CSE 373: Data Structures and Algorithms

Lecture 12: Binary Heaps

Instructor: Lilian de Greef
Quarter: Summer 2017

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

Meaning:

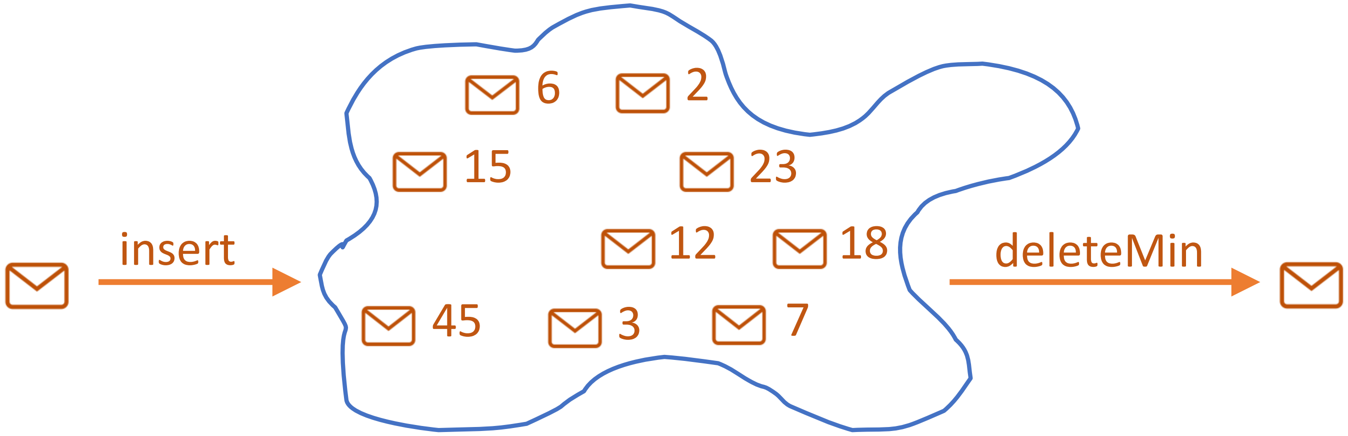
- A **priority queue** holds *compare-able data*
- Key property:

  `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

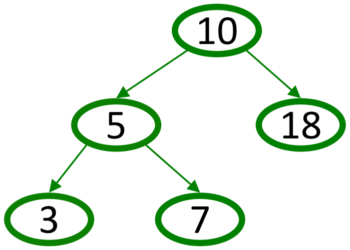
But first let's analyze some "obvious" ideas for *n* data items

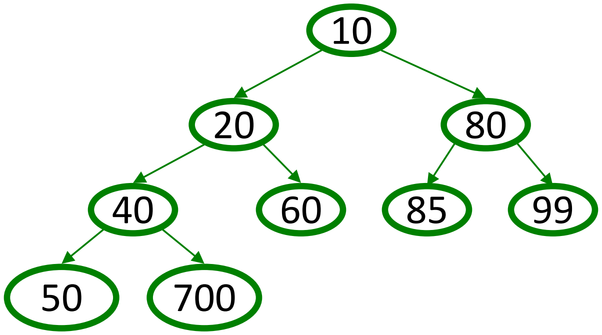
data	*insert algorithm / time*	*deleteMin algorithm / time*
unsorted array	add at end	search
unsorted linked list	add at front	search
sorted circular array	search / shift	move front
sorted linked list	put in right place	remove at front
binary search tree	put in right place	leftmost
AVL tree	put in right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

