CSE 332: Data Structures & Parallelism
Lecture 4: Priority Queues and Heaps

Arthur Liu

Summer 2022

Announcements

- Checkpoint 1 due last night
- EX02 due Friday

Today – Priority Queues and Heaps

- Priority Queue ADT
- Binary Min-Heap Datastructure

- (More recurrences on Friday)

Scenario

What is the difference between waiting for service at a pharmacy versus an ER?

Pharmacies usually follow the rule First Come, First Served — Queue — FIFO

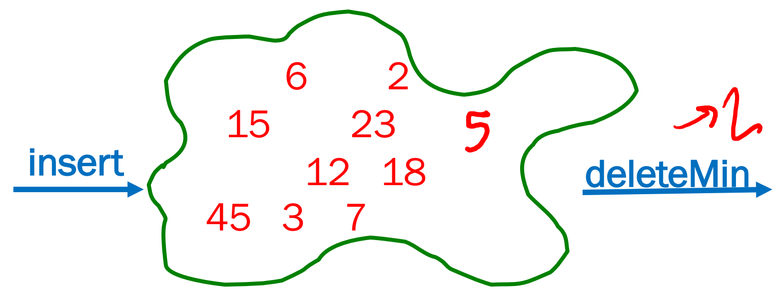
Emergency Rooms assign priorities based on each individual's need — Priority Queue

Priority Queue ADT

Priority Queue ADT

State:
- Set of comparable elements
  - Order based on "priority"

Operations:
- **insert(element)**
- **deleteMin()** – returns the element with the smallest priority, removes it from the collection
- **findMin()**



- Assume each item has a "priority"
  - The *lesser* item is the one with the *greater* priority
  - So "priority 1" is more important than "priority 4"
  - Just a convention, could also do a maximum priority

Aside: We will use ints as data <u>and</u> priority

For simplicity in lecture, we'll often suppose items are just **`int`**s and the **`int`** is also the priority

- So an operation sequence could be

```
insert 6
insert 5
x = deleteMin  // Now x = 5.
```

– **`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 is the data

Applications

Like all good ADTs, the priority queue arises often

- Sometimes "directly", sometimes less obvious

- Run multiple programs in the operating system
  - "critical" before "interactive" before "compute-intensive"
  - Maybe let users set priority level
- Treat hospital patients in order of severity (or triage) ← huffman encoding
- Select print jobs in order of decreasing length?
- Forward network packets in order of urgency
- Select most frequent symbols for data compression (peep CSE143)
- Sort: **`insert`** all, then repeatedly **`deleteMin`** → heap sort

More applications

- “Greedy” algorithms
  - Select the ‘best-looking’ choice at the moment
  - Will see an example when we study graphs in a few weeks
- Discrete event simulation (system modeling, virtual worlds, …)
  - Simulate how state changes when events fire
  - Each event *e* happens at some time t and generates new events *e1*, …, *en* at times *t+t1*, …, *t+tn*
  - Naïve approach: advance “clock” by 1 unit at a time and process any events that happen then
  - Better:
    - *Pending events* in a priority queue (priority = time happens)
    - Repeatedly: **`deleteMin`** and then **`insert`** new events
    - Effectively, “set clock ahead to next event”

Preliminary Implementations of Priority Queue ADT

	insert	deleteMin
Unsorted Array	insert at end O(1)	O(N)
Unsorted Linked-List	O(1)	O(N)
Sorted Circular Array	log(N) shift O(N) → O(N)	O(1)
Sorted Linked-List	O(N)	O(1)
Binary Search Tree (BST)	O(N)	O(N)

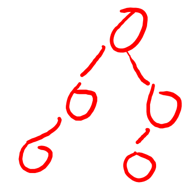
Notes: Worst case, Assume arrays have enough space

Need a good data structure!

- Next we will show an efficient, non-obvious data structure for this ADT
  - But first let's analyze some "obvious" ideas for *n* data items
  - All times worst-case; assume arrays "have room"

data	*insert algorithm / time*		*deleteMin algorithm / time*	
unsorted array	add at end	$O(1)$	search	$O(n)$
unsorted linked list	add at front	$O(1)$	search	$O(n)$
sorted circular array	search / shift	$O(n)$	move front	$O(1)$
sorted linked list	put in right place	$O(n)$	remove at front	$O(1)$
binary search tree	put in right place	$O(n)$	leftmost	$O(n)$

Aside: More on possibilities



- Note: If priorities are inserted in random order, binary search tree will likely do better than *O*(*n*)
  - *O*(**`log`** *n*) **`insert`** and *O*(**`log`** *n*) **`deleteMin`** on average
  - Could get same performance from a *balanced* binary search tree (e.g. AVL tree we will study later)

- One more idea: if priorities are 0, 1, …, *k* can use array of lists
  - **`insert`**: add to front of list at **`arr[priority]`**, *O*(1)
  - **`deleteMin`**: remove from lowest non-empty list *O(k)*

