CSE 373: Data Structures and Algorithms

Lecture 12: Binary Heaps

Instructor: Lilian de Greef
Quarter: Summer 2017

Today

- Announcements
- Binary Heaps
  - `insert`
  - `delete`
  - Array representation of tree
  - Floyd's Method of `buildTree`
  - d-heaps

Announcements

- Midterm on Friday
  - Practice midterms on course website
  - Note that some may cover slightly different material
  - Will start at 10:50, will end promptly at 11:50 (even if you’re late), so be early

- Will have homework 3 grades back before midterm

- Reminder: course feedback session on Wednesday

Priority Queue ADT

Like a Queue, but with priorities for each element.

Priority Queue ADT

Meaning:

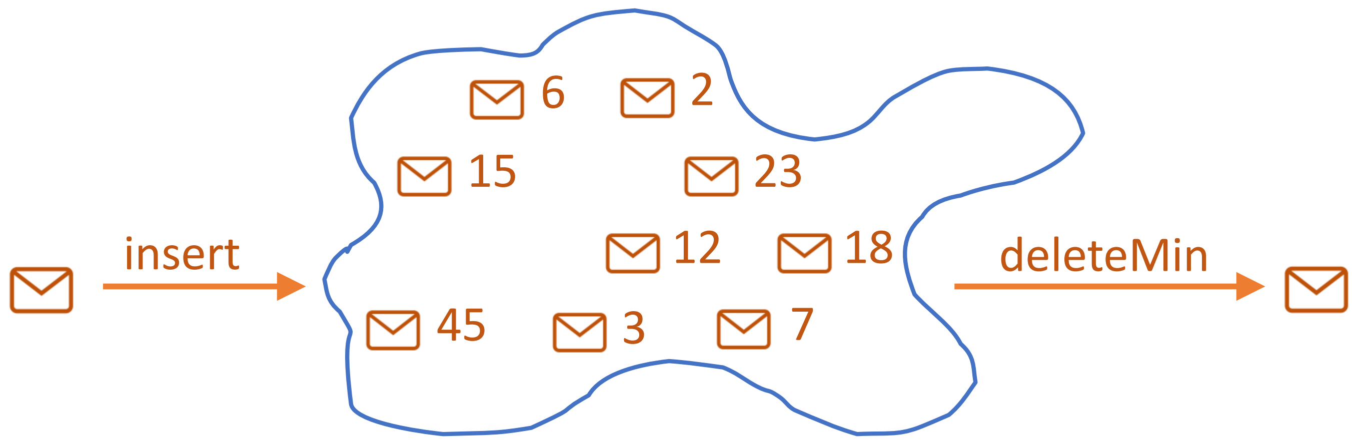
- A **priority queue** holds *compare-able data*
- Key property:

  `deleteMin` returns and deletes the item with the highest priority

  (can resolve ties arbitrarily)

Operations:

- `deleteMin`
- `insert`
- `isEmpty`



Finding a good data structure

Will show an efficient, non-obvious data structure for this ADT

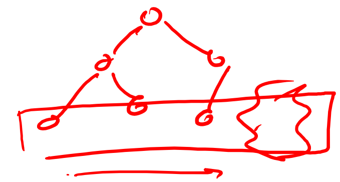
But first let's analyze some "obvious" ideas for *n* data items

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 $O(n)$
AVL tree	put in right place $O(\log n)$	leftmost $O(\log n)$

Binary Heaps

Data Structure for Priority Queue

Our data structure binary heap

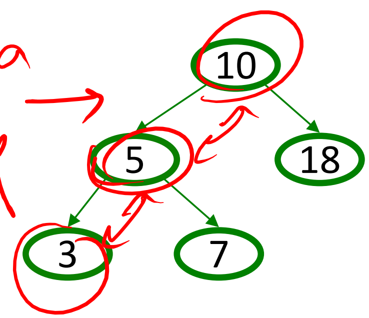


A *binary min-heap* (or just *binary heap* or just *heap*) has:

