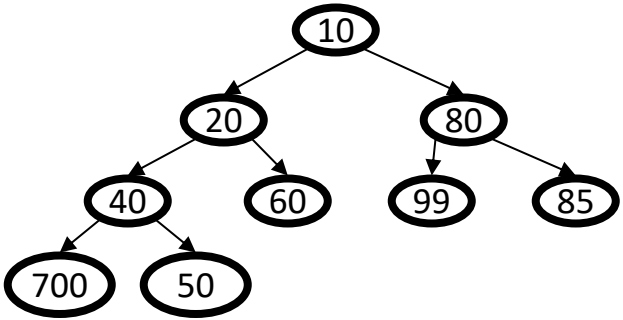


A. Yes, no, yes, yes
B. Yes, yes, yes, yes
C. Yes, no, no, yes
D. Yes, no, yes, no
E. No, no, yes, no
F. I'm not sure …

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*)

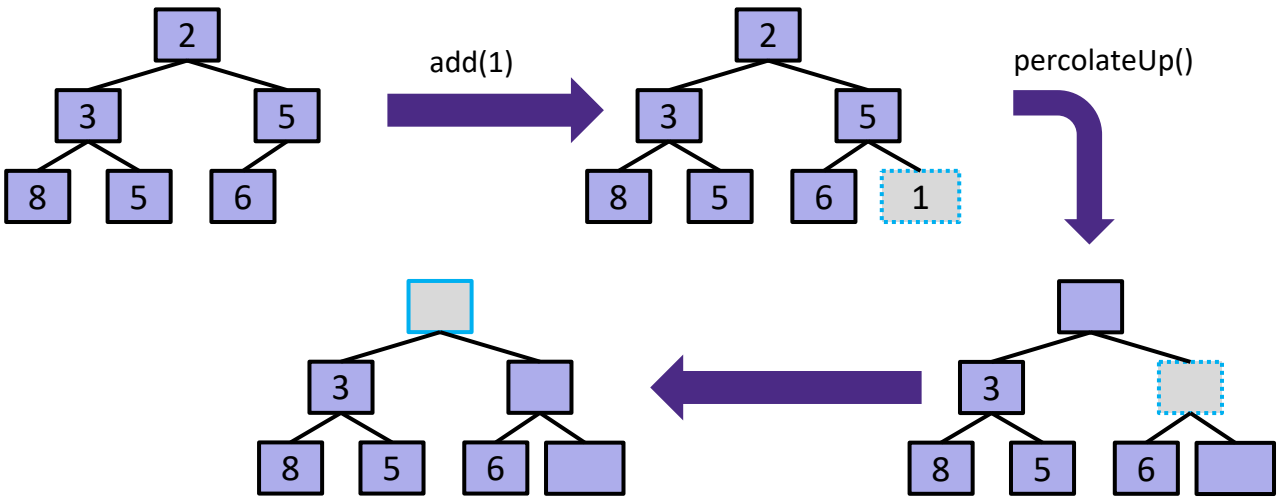


Overall strategy:

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

Binary Heap: add()

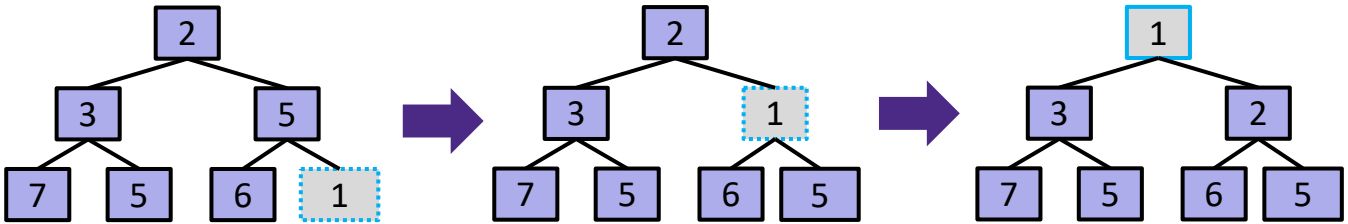
- Put new node in rightmost position of the last row
- "Percolate up" to correct layer