Our Data Structure: The Heap

The Heap:

- Worst case: O(log n) for **insert**
  - If items arrive in random order, then the average-case of insert is O(1) !!
- Worst case: O(log n) for **deleteMin**
- 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

- Do “log”s remind you of anything? 🌲🌲 We will ***visualize*** our heap as a tree

Q: Reviewing Some Tree Terminology

root(**T**):

leaves(**T**): D-F, I, J-N

children(B): D, E, F

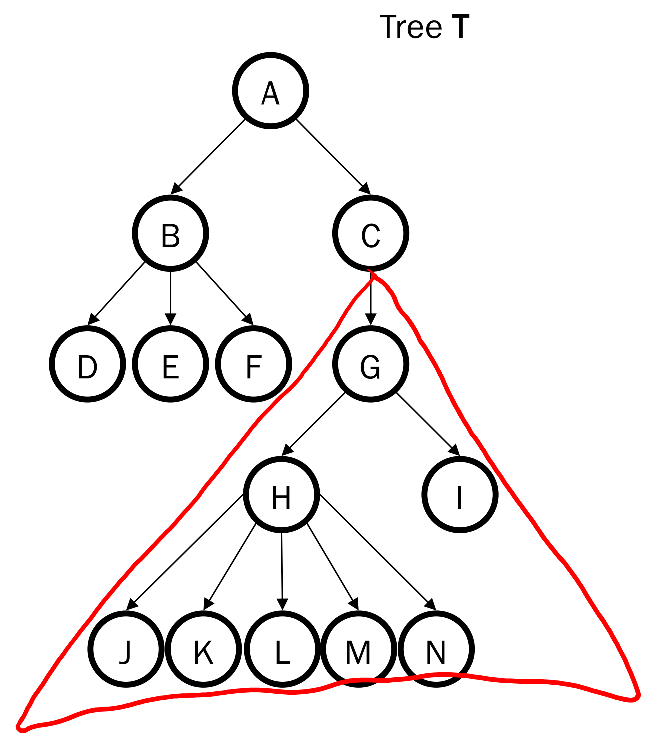
parent(H):

siblings(E):

ancestors(F):

descendents(G):

subtree(G):