Our data structure

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

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

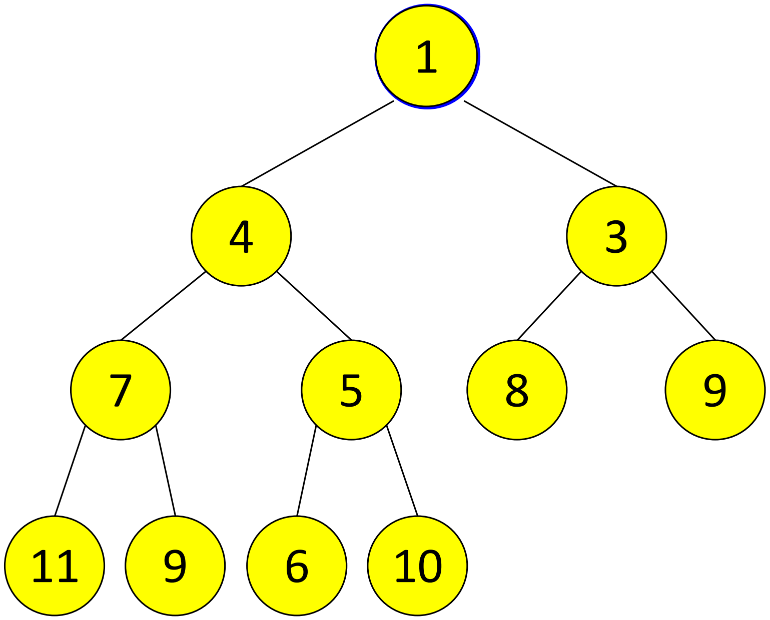




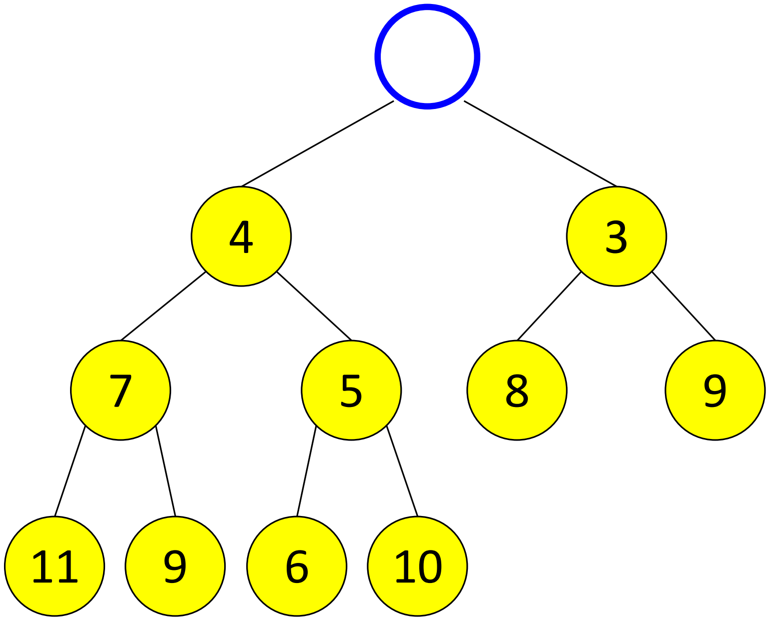
So:

- Where is the highest-priority item?
- Where is the lowest priority?
- What is the height of a heap with *n* items?

`deleteMin`: Step #1

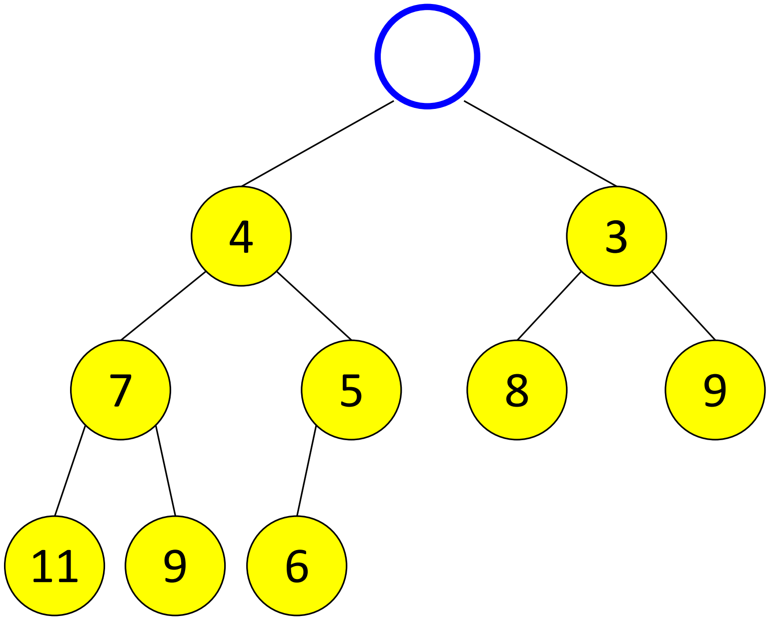


`deleteMin`: Step #2 (Keep Structure Property)



