CSE 373: Data Structures and Algorithms

Lecture 13: Finish Binary Heaps

Instructor: Lilian de Greef
Quarter: Summer 2017

Today

- Announcements
- Binary Heaps
  - Wrap up array representation of tree
  - Floyd's Method of `buildTree`
  - d-heaps

Announcements

- Midterm on Friday
  - Will start at 10:50, will end promptly at 11:50 (even if you're late), so be early
  - Anything we've covered is fair game (including this lecture)
  - Only bring pencils and erasers
  - Turn off / silence and put away any devices (e.g. phone) before exam
- Section
  - Will go over solutions for select problems from practice set
  - Practice set posted on course webpage (under Sections)
  - Recommendation: do the practice problems, then use section to go over the questions you found hardest (there isn't enough time to cover all of them)
- Homework 3 grades come out today!
- Course feedback today! (anonymous, confidential, something I have set up)

A cool MyClient!

```
Welcome to Word Association Game : (enter "exit"):
Guess the association.
Word : dirt free
1 : capable
2 : superficiality
3 : dainty
Enter (1 ,2 or 3)
>>>3
Correct!
Points : 1

Word : guts
1 : strength
2 : brass
3 : energid
Enter (1 ,2 or 3)
>>>2
Strike : 1
Correct answer was strength

Word : avian
1 : see life
2 : nesting
3 : be a success
Enter (1 ,2 or 3)
>>>
```

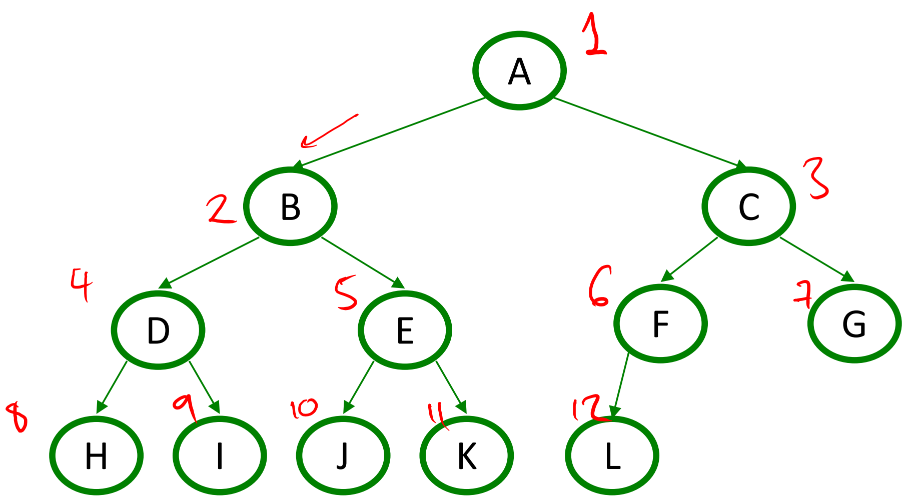
Another cool MyClient!

```
Welcome to "Let's Keep It Short!"
Enter the thesaurus prefered:
large_thesaurus.txt
Please enter the text you would like to keep it short (enter "exit" to exit):
Here is an example sentence to show off this cool program!
Here you go! This should keep you under the word limit (if there is one)!

now an type mot to fix off OK program!

Please enter the text you would like to keep it short (enter "exit" to exit):
Entertaining purple elephants give children enjoyable presents.
Here you go! This should keep you under the word limit (if there is one)!

fun rod elephants be get fair presents.

Please enter the text you would like to keep it short (enter "exit" to exit):
I hope my instruction of the abstraction and comparison between data structures
is illuminating.
Here you go! This should keep you under the word limit (if there is one)!

hope gen of the tic and sub between ken structures illuminating.

Please enter the text you would like to keep it short (enter "exit" to exit):
```

Finishing up Binary Heaps

Data Structure for Priority Queue, implemented with arrays!