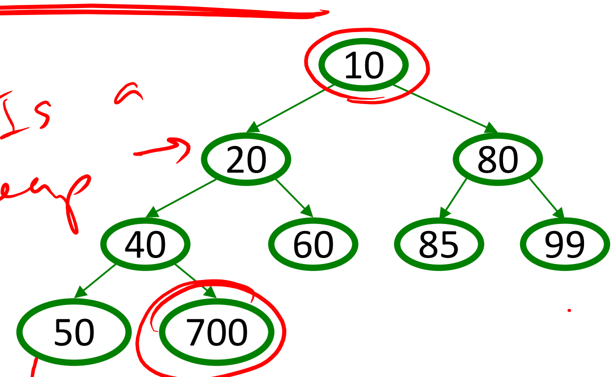
- Structure property: a complete binary tree
- Heap property: The priority of every (non-root) node is less important than the priority of its parent

NOT A BST

Not a heap



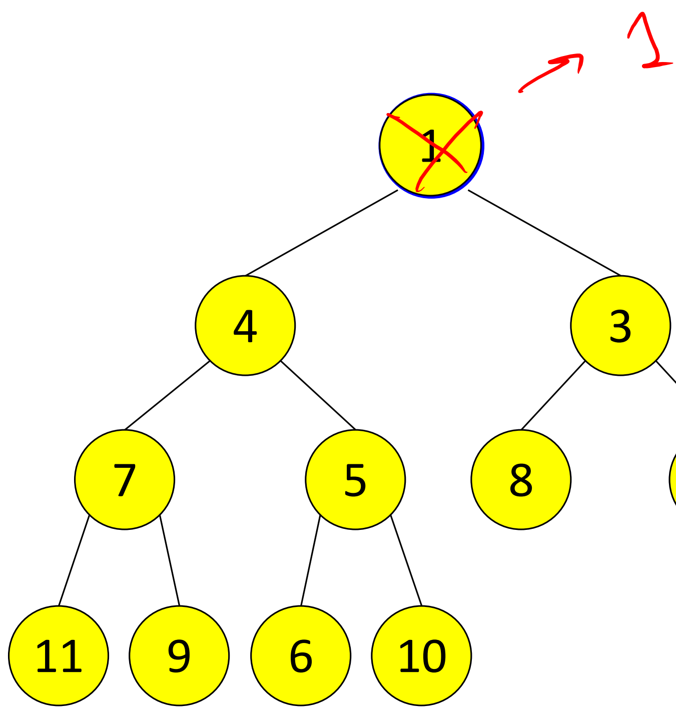
Is a heap



So:

- Where is the highest-priority item? the root!
- Where is the lowest priority? somewhere in the leaves
- What is the height of a heap with *n* items? log n (complete tree)

`deleteMin`: Step #1

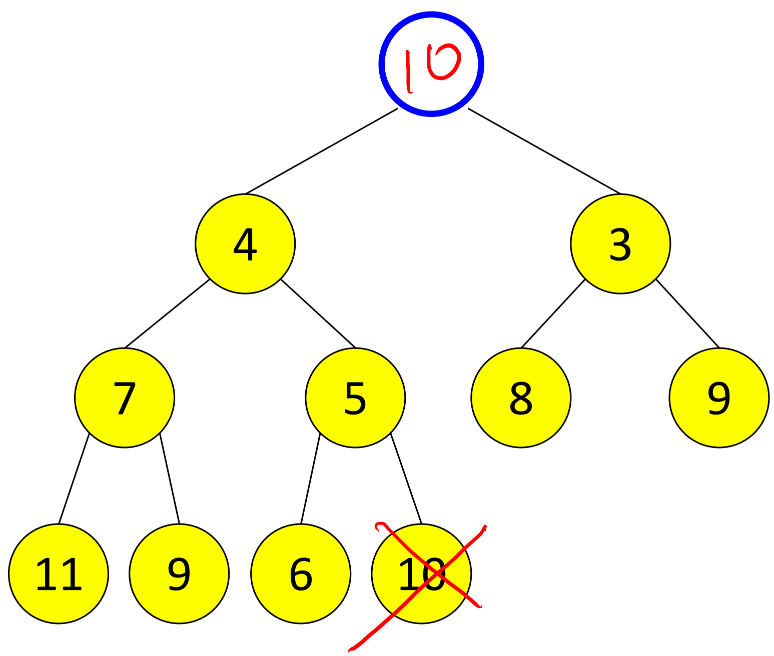


1. Delete (and later return) the value at <u>root</u>

Now we have a "hole" at the root

↳ replace it with another node

`deleteMin`: Step #2 (Keep Structure Property)