Want to keep structure property

`deleteMin`: Step #3

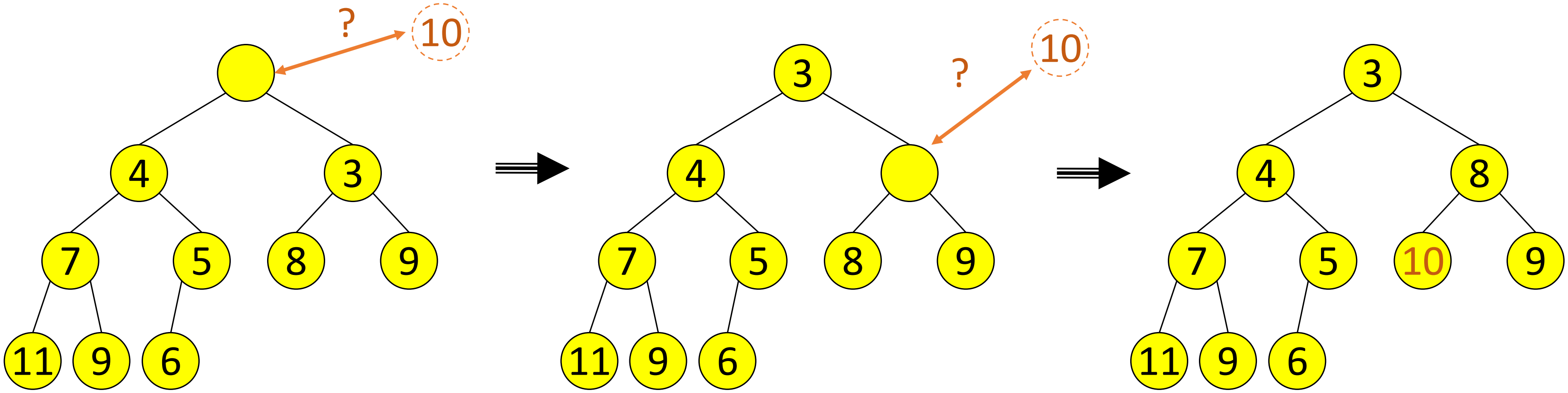


Want to restore heap property

`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?