Binary Trees Implemented with an **Array**



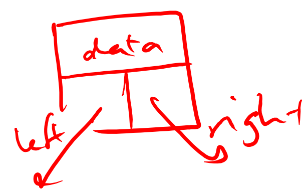
From node `i`:

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



3 pieces of info

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

Pro

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)

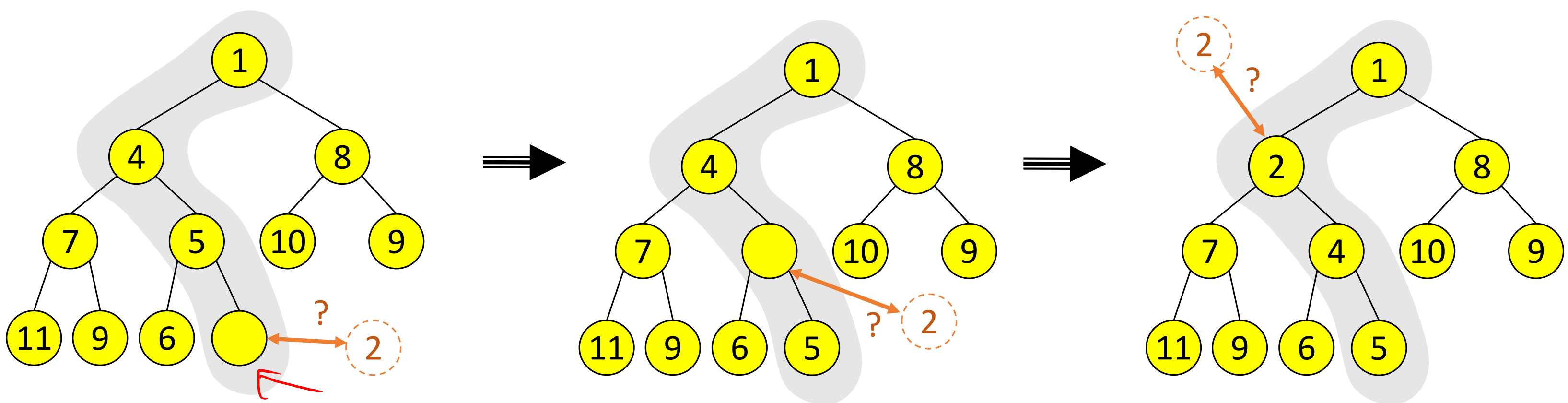
Pros outweigh cons: min-heaps almost always use array implementation

Heap `insert`:

minHeap: lower value has higher priority.

1. Put new data in new location *(preserve structure property)*
2. **Percolate up:** *(restore heap property)*
   - If higher priority than parent, swap with parent
   - Repeat until parent is more important or reached root

maxHeap: larger values have higher priority

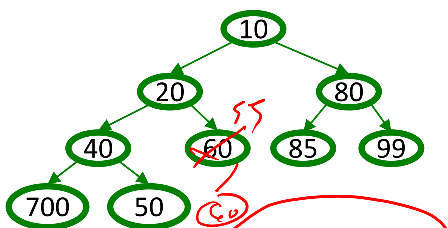


Semi-Pseudocode: `insert` into binary heap

```
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;
}
```