A: Reviewing Some Tree Terminology

root(**T**): A

leaves(**T**): D-F, I, J-N

children(B): D, E, F

parent(H): G

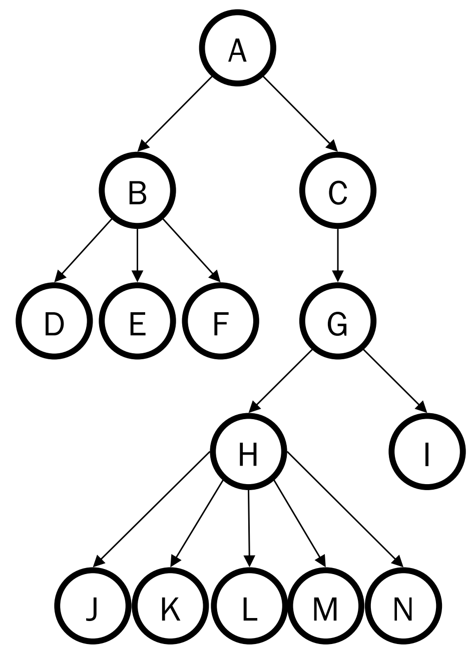
siblings(E): D, F

ancestors(F): B, A

descendents(G): H, I, J-N

subtree(G): G and its children

Tree **T**



Q: Some More Tree Terminology

of edges

depth(B): 1

height(G): 2

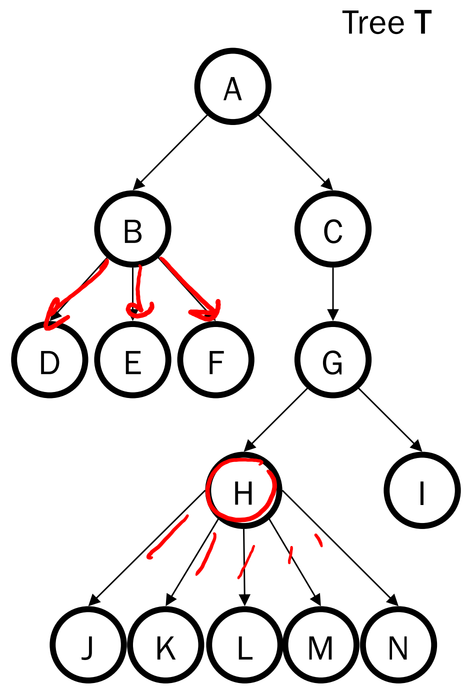
height(**T**): 4

degree(B): 3   degree(H) = 5

branching factor(**T**): 0 – 5

depth(H): 3

height(I): 0



A: Some More Tree Terminology

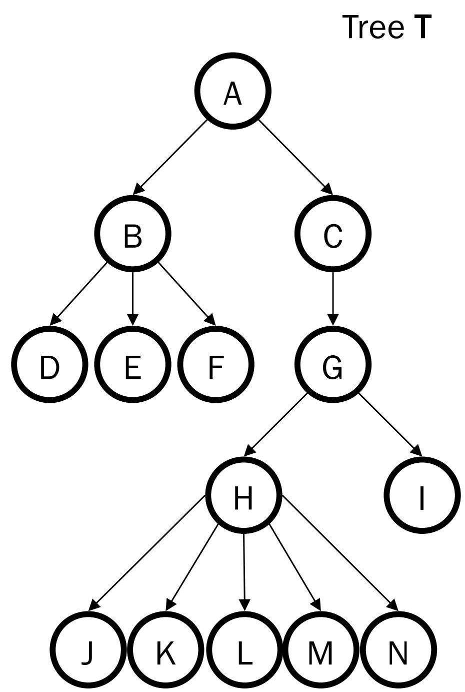
depth(B): *1*

height(G): *2*

height(**T**): *4*

degree(B): *3*

branching factor(**T**): *0-5*



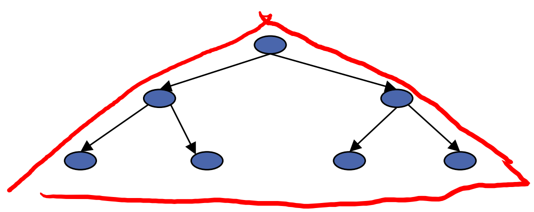
Types of Trees

Binary tree: Every node has ≤2 children

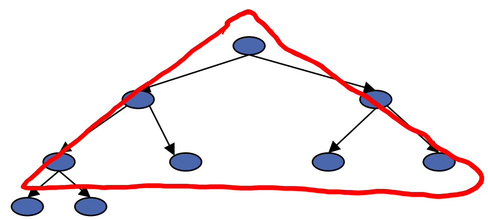
n-ary tree: Every node has ≤n children

Perfect tree: Every row is completely full

Complete tree: All rows except possibly the bottom are completely full, and it is filled from left to right

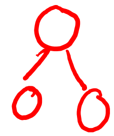


Perfect Tree

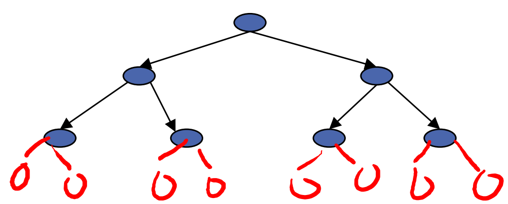


Complete Tree

More on Perfect Trees



Perfect tree: Every row is completely full



Perfect Tree

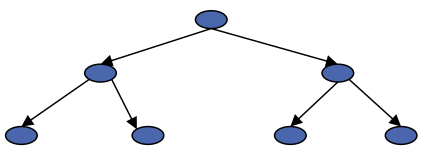
$1 + 2 + 4 + 8 \ldots$

$$\sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$

height	# of nodes	# of leaves
0	1	1
1	3	2
2	7	4
3	15	8
h	$2^{h+1}-1$	2^h

More on Perfect Trees

Perfect tree: Every row is completely full



Perfect Tree

height	# of nodes	# of leaves
0	1	1
1	3	2
2	7	4
3	15	8
h	$2^{h+1} - 1$	2^h

Some Basic Tree Properties

Nodes in a perfect binary tree of height h?

$2^{h+1}-1$

Leaf nodes in a perfect binary tree of height h?

2^h

Height of a perfect binary tree with n nodes?

$\lfloor \log_2 n \rfloor$

Height of a ***complete*** binary tree with n nodes?

$\lfloor \log_2 n \rfloor$

$2^{h+1} - 1 \geq n$

$2^{h+1} \geq n+1$

$\log(2^{h+1}) \geq \log(n+1)$

$h+1 \geq \log(n+1)$

$h \geq \log(n+1) - 1$

$h \geq \text{ceil}\,(\log(n+1)-1)$

Now Formalizing: Binary Min-Heap Datastructure

More commonly known as a binary heap or simply a heap

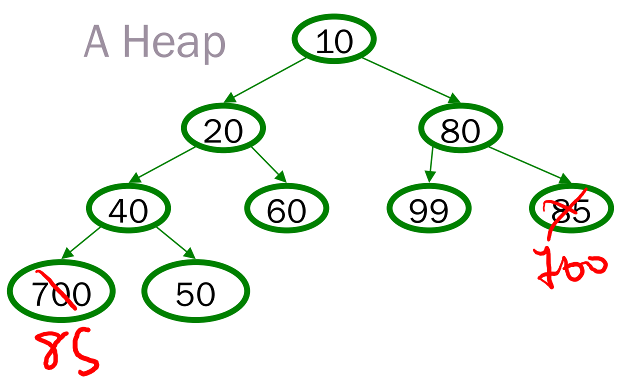
- Structure Property:
  A complete [binary] tree
- Heap-Order Property:
  Every non-root node has a priority value larger than (or possibly equal to) the priority of its parent