`deleteMin`: Run Time Analysis

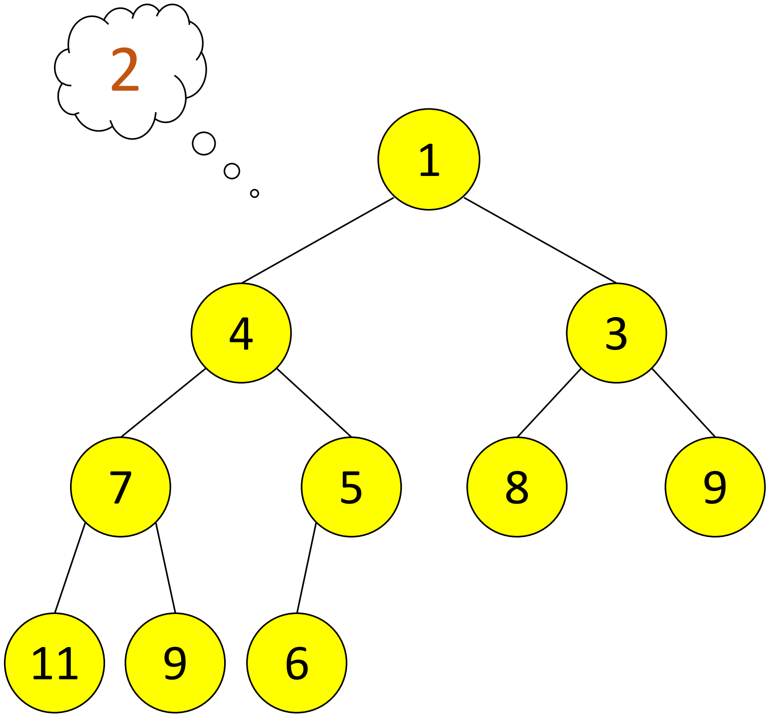
- Run time is

- A heap is a

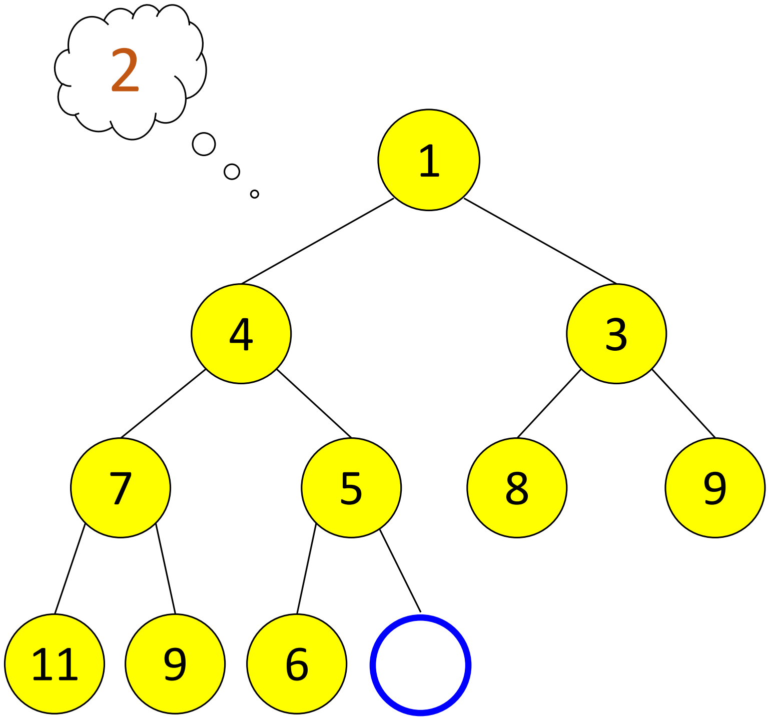
- So its height with *n* nodes is

- So run time of `deleteMin` is

`insert`: Step #1



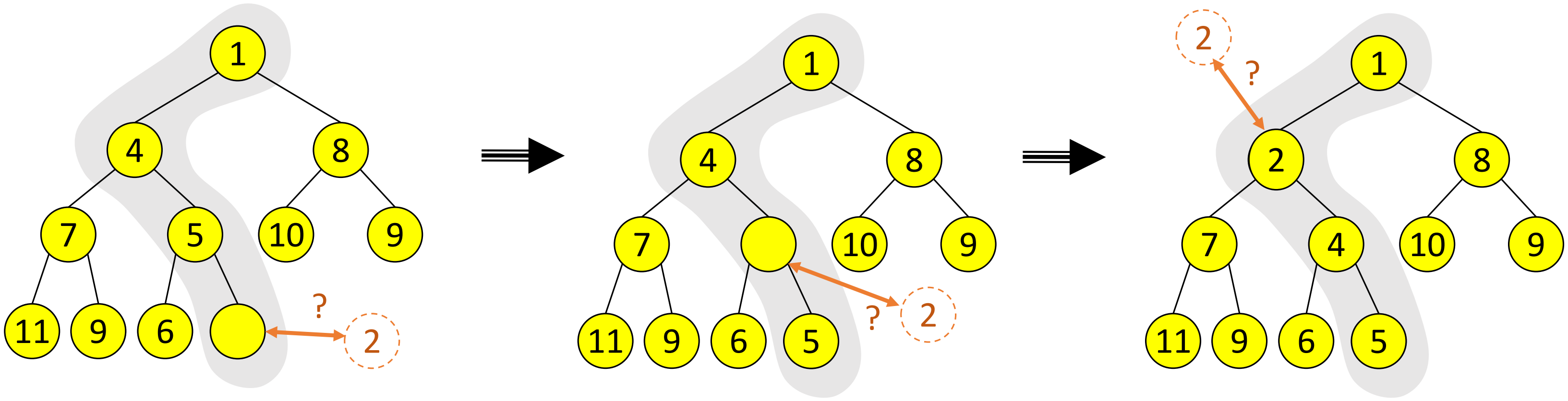
`insert`: Step #2



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

Percolate up:

- Put new data in new location
- If higher priority than parent, swap with parent
- Repeat until parent is more important or reached root



What is the running time?

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

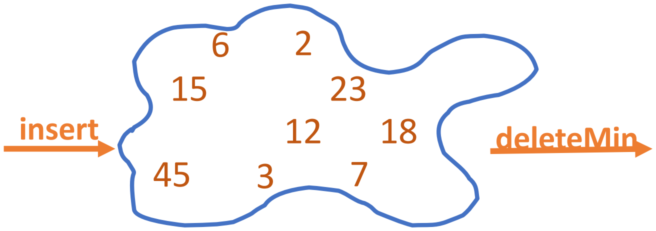
Overall strategy:

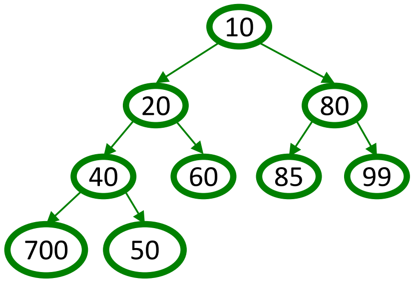
1. *Preserve structure property*
2. *Restore heap property*