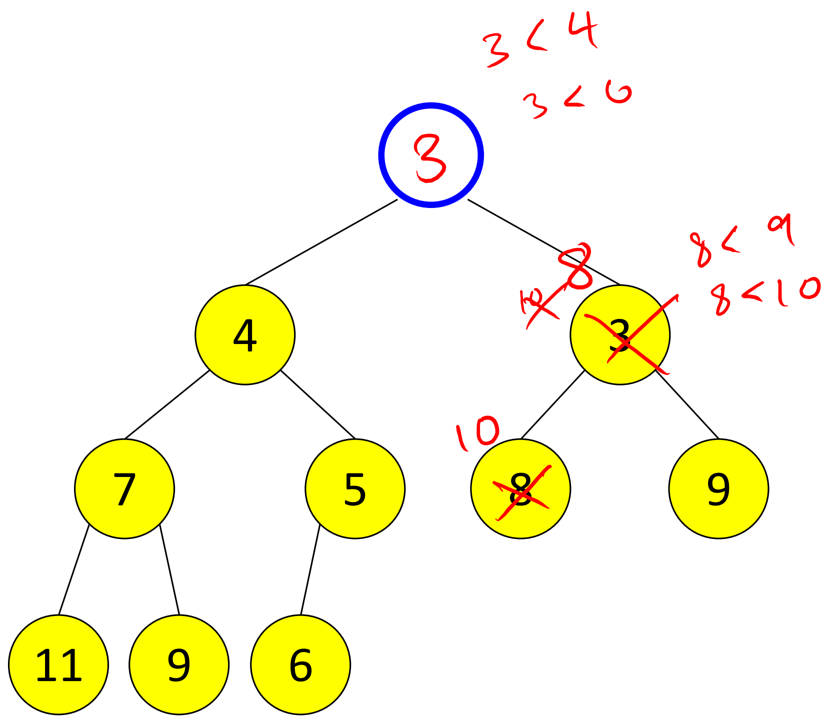


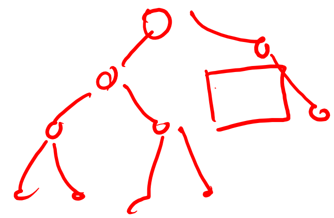
(a complete binary tree)

Want to keep structure property

2. Pick last node on the bottom row and "move" it to the "hole"

`deleteMin`: Step #3





Want to restore heap property

3. "Percolate down"

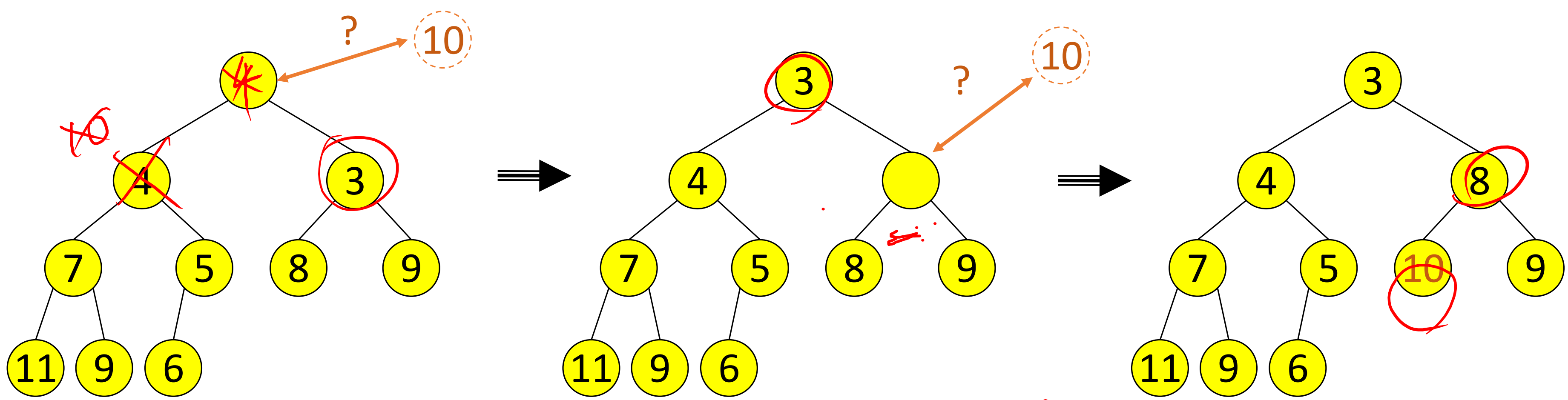
If lower priority than child, swap the most important child