Now Formalizing: Binary Min-Heap Datastructure

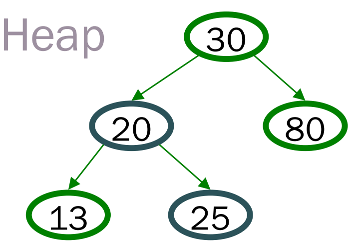
More commonly known as a binary heap or simply a heap

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

A Heap

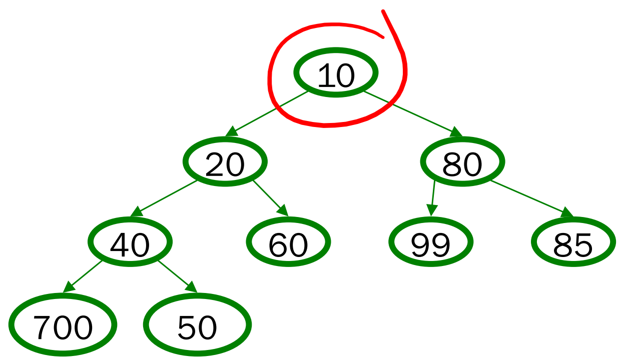


Not a Heap



Properties of Binary Min-Heap

- Where is the minimum priority item? root
- What is the height of a heap with n items? O(log(n))

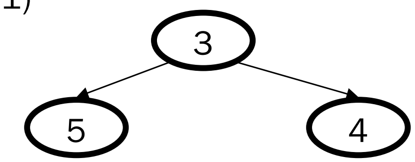


Poll Everywhere

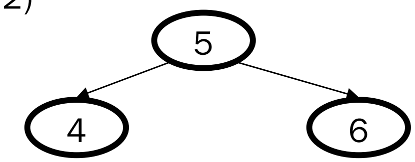
pollev.com/artliu

Are these valid binary heaps?

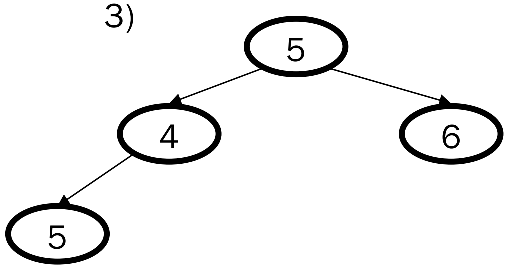
1)



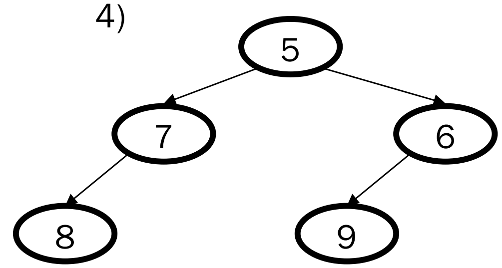
2)



3)



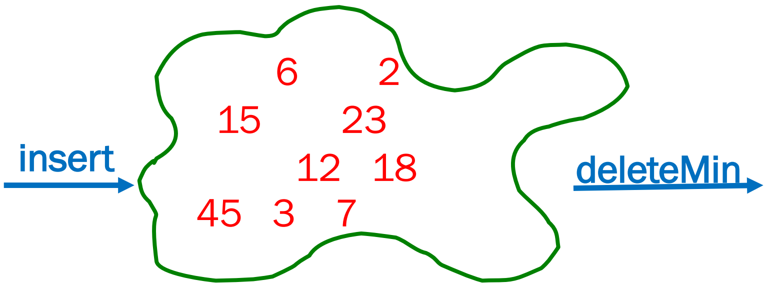
4)