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*

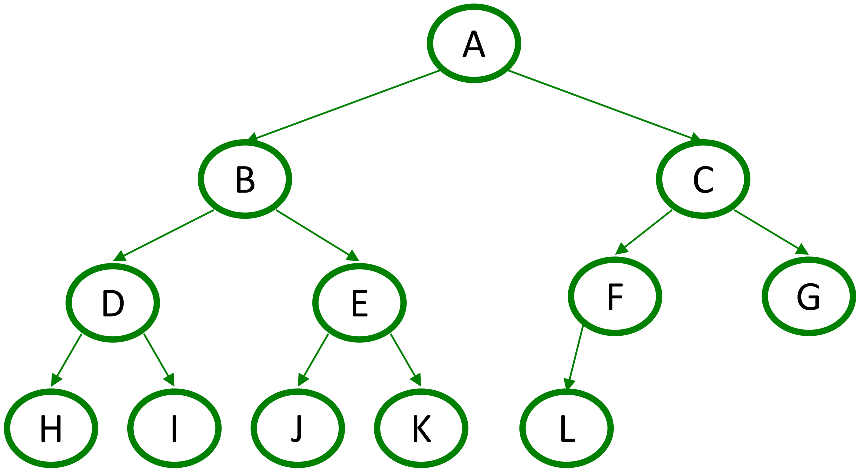
Summary: Priority Queue ADT





- Priority Queue ADT:
  - `insert` comparable object,
  - `deleteMin`
- Binary heap data structure:
  - Complete binary tree
  - 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

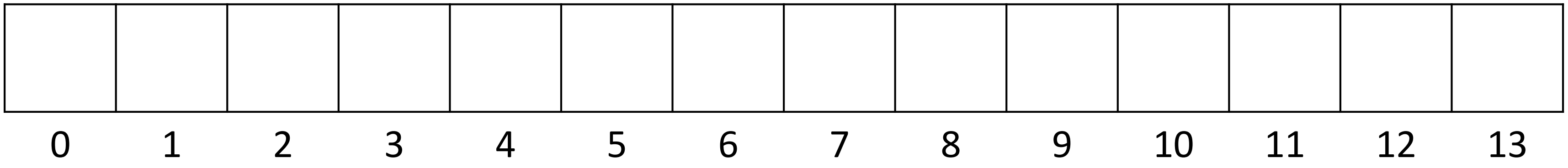
Binary Trees Implemented with an **Array**



From node `i`:

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

(wasting index 0 is convenient for the index arithmetic)



Judging the array implementation

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 would waste more space if tree were not complete
- 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	15	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)

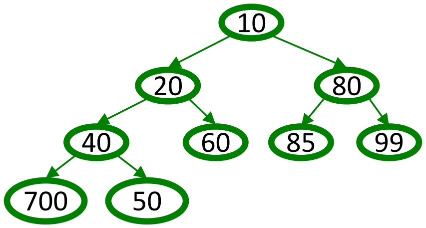
	4	32	16	67	105	43	
0	1	2	3	4	5	6	7

(extra space for your scratch work a notes)