check if at root or has higher priority



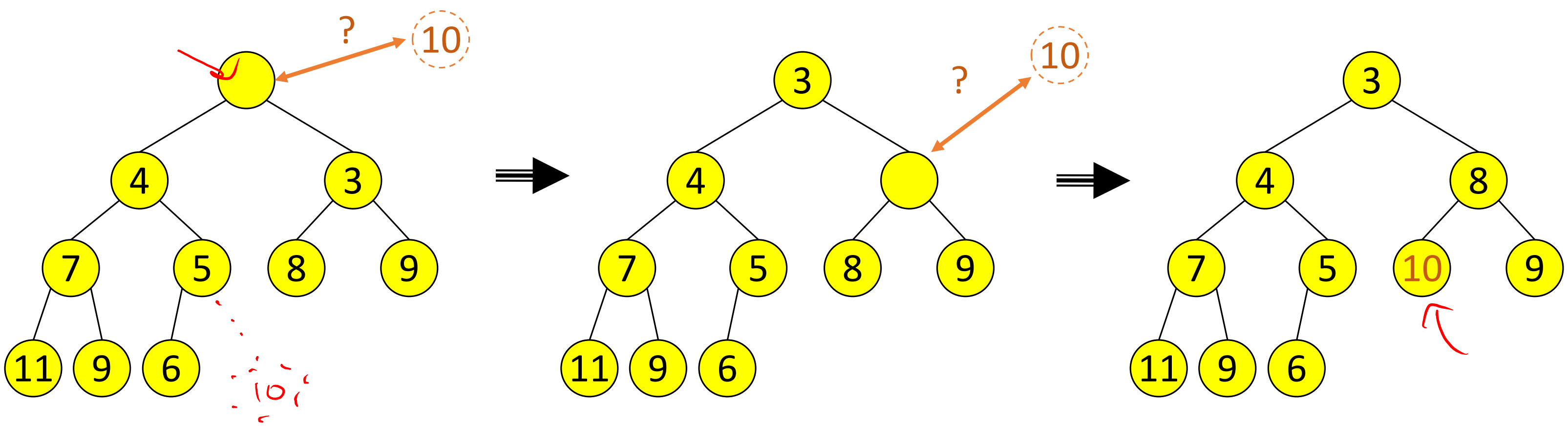
This pseudocode uses ints. In real use, you will have data nodes with priorities.

	10	**20**	**80**	**40**	~~**60**~~ 55	**85**	**99**	**700**	**50**	~~55~~ 60			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

10

Heap `deleteMin`:

1. Remove (and later return) item at root
2. "Move" the last item in bottom row to the root *(preserve structure property)*
3. **Percolate down:** *(restore heap property)*
   - 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

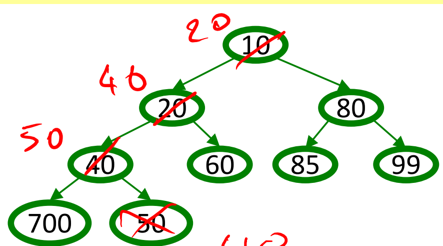


Semi-Pseudocode: `deleteMin` from binary heap

```
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;
}
```

is it a leaf?

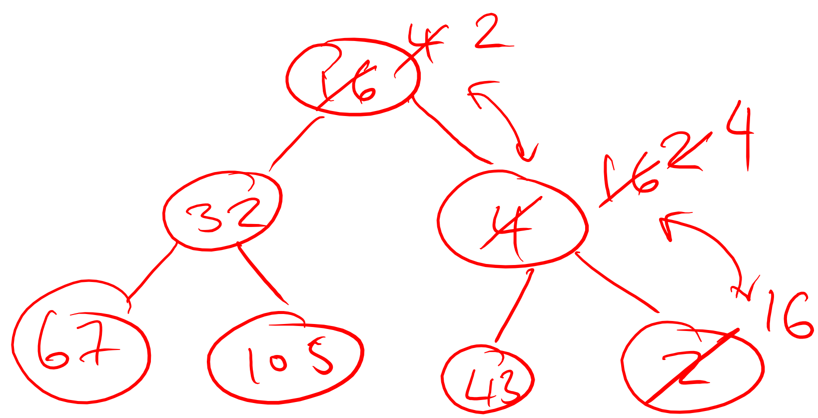


	~~50~~ 20	~~50~~ 40		50									
	~~10~~	~~20~~	80	~~40~~	60	85	99	700	~~50~~				
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Example