Implementing Priority Queue ADT

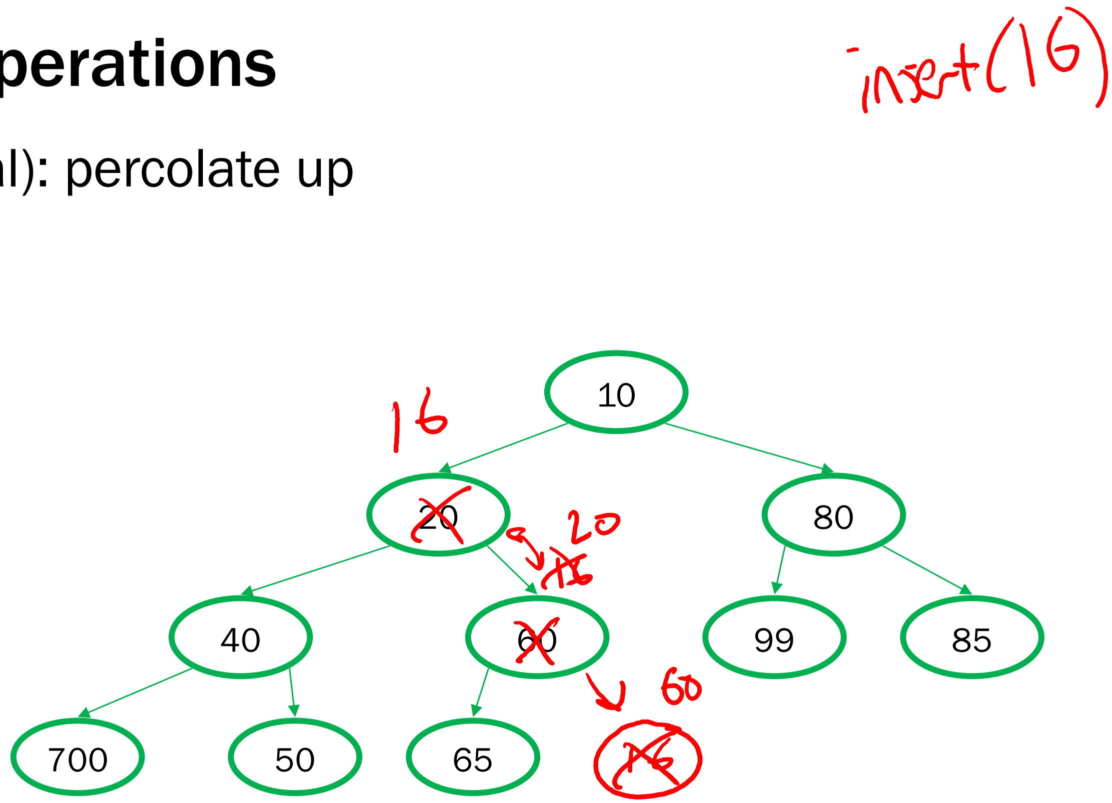
Reminder :)

Priority Queue ADT
State: • Set of comparable elements   • Order based on "priority" Operations: • **insert(element)** • **deleteMin()** – returns the element with the smallest priority, removes it from the collection • **findMin()**



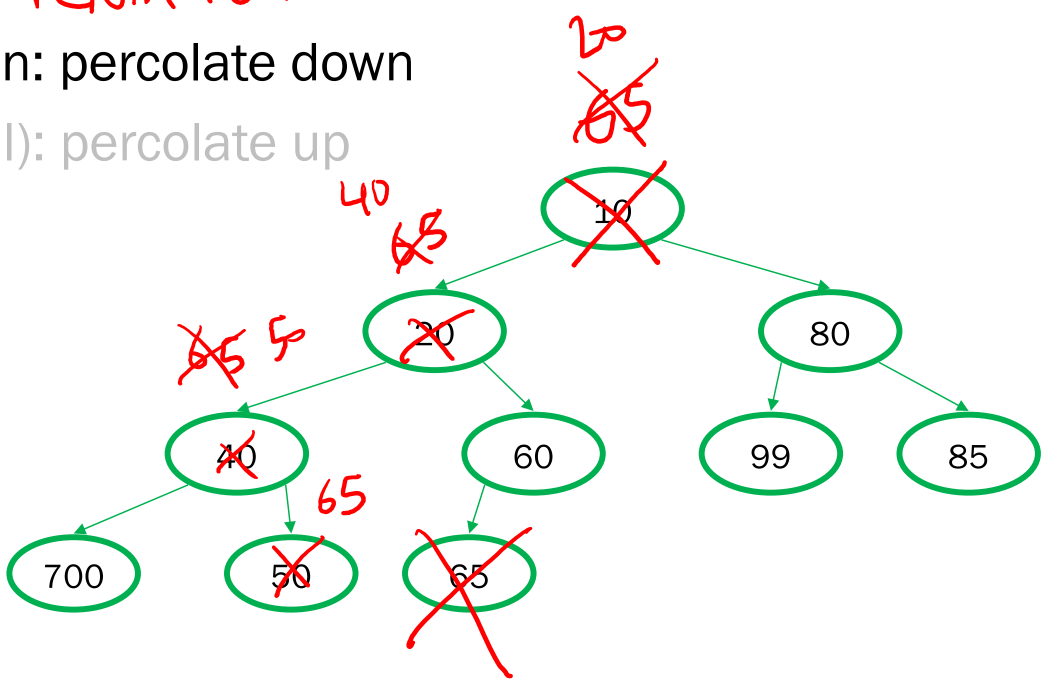
Heap Operations

- insert(val): percolate up



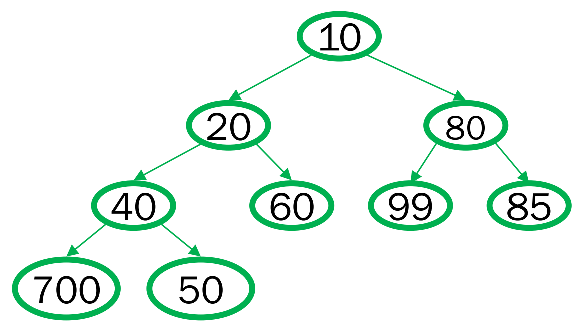
Heap Operations

- findMin: return root
- deleteMin: percolate down
- insert(val): percolate up



Operations: basic idea

- **`findMin`**:
  return **`root.data`**
- **`deleteMin`**:
  1. **`answer = root.data`**
  2. Move right-most node in last row to root to restore structure property
  3. “Percolate down” to restore heap order property
- **`insert`:**
  1. Put new node in next position on bottom row to restore structure property
  2. “Percolate up” to restore heap order property