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;
}
```

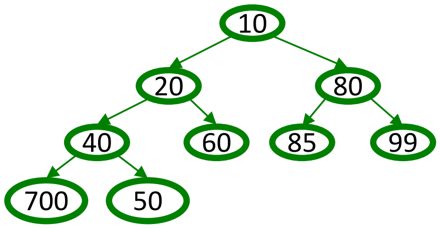


This pseudocode uses ints. In real use, you will have data nodes with priorities.

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

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) {
  left  = 2*hole;
  right = left + 1;
  if(right > size ||
     arr[left] < arr[right])
    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

Example

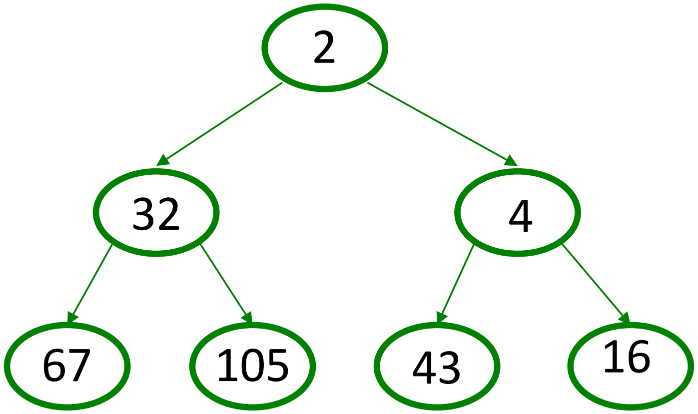
1. `insert` (in this order): 16, 32, 4, 67, 105, 43, 2
2. `deleteMin` once

0	1	2	3	4	5	6	7

Example

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

0	1	2	3	4	5	6	7



Other operations

- `decreaseKey`: given pointer to object in priority queue (e.g., its array index), lower its priority value by p
  - Change priority and percolate up
- `increaseKey`: given pointer to object in priority queue (e.g., its array index), raise its priority value 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 $p = \infty$, then `deleteMin`

Running time for all these operations?

Build Heap

- Suppose you have *n* items to put in a new (empty) priority queue
  - Call this operation `buildHeap`

- *n* `inserts`
  - Only choice if ADT doesn't provide `buildHeap` explicitly
  -

- Why would an ADT provide this unnecessary operation?
  - Convenience
  - Efficiency: an $O(n)$ algorithm
  - Common issue in ADT design: how many specialized operations

`heapify` (Floyd’s Method)

1. Use *n* items to make any complete tree you want