repeat

`deleteMin`: Step #3 (Restore Heap Property)

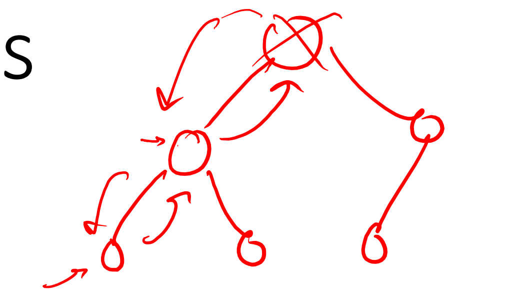
Percolate down:

- Compare priority of item with its children
- If item has lower priority, swap with the most important child
- Repeat until both children have lower priority or we've reached a leaf node

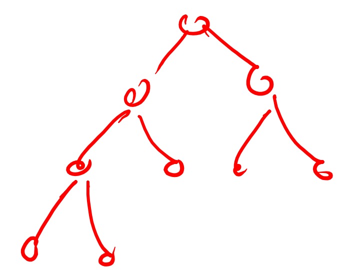


What is the run time? $O(\log n) = O(\text{height of heap})$

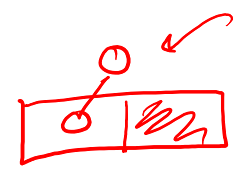
`deleteMin`: Run Time Analysis



- Run time is O(height heap)
- A heap is a complete tree



- So its height with *n* nodes is log n
- So run time of `deleteMin` is O(log n)



`insert`: Step #1

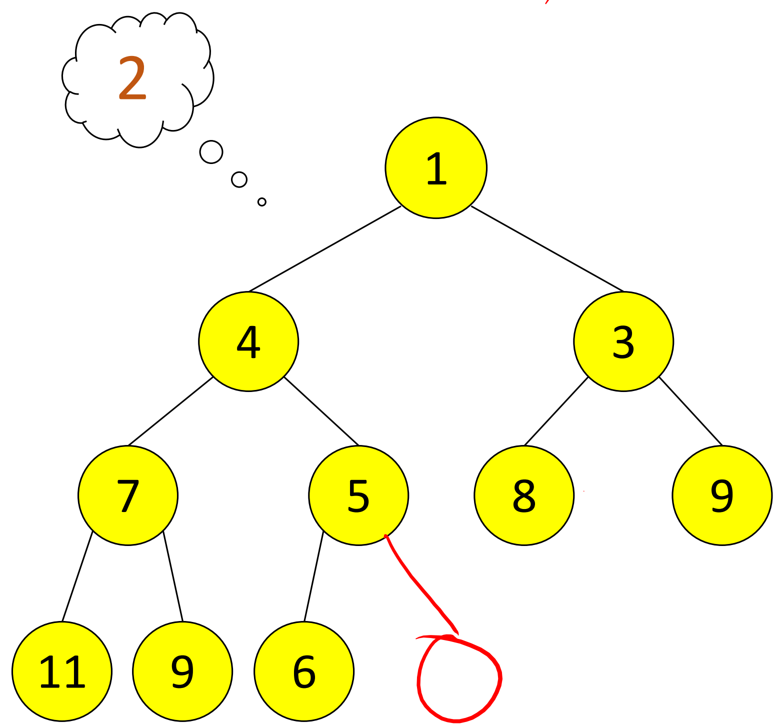
Maintain Structure Property

Put data into next available spot

(end of last row)