percolateUp() Helper Function

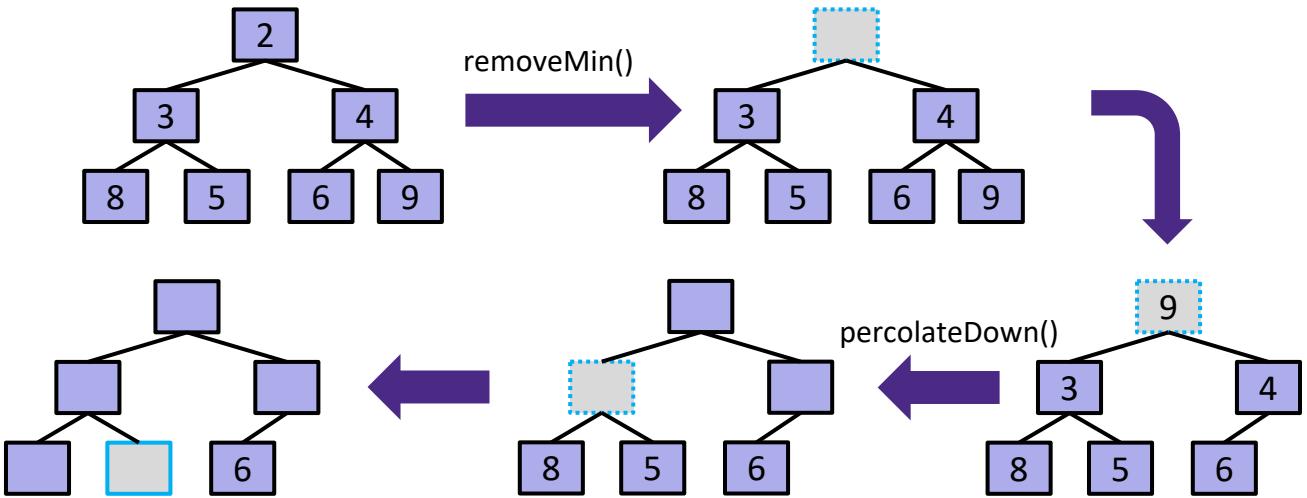
- percolateUp():
  - Put new item in new location
  - If parent larger, swap with parent, and continue
  - Done when parent ≤ item or reached root

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



Binary Heap: removeMin()

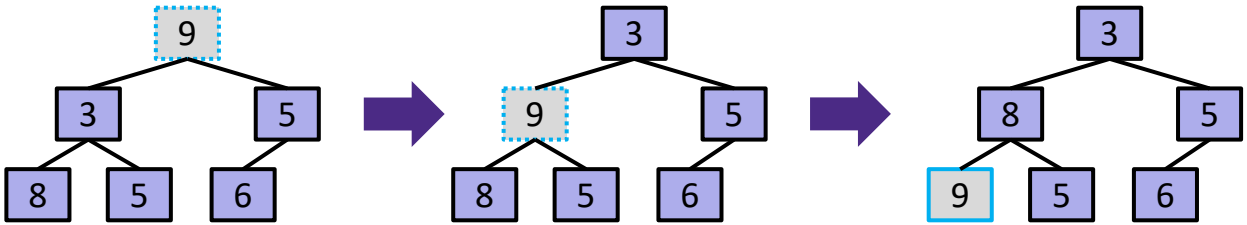
- Move rightmost node in last row to the root
- "Percolate down" to correct layer



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
- Binary Heap
  - Tree Visualization and Operations
  - **Array Representation**
  - BuildHeap

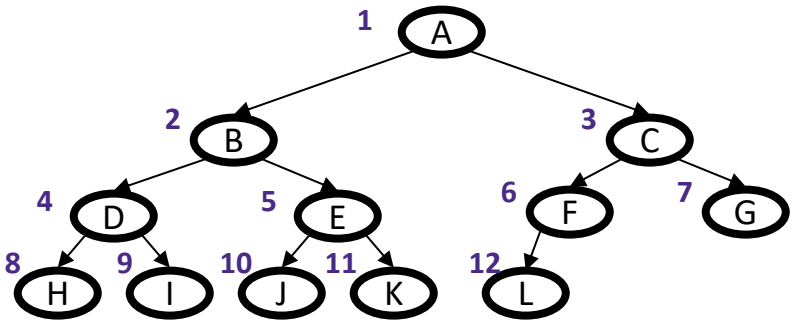
A Clever Trick for Storing the Heap…

- 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:
  - right child:
  - parent:

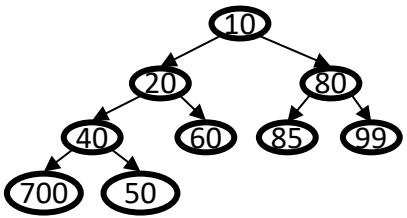


	A	B	C	D	E	F	G	H	I	J	K	L	
0	1	2	3	4	5	6	7	8	9	10	11	12	13

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:

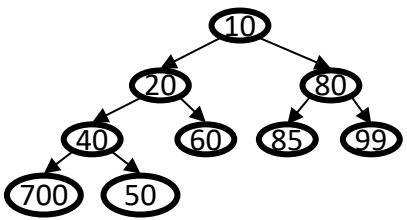
- This pseudocode uses ints. In real use, you will have nodes with priorities and values
- P2 doesn't skip 0 index; starts counting from 0

	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;
}
```

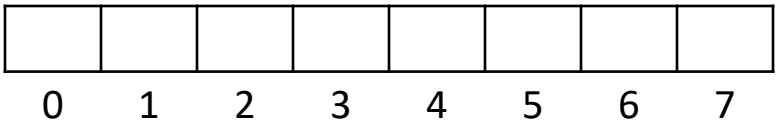


	10	20	80	40	60	85	99	700	50				
0	1	2	3	4	5	6	7	8	9	10	11	12	13

gradescope

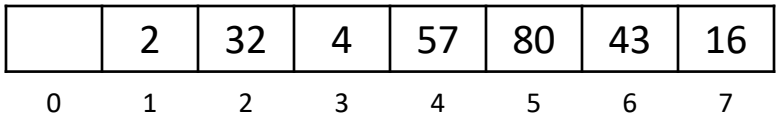
gradescope.com/courses/275833

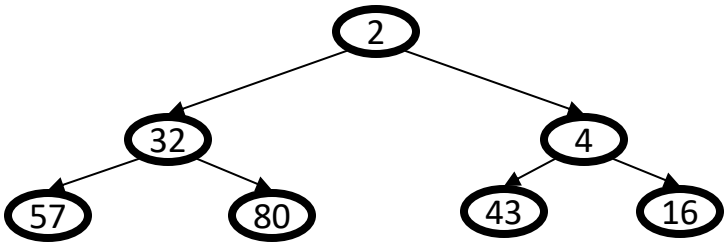
1. add: 16, 32, 4, 57, 80, 43, 2
2. deleteMin



Activity Answer: After add()s

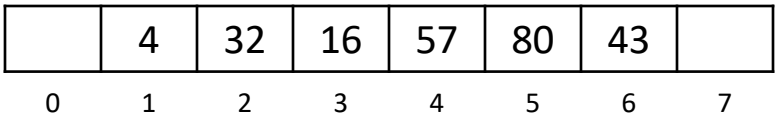
1. add: 16, 32, 4, 57, 80, 43, 2
2. deleteMin

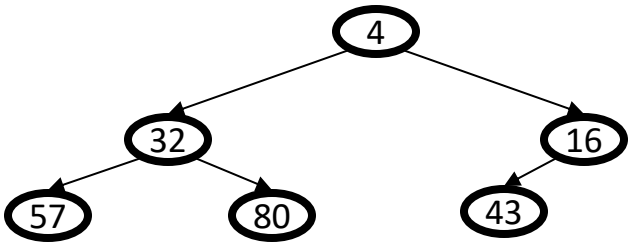




Activity Answer: After deleteMin()

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





Lecture Outline

- Priority Queue ADT
- Binary Heap
  - Tree Visualization and Operations
  - Array Representation
  - **BuildHeap**
  - Heap Wrapup

Other Operations

- **`decreasePriority`**: given pointer to object in priority queue (e.g., its array index), lower its priority by p
  - Change priority and percolate up

- **`increasePriority`**: given pointer to object in priority queue (e.g., its array index), raise its priority by p
  - Change priority and percolate down