   - That is, put them in array indices 1,…,*n*

2. Fix the heap-order property
   - Bottom-up: percolate down starting at nodes one level up from leaves, work up toward the root

`heapify` (Floyd’s Method): Example

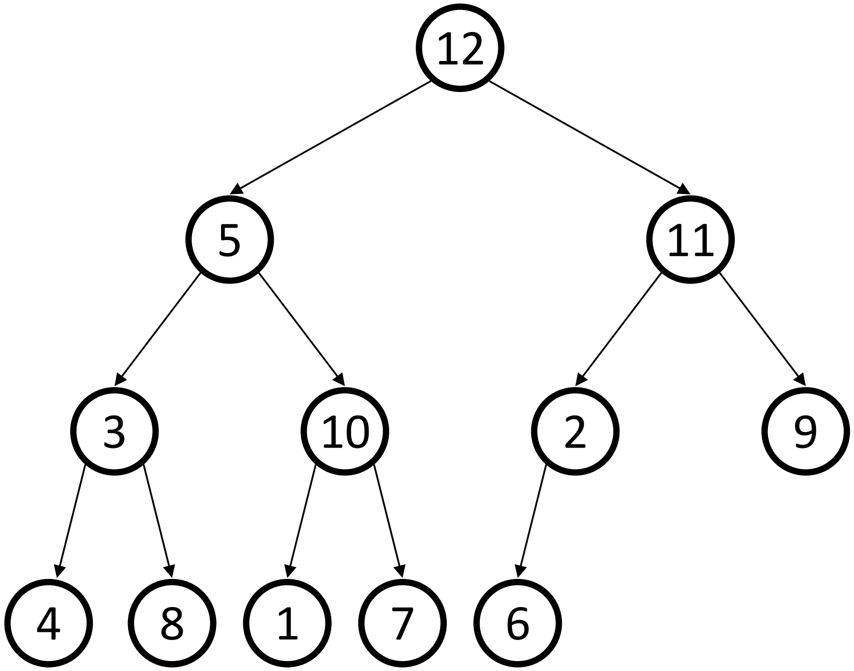
1. Use *n* items to make any complete tree you want
2. Fix the heap-order property from bottom-up

Which nodes break the heap-order property?

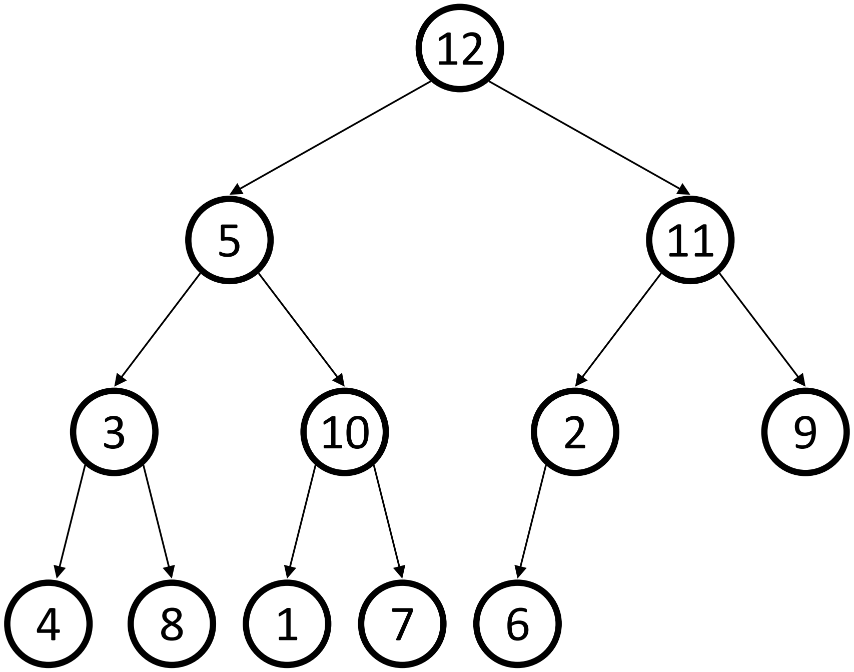
Why work from the bottom-up to fix them?

Why start at one level above the leaf nodes?

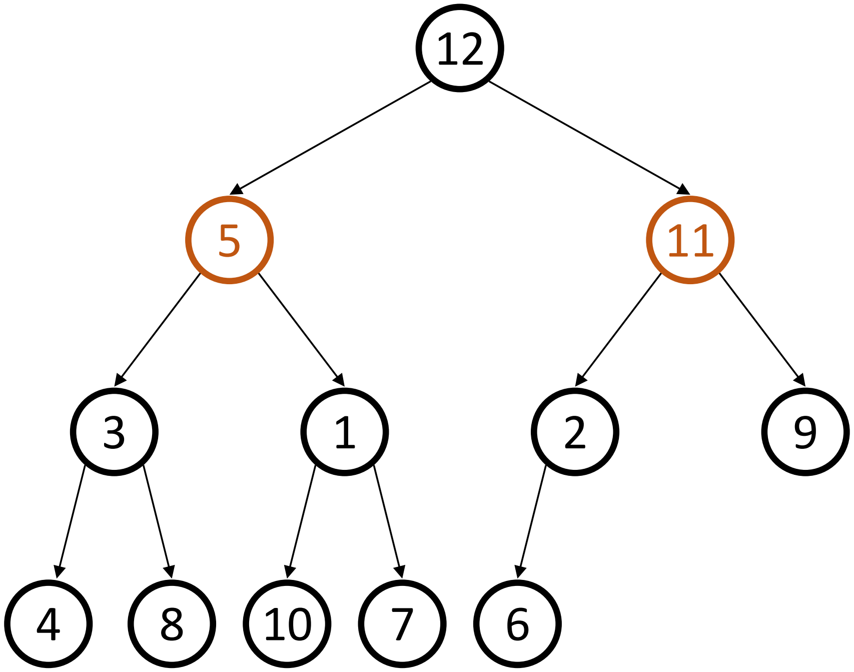
Where do we start here?



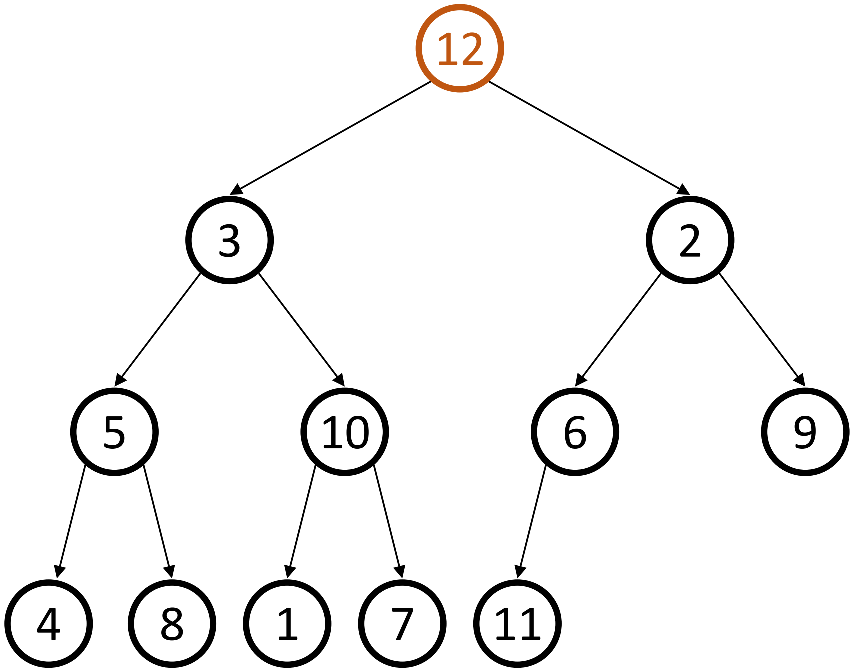
`heapify` (Floyd's Method): Example



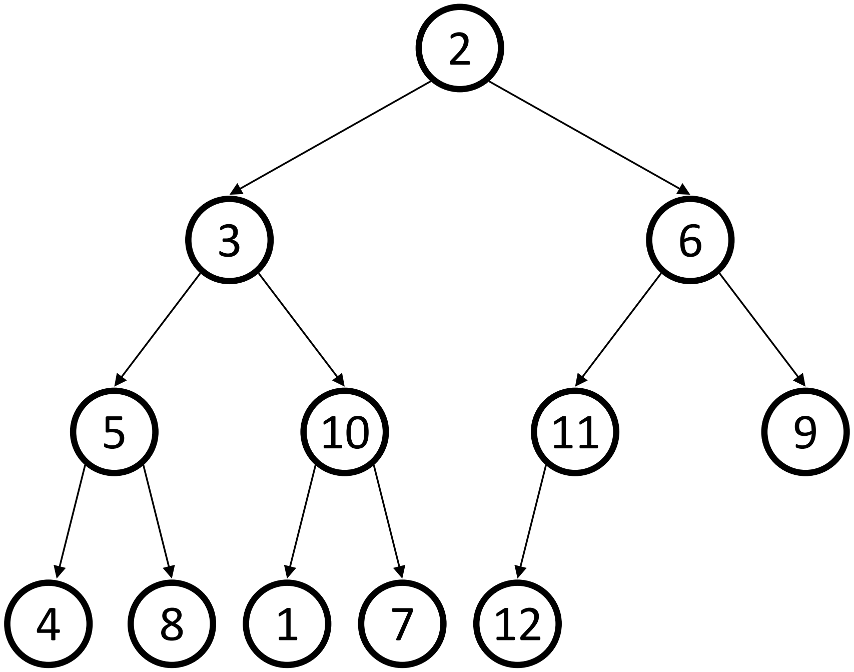
`heapify` (Floyd’s Method): Example



`heapify` (Floyd’s Method): Example



`heapify` (Floyd's Method): Example



`heapify` (Floyd’s Method)

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

But is it right? … it “seems to work”

- Let’s *prove* it restores the heap property
- Then let’s *prove* its running time

Correctness

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

Loop Invariant: For all `j` > `i`, `arr[j]` is higher priority 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 more iteration: loop body and `percolateDown` make `arr[i]` higher priority than children without breaking the property for any descendants

So after the loop finishes, all nodes are less than their children

Efficiency

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

Easier argument: `buildHeap` is where *n* is `size`