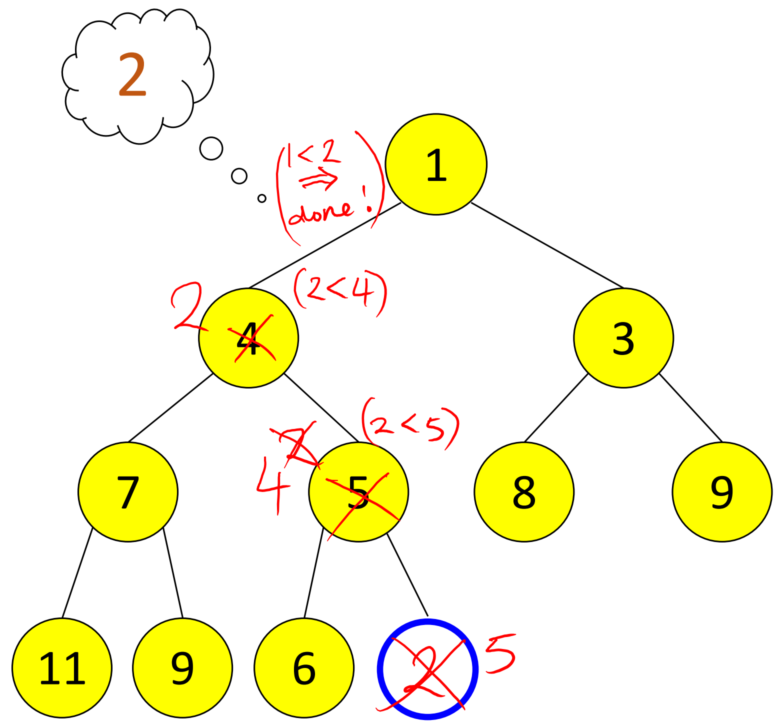
to maintain structure

(complete binary tree).



`insert`: Step #2 Restore Heap Property

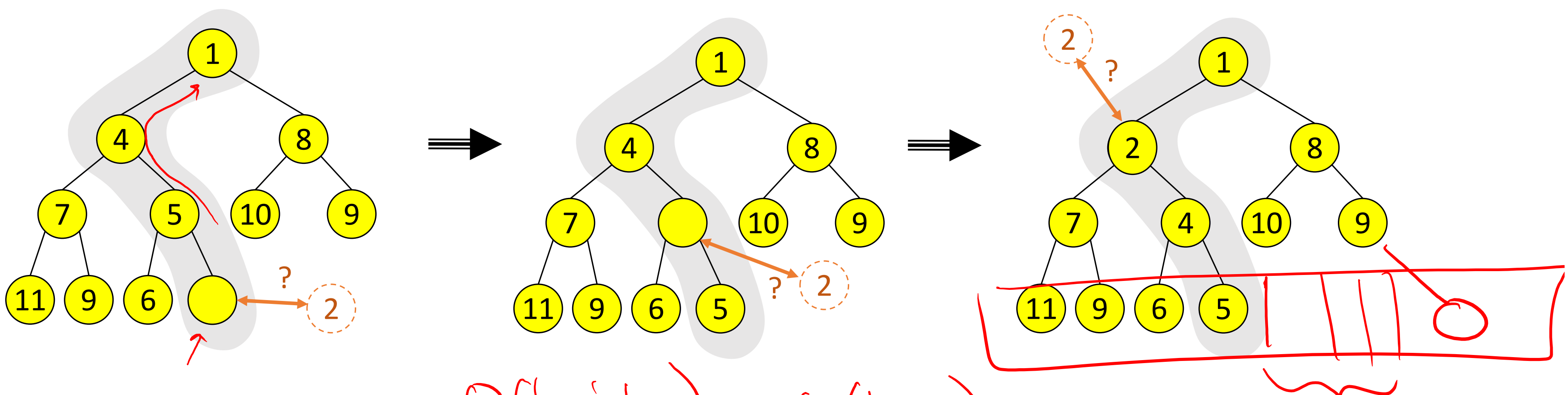
- compare data with its parent
- swap with parent if it's more important than parent
- repeat