- **`remove`**: given pointer to object in priority queue (e.g., its array index), remove it from the queue
  - `decreaseKey` with p = ∞, then `deleteMin`

- Running time for all these operations?

One Final Operation: buildHeap

- `buildHeap`() takes an array of size N and applies the heap-ordering principle to it

- Naïve implementation:
  - Start with an empty array (representing an empty binary heap)
  - Call `add`() N times
  - Runtime: ??

- Can we do better?
  - If we only have **`add`** and **`deleteMin`** operations, **NO**
  - There is a faster way -- O(n) -- but requires the data structure to have a specialized **`buildHeap`** operation
  - **Is it convenient? Efficient? Simple?**

Floyd's `buildHeap` Method

- Recall our general strategy for working with the heap:
  - Preserve structure property
  - (Break and) Restore heap ordering property

- Floyd's buildHeap:
  - Create a complete tree by putting the n items in an array
    - *Structure property!*
  - Treat the array as a binary heap and fix the heap-order property
    - *Order property!*
  - Exactly how we do this is where we gain efficiency

Reminder: a priority queue contains *priorities* and *values*; an *item* or *data* refers to the (priority, value) pair

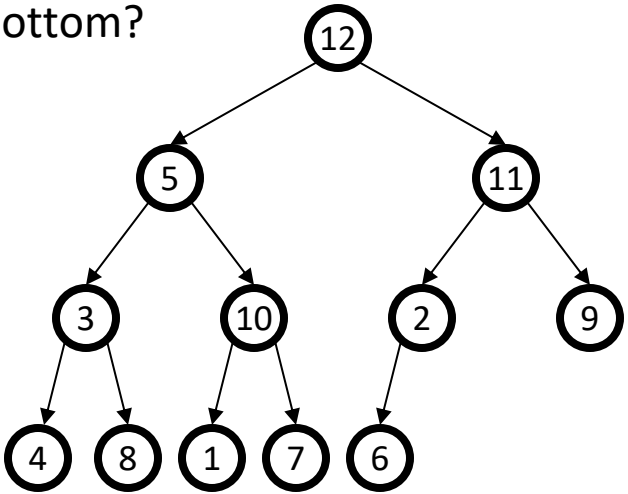