- loop iterations
- Each iteration does one `percolateDown`, each is

This is correct, but there is a more precise (“tighter”) analysis of the algorithm…

Efficiency

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

Better argument: `buildHeap` is where *n* is `size`

- `size/2` total loop iterations: *O*(*n*)
- 1/2 the loop iterations `percolateDown` at most
- 1/4 the loop iterations `percolateDown` at most
- 1/8 the loop iterations `percolateDown` at most
- …
- ((1/2) + (2/4) + (3/8) + (4/16) + …) < 2 (page 4 of Weiss)

$$\sum_{i=1}^{\infty} \frac{i}{2^i} = 2$$

Lessons from `buildHeap`

- Without providing `buildHeap`, clients can implement their own that runs in worst case
- By providing a specialized operation (with access to the internal data), we can do worst case
  - Intuition: Most data is near a leaf, so better to percolate down
- Can analyze this algorithm for:
  - Correctness: Non-trivial inductive proof using loop invariant
  - Efficiency:
    - First (easier) analysis proved it was O(*n* `log` *n*)
    - Tighter analysis shows same algorithm is *O*(*n*)

Other branching factors for Heaps

***d*-heaps**: have *d* children instead of 2

- Makes heaps shallower

Example: 3-heap

- Only difference: three children instead of 2
- Still use an array with all positions from 1 … heapSize

Indices for 3-heap

Index	Children Indices
1	
2	
3	
4	
5	
…	…

Wrapping up Heaps

- What are heaps a data structure for?
- What is it usually implemented with? Why?
- What are some example uses?