`insert`: Step #2 (Restore Heap Property)

Percolate up:

- Put new data in new location ← (maintain structure property)
- If higher priority than parent, swap with parent
- Repeat until parent is more important or reached root



What is the running time? $O(\text{height}) = O(\log n)$

Summary: basic idea for operations

`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 property

`insert:`

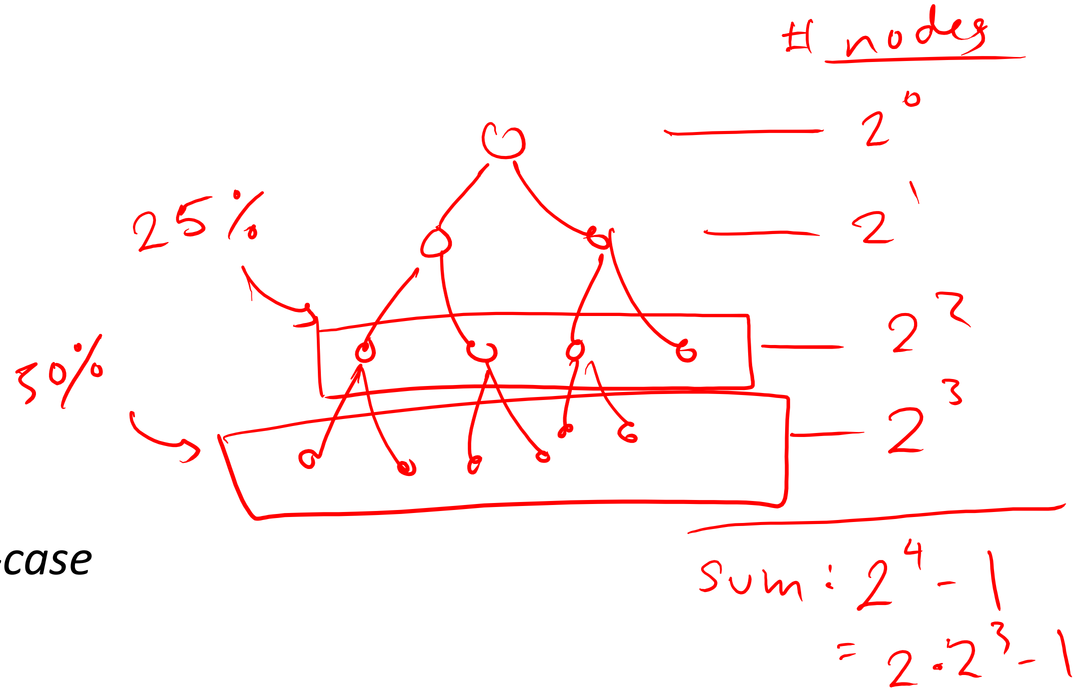
1. Put new node in next position on bottom row to restore structure property
2. "Percolate up" to restore heap property

Overall strategy:

1. *Preserve structure property*
2. *Restore heap property*

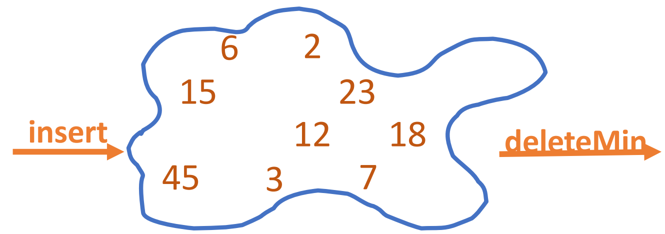
Binary Heap

- Operations
  - $O(\log n)$ `insert`
  - $O(\log n)$ `deleteMin` *worst-case*
  - *Very* good constant factors
  - *If* items arrive in random order, then `insert` is $O(1)$ on *average*