Robert Floyd

- Turing Award winner
  - Floyd-Warshall algorithm (all-pairs shortest path)
  - Programming parsing and semantics
- Invented in-place Heapsort

By Source, Fair use, https://en.wikipedia.org/w/index.php?curid=59539154

Thinking about `buildHeap`

- Say we start with this array: [12,5,11,3,10,2,9,4,8,1,7,6]
- Where should we start? Top vs bottom?
- To "fix" the ordering can we use:
  - percolateUp?
  - percolateDown?



Floyd's `buildHeap` Method

- Bottom-up:
  - Leaves are already in heap order
  - Work *up toward the root* one level at a time, percolating *downwards*

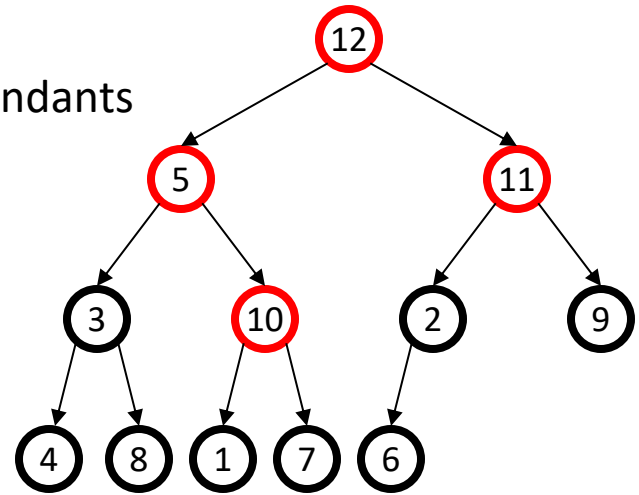
```
void buildHeap(arr) {
   n = arr.length
   for (i = n/2; i>0; i--) {
     val  = arr[i];
     hole = percolateDown(i, val);
     arr[hole] = val;
   }
}
```

Note: P1 doesn't skip; starts counting from 0

`buildHeap` Example

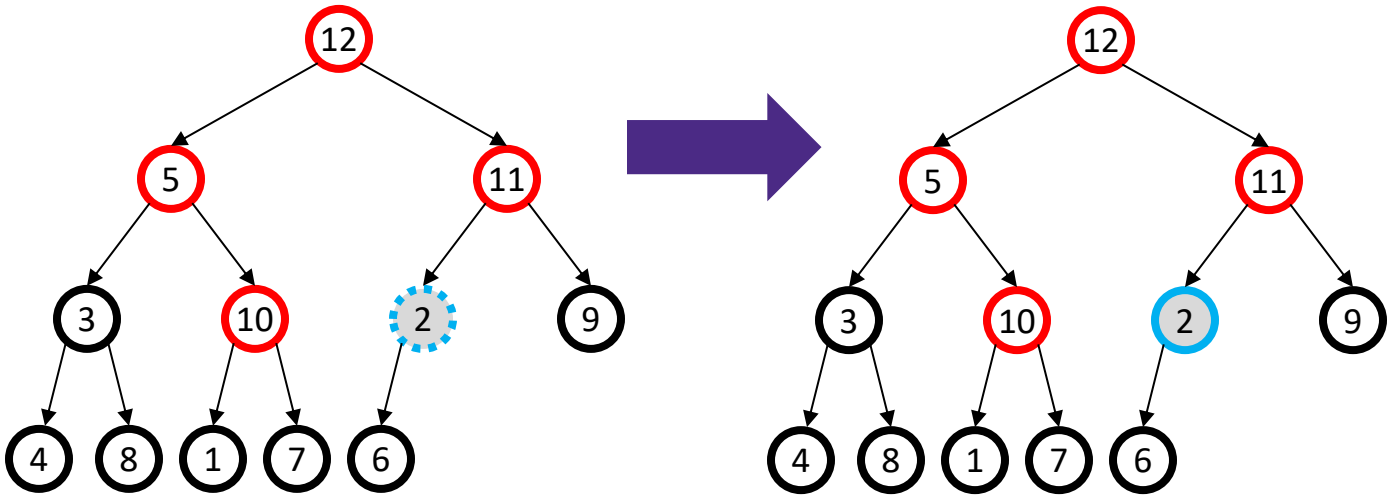
- Say we start with this array: [12,5,11,3,10,2,9,4,8,1,7,6]
  - In tree form for readability

- Red for node not less than descendants
  - Ie, heap-order problem
  - Notice no leaves are red!



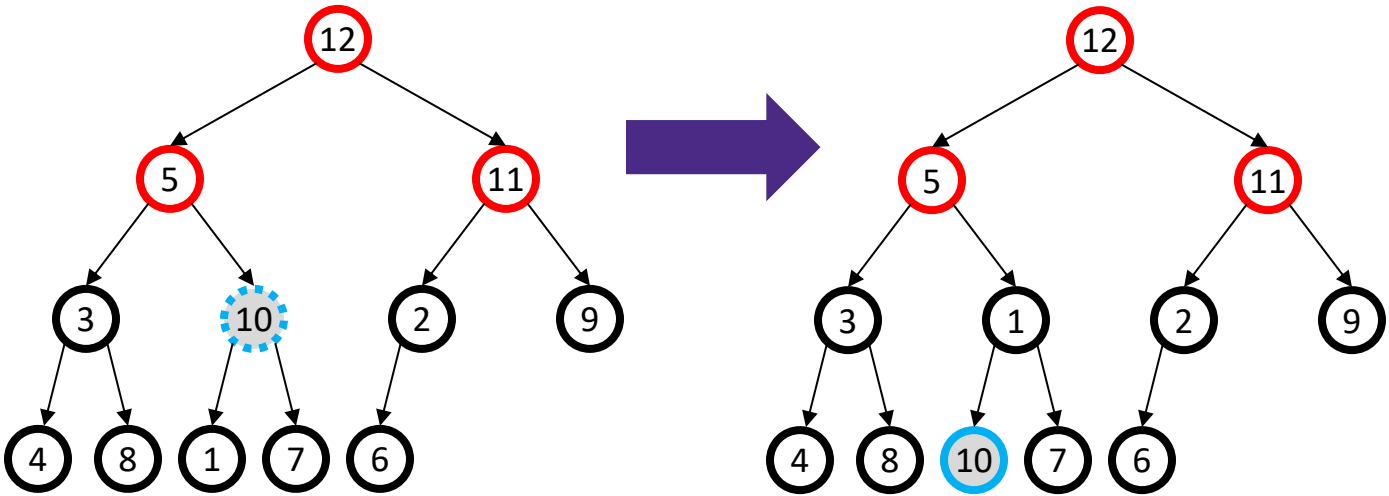
`buildHeap` Example: Step 1

- Happens to already be less than child



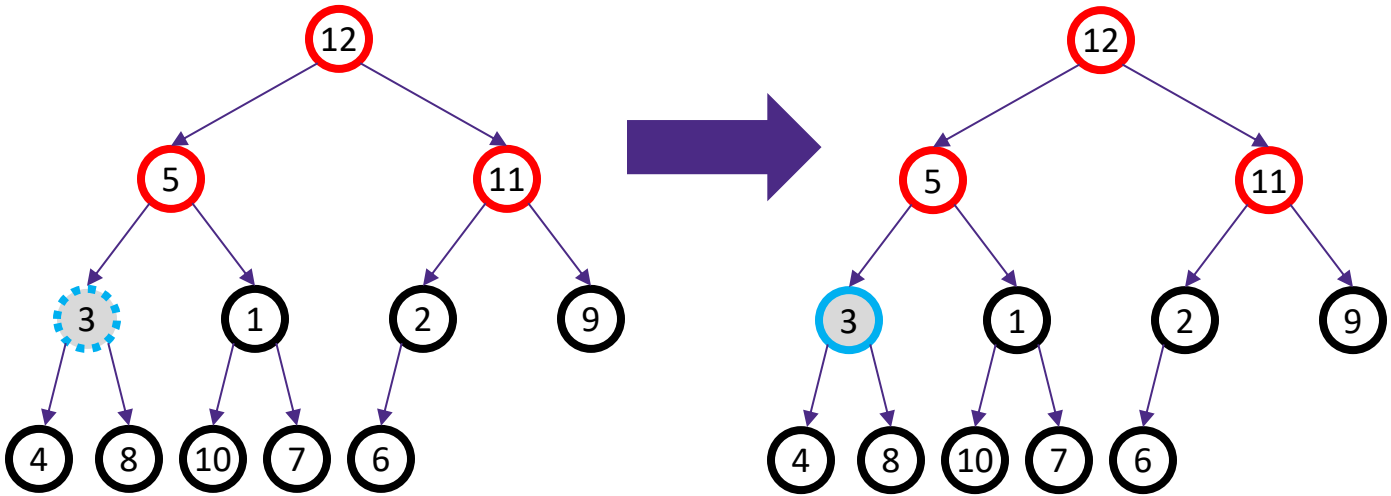