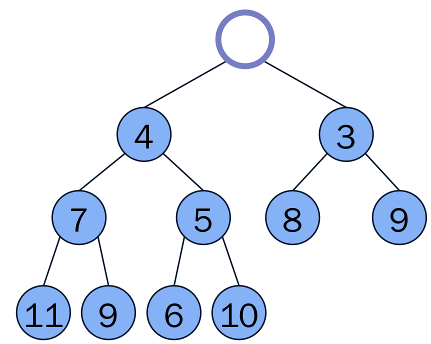
Overall strategy:

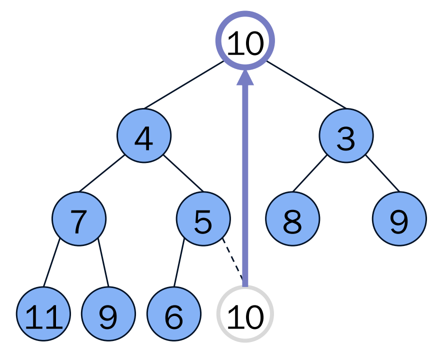
- *Preserve complete tree structure property*
- *This may break heap order property*
- *Percolate to restore heap order property*

DeleteMin Implementation

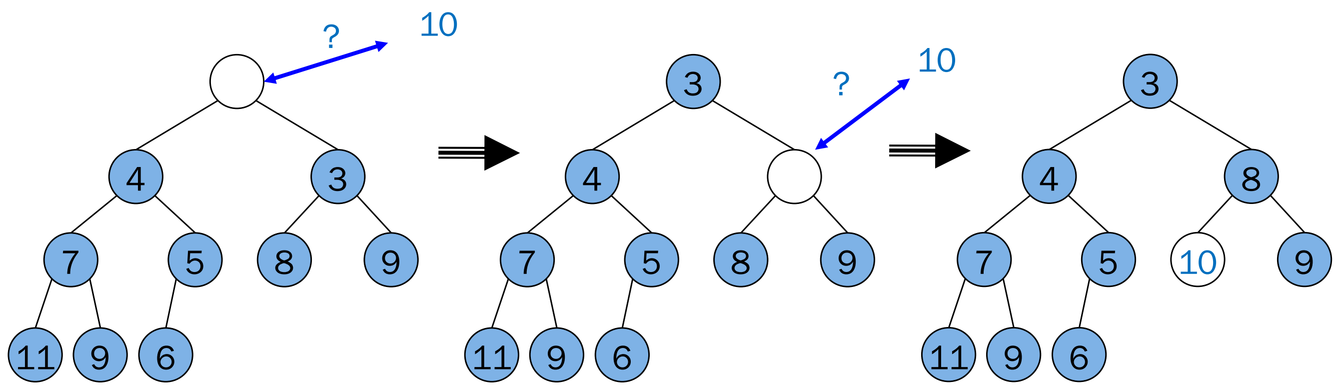
1. Delete value at root node (and store it for later return)
2. There is now a "hole" at the root. We must "fill" the hole with another value, must have a tree with one less node, and it must still be a complete tree
3. The "last" node is the obvious choice, but now the heap order property is violated
4. We percolate down to fix the heap order:

```
While greater than either child
   Swap with smaller child
```





Percolate Down



Percolate down:

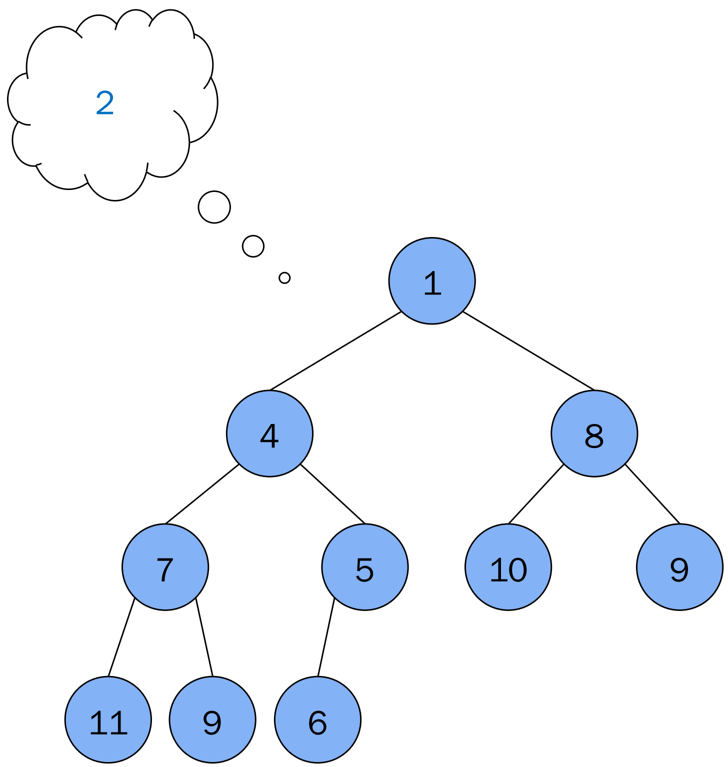
- Keep comparing with both children
- Move smaller child up and go down one level
- Done if both children are ≥ item or reached a leaf node
- Why does this work? What is the run time?

DeleteMin: Run Time Analysis

- Run time is *O*(height of heap)
- A heap is a complete binary tree
- Height of a complete binary tree of *n* nodes?
  height = $\lfloor \texttt{log}_2(n) \rfloor$
- Run time of **`deleteMin`** is $O(\texttt{log}\ n)$

Insert

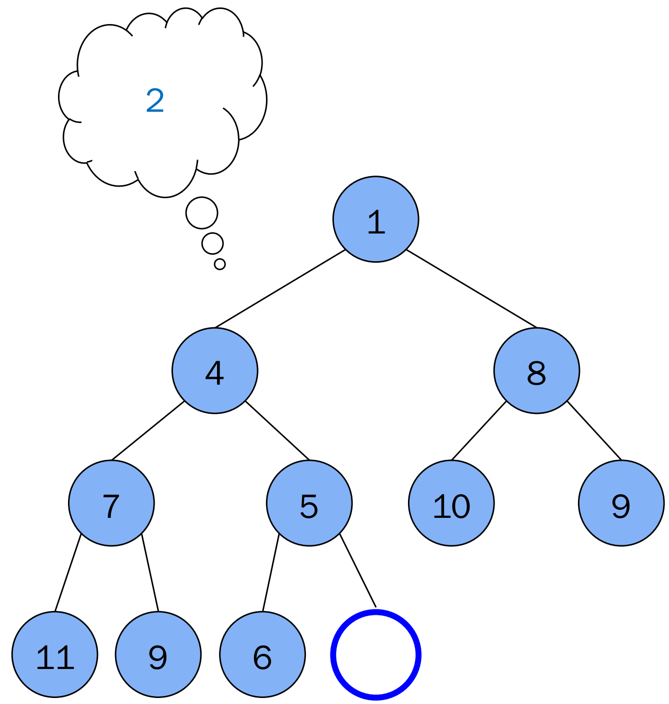
- Add a value to the tree
- Structure and heap order properties must still be correct afterwards



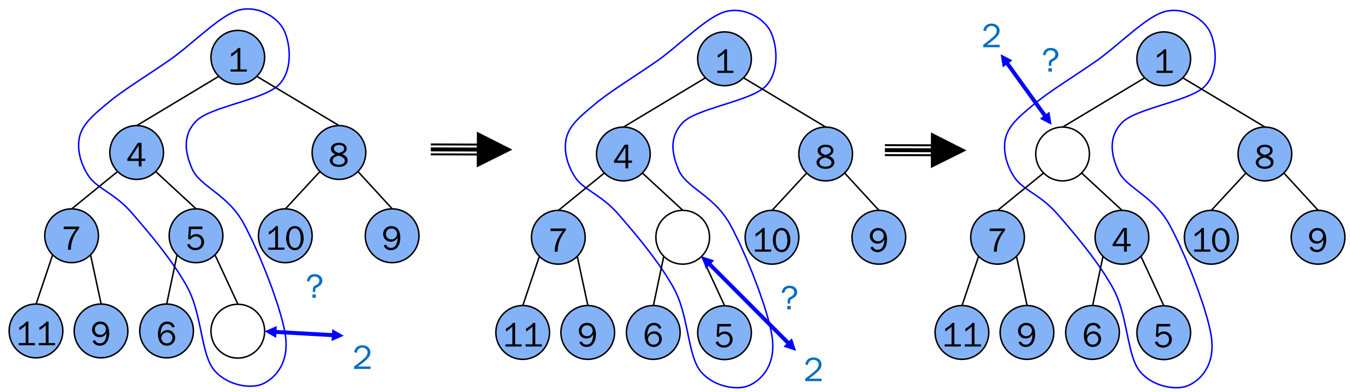
Insert: Maintain the ***Structure*** Property

- There is only **one** valid tree shape after we add one more node!