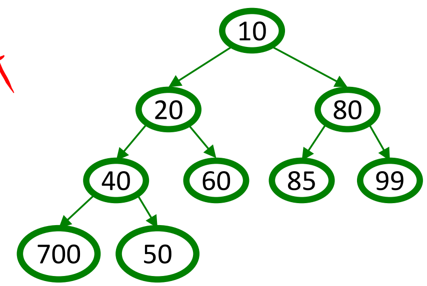


75% of values are in the bottom 2 rows

Summary: Priority Queue ADT



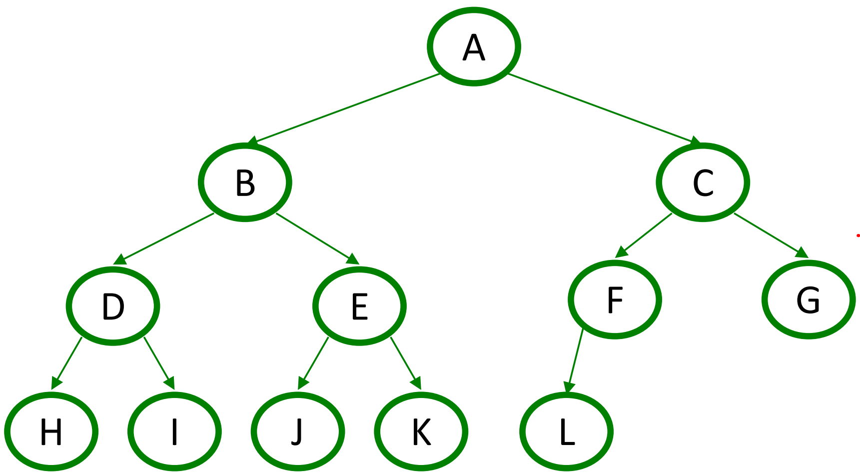
- Priority Queue ADT:
  - `insert` comparable object,
  - `deleteMin`



- Binary heap data structure:
  - Complete binary tree ← not BST
  - Each node has less important priority value than its parent

- `insert` and `deleteMin` operations = *O*(height-of-tree)=*O*(`log` *n*)
  - `insert`: put at new last position in tree and percolate-up
  - `deleteMin`: remove root, put last element at root and percolate-down

Binary Trees Implemented with an **Array**



From node `i`:

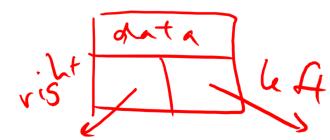
left child: `i*2`
right child: `i*2+1`
parent: `i/2`

(wasting index 0 is convenient for the index arithmetic)

	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

Judging the array implementation

e.g. Tree Node has



Pros:

- Non-data space: just index 0 and unused space on right
  - In conventional tree representation, one edge per node (except for root), so *n*-1 wasted space (like linked lists)

    Array cell only has data
  - Array would waste more space if tree were not complete ← (Con)
- Multiplying and dividing by 2 is very fast (shift operations in hardware)
- Last used position is just index

Cons:

- Same might-be-empty or might-get-full problems we saw with array-based stacks and queues (resize by doubling as necessary)

Pros outweigh cons: min-heaps almost always use array implementation

Practice time!

Starting with an empty array-based binary heap, which is the result after

1. `insert` (in this order): 16, 32, 4, 67, 105, 43, 2
2. `deleteMin` once

0	1	2	3	4	5	6	7

A)

	4	16	32	43	67	105	
0	1	2	3	4	5	6	7

B)

	16	32	4	67	105	43	
0	1	2	3	4	5	6	7

C)

	4	32	16	43	105	67	
0	1	2	3	4	5	6	7

D)

	4	32	16	67	105	43	
0	1	2	3	4	5	6	7

Semi-Pseudocode: `insert` into binary heap

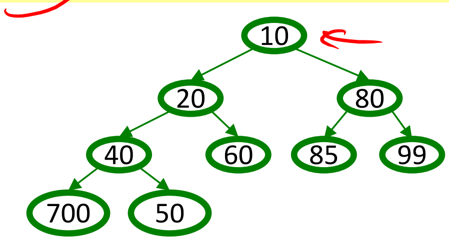
```
void insert(int val) {
  if(size==arr.length-1)
    resize();
  size++;
  i=percolateUp(size,val);
  arr[i] = val;
}
```

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

is root

parent w/ lower priority

index

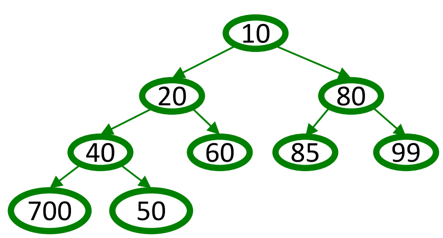


This pseudocode uses ints. In real use, you will have data nodes with priorities.