`buildHeap` Example: Step 2

- Percolate down (notice that this moves up '1')



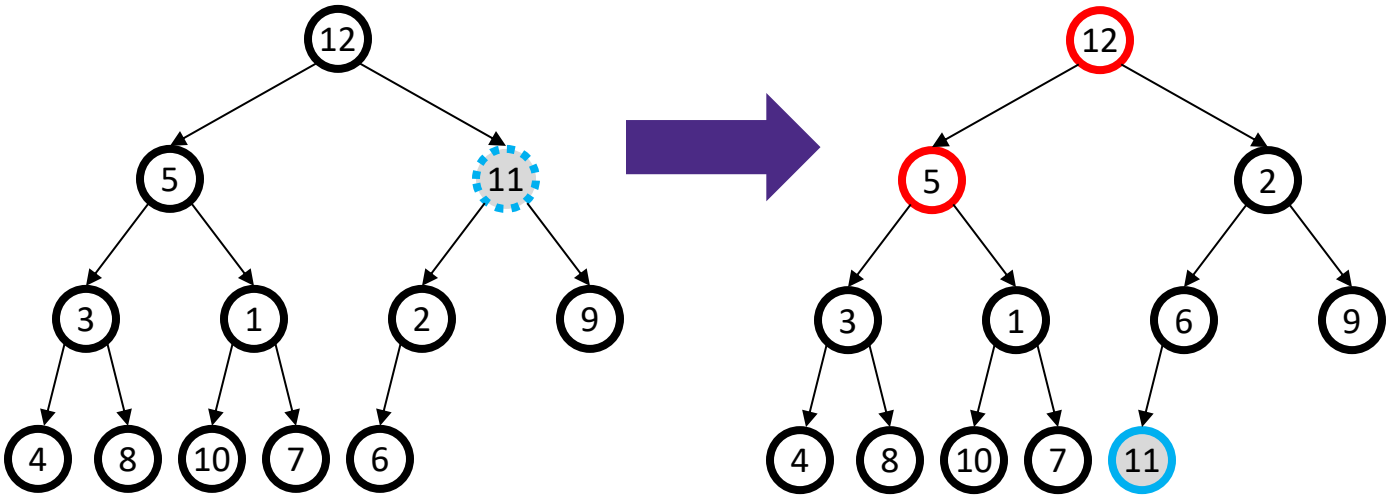
`buildHeap` Example: Step 3

- Another nothing-to-do step



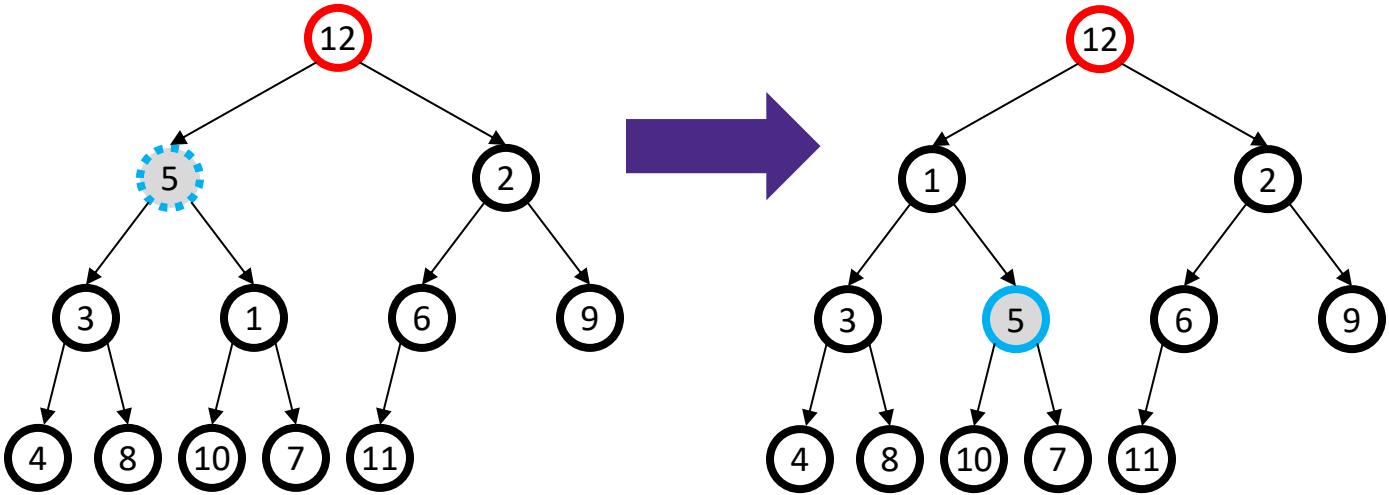
`buildHeap` Example: Step 4

- Percolate down. Which nodes got moved?



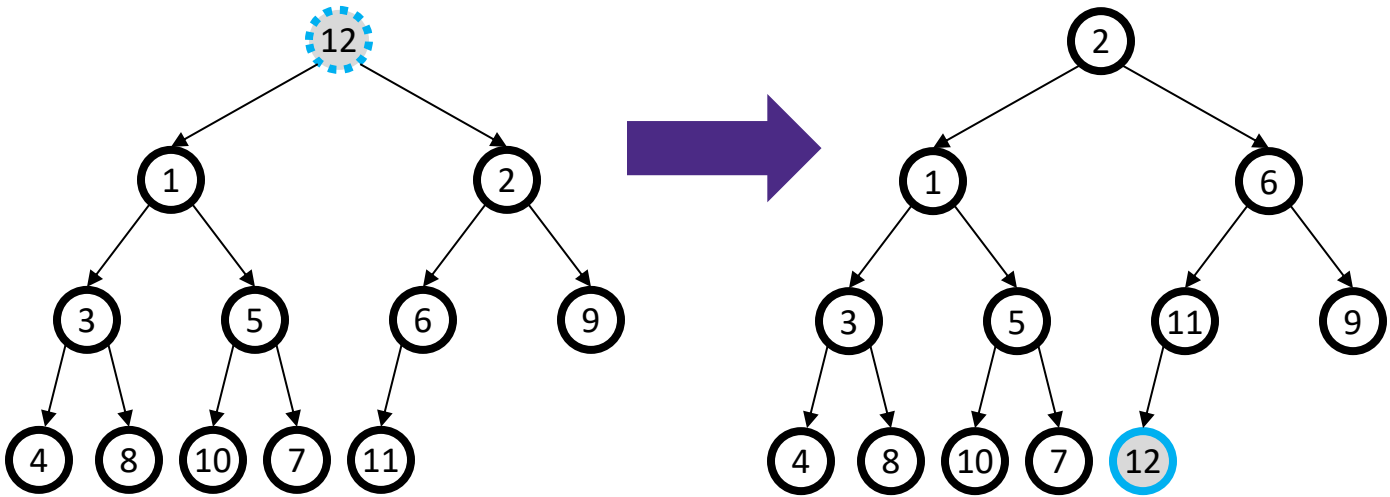
`buildHeap` Example: Step 5

- Again, percolate down



`buildHeap` Example: Step 6

- Lastly, percolate down as necessary



But is it right?

- "Seems to work"
  - Let's *prove* it restores the heap property (correctness)
  - Then let's *prove* its running time (efficiency)

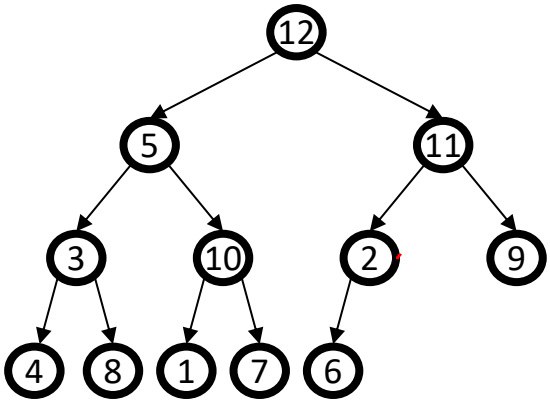
```
void buildHeap(arr) {
   n = arr.length
   for(i = n/2; i>0; i--) {
     val  = arr[i];
     hole = percolateDown(i, val);
     arr[hole] = val;
   }
}
```

Floyd's `buildHeap`: Correctness

- ***Loop Invariant***: For all `j>i`, `arr[j]` is less than its children
  - True initially: If `j > size/2`, then `j` is a leaf
    - Otherwise its left child would be at position `>size`
  - True after one iteration: loop body and `percolateDown` make `arr[i]` less than children without breaking the property for any descendants
- Therefore, after loop terminates, ***all nodes are less than their children***