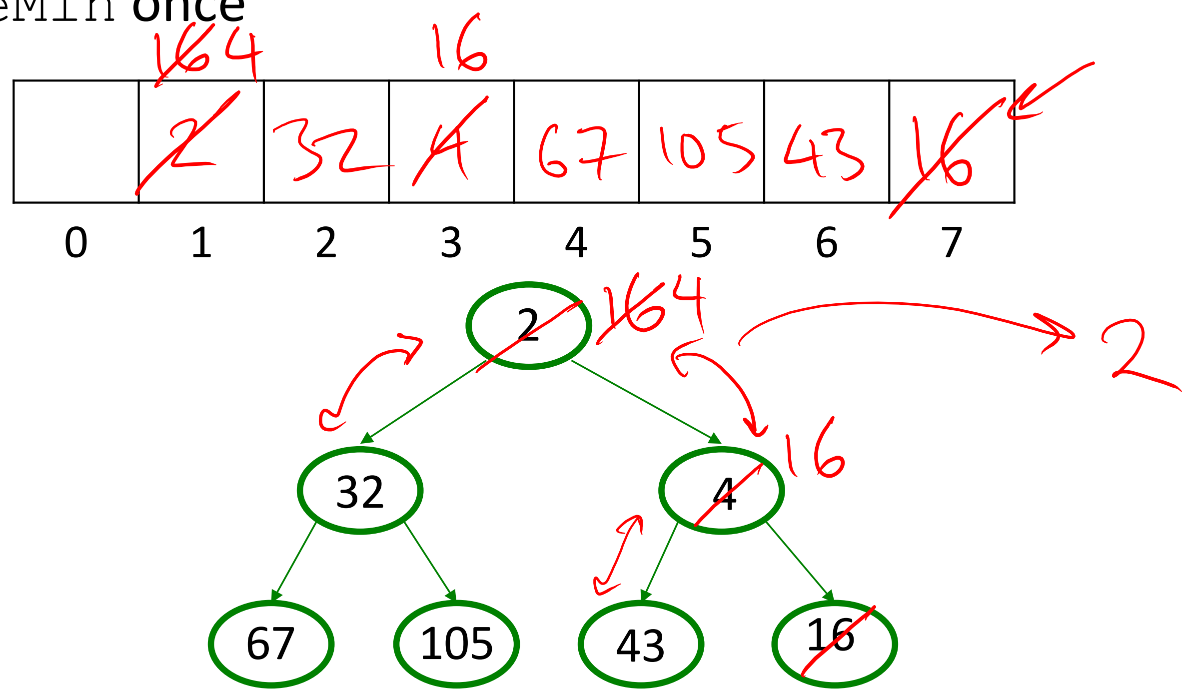
1. `insert` (in this order): 16, 32, 4, 67, 105, 43, 2
2. `deleteMin` once

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



Example

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



Other operations (Bonus Material)

- `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
  - Run time: $O(n \log n)$

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

`heapify` (Floyd's Method)

1. Use *n* items to make any complete tree you want (structure property)
   - 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

12, 5, 11, 3, 10 . . .

Which nodes break the heap-order property?