- So put our new data there and then focus on restoring the heap order property



Insert: Maintain the ***Heap Order*** property



Percolate up:
- Put new data 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?

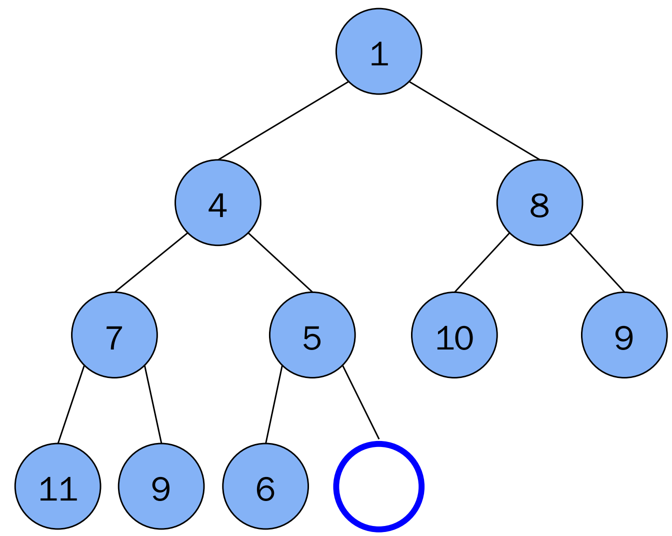
Clever trick for storing the heap…

Need to have access to “next to use” position in the tree. Requires at minimum log(n)…

How could we get O(1) average-case insertion?

Hint: why did we insist the tree be complete?

- All complete trees have the same edges, so we don’t need to explicitly represent edges



Note: Exercises and P2 start counting from 0

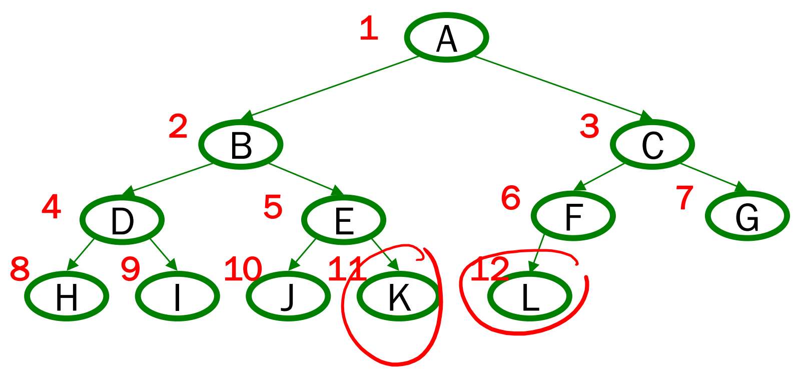
Array Representation of a Binary Heap

From node i:

left child: 2i

right child: 2i+1

parent: floor(i/2)



	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

- We skip index 0 to make the math simpler
- Actually, it can be a good place to store the current size of the heap

Note: Exercises and P2 start counting from 0

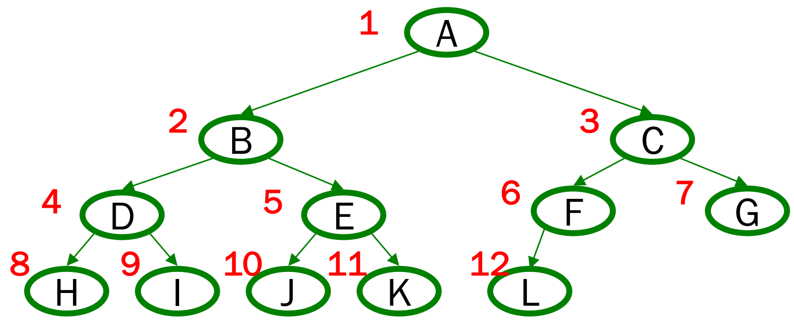
Array Representation of a Binary Heap

From node i:

left child: 2i

right child: 2i+1

parent: floor(i / 2)



	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

- We skip index 0 to make the math simpler
- Actually, it can be a good place to store the current size of the heap

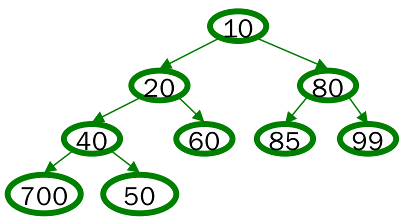
Note: Exercises and P2 start counting from 0

Pseudocode: insert

This pseudocode uses ints. In real use, you will have data nodes with priorities.

```
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

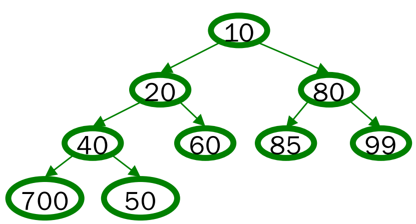
Note: Exercises and P2 start counting from 0

Pseudocode: deleteMin

This pseudocode uses ints. In real use, you will have data nodes with priorities.

```
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