```
void buildHeap(arr) {
   n = arr.length
   for(i = n/2; i>0; i--) {
     val  = arr[i];
     hole = percolateDown(i, val);
     arr[hole] = val;
   }
}
```

Floyd's `buildHeap`: Correctness Example




```
void buildHeap(arr) {
   n = arr.length
   for(i = n/2; i>0; i--) {
     val  = arr[i];
     hole = percolateDown(i, val);
     arr[hole] = val;
   }
}
```

	12	5	11	3	10	2	9	4	8	1	7	6
0	1	2	3	4	5	6	7	8	9	10	11	12

Floyd's `buildHeap`: Efficiency (1 of 2)

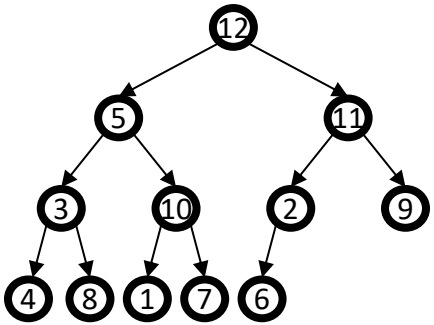
- Easy argument: `buildHeap` is O(n log n) where n is array size
  - n/2 loop iterations
  - Each iteration does one `percolateDown`, which are O(log n) each
  - So Floyd's `buildHeap` is n/2 * log n = O(n log n)

- This is correct, but there is a more precise ("tighter") analysis