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

Semi-Pseudocode: `deleteMin` from binary heap

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

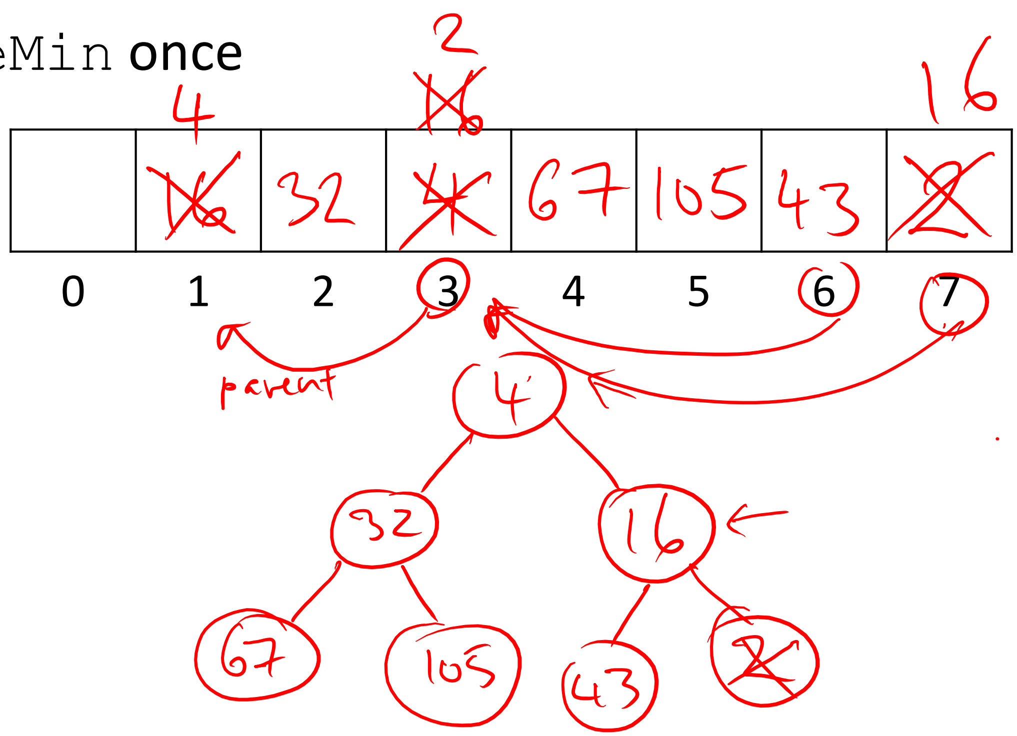



```
int percolateDown(int hole,
                  int val) {
 while(2*hole <= size) {      not leaf
  left  = 2*hole;
  right = left + 1;           children
  if(right > size ||
     arr[left] < arr[right])  check priority
    target = left;
  else
    target = right;
  if(arr[target] < val) {
    arr[hole] = arr[target];  swap
    hole = target;
  } else
      break;                  if done
 }
 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

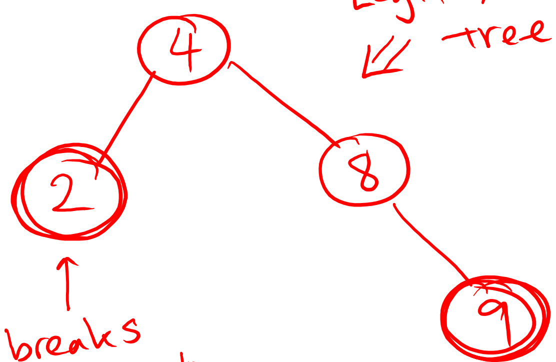
Example

1. `insert` (in this order): 16, 32, 4, 67, 105, 43, 2 ← percolateUp
2. `deleteMin` once



Why AVL Trees can<u>not</u> implement binary heaps via counter example:

Legit AVL tree



breaks Heap Property

breaks structure property