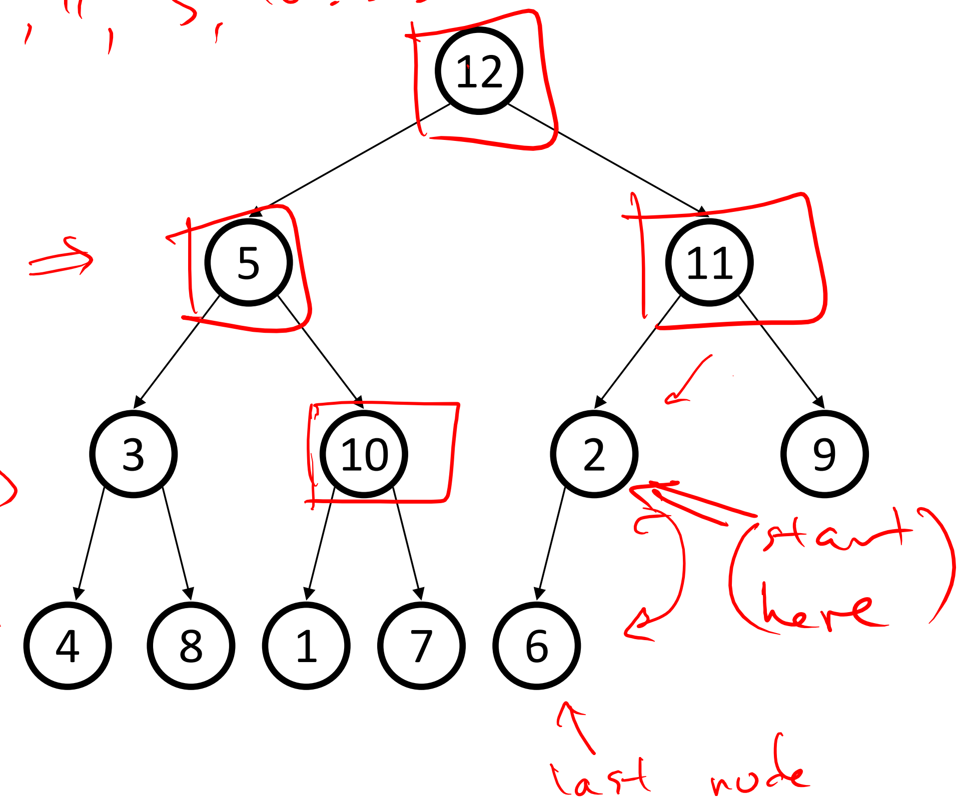
Why work from the bottom-up to fix them?

Save repeated work, top-down is same as inserting n nodes

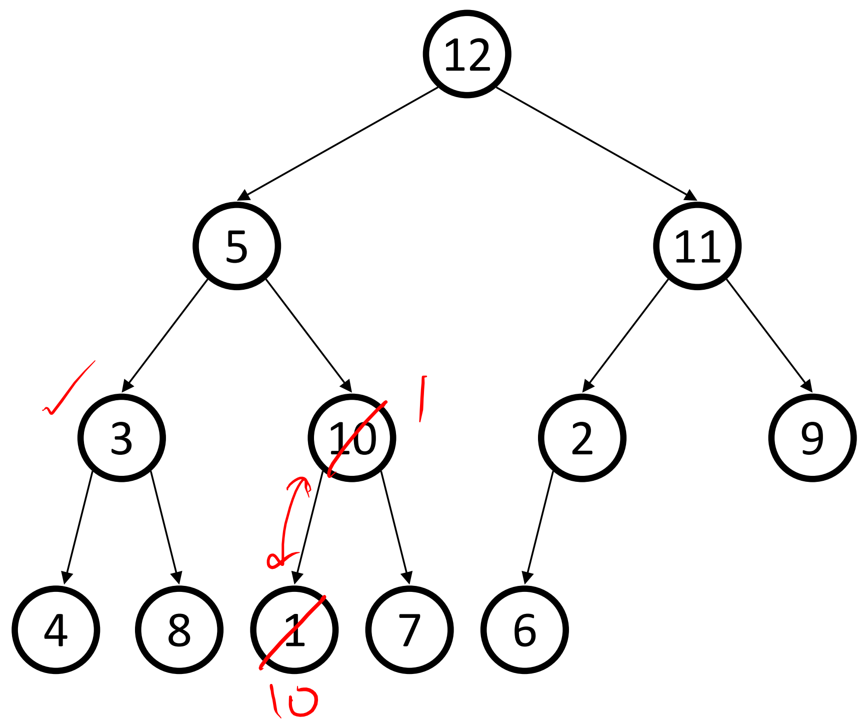
Why start at one level above the leaf nodes?