```
void buildHeap() {
   for(i = size/2; i>0; i--) {
    val  = arr[i];
     hole = percolateDown(i,val);
    arr[hole] = val;
  }
}
```

Floyd's `buildHeap`: Efficiency (2 of 2)

- Better argument: `buildHeap` is O(n) where n is array size
  - n/2 total loop iterations: O(n)
    - 1/2 of the loop iterations percolate at most **1 step**
    - 1/4 of the loop iterations percolate at most **2 steps**
    - 1/8 of the loop iterations percolate at most **3 steps**
    - … etc …
  - But we know (1 + (**1**/2) + (**2**/4) + (**3**/8) + …) = 2
    - See page 4 of Weiss
    - Also see Weiss 6.3.4, sum of heights of nodes in a perfect tree
  - So Floyd's `buildHeap` is n/2 * 2 = O(n)



Lessons from `buildHeap`

- Without `buildHeap`, our ADT let clients implement their own in θ(n log n) worst case
  - Worst case is inserting lower priorities later

- By providing a specialized operation (with access to the internal data structure), we can do O(n) worst case
  - Intuition: Most items are near a leaf, so better to percolate down

- Can analyze this algorithm for:
  - Correctness: Non-trivial inductive proof using loop invariant
  - Efficiency:
    - First analysis easily proved it was O(n log n)
    - A "tighter" analysis shows same algorithm is O(n)

Heap vs other Priority Queue implementations

	add	deleteMin
Heap	**O(logN), O(1) average / expected case**	**O(logN)**
Unsorted Array	add at end: O(1)	search: O(N)
Unsorted Singly-linked Linked List	add at front: O(1)	search: O(N)
Sorted Circular Array	search + shift: O(N)	move front pointer: O(1)
Sorted Doubly-linked 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