Leaves can't percolate down

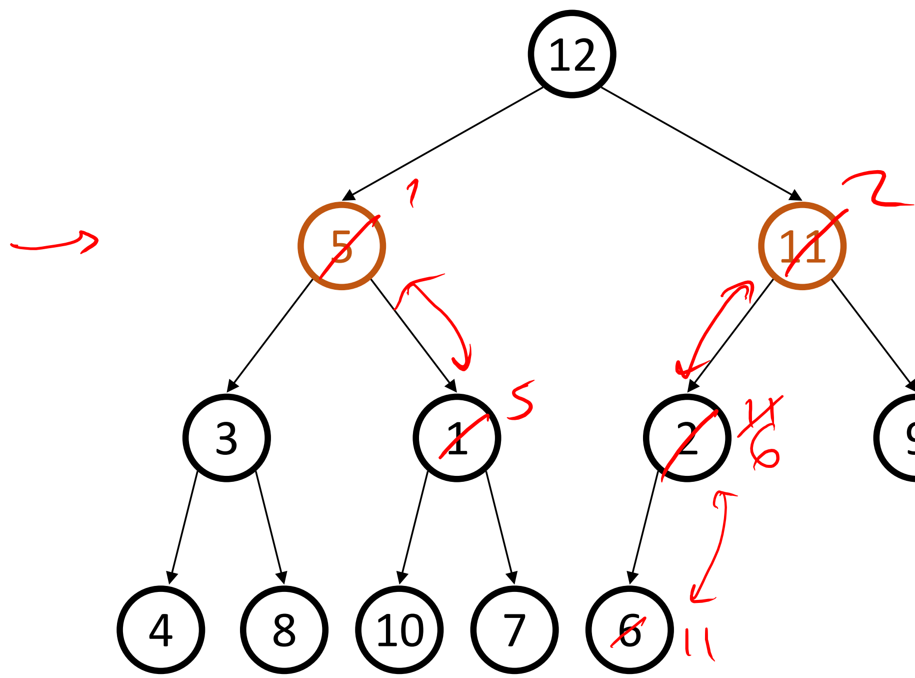
Where do we start here?



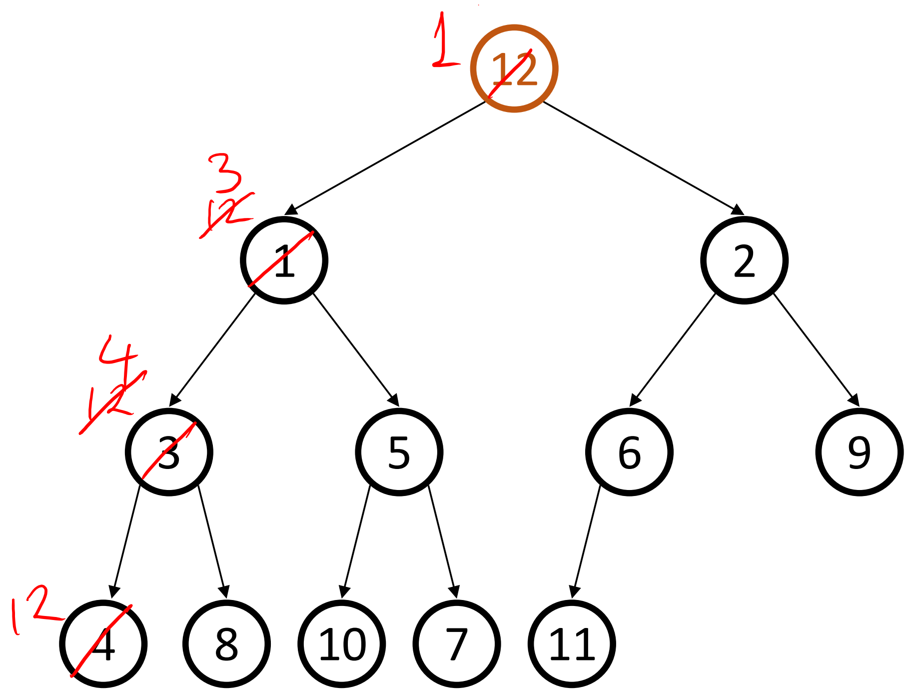
`heapify` (Floyd's Method): Example



heapify (Floyd's Method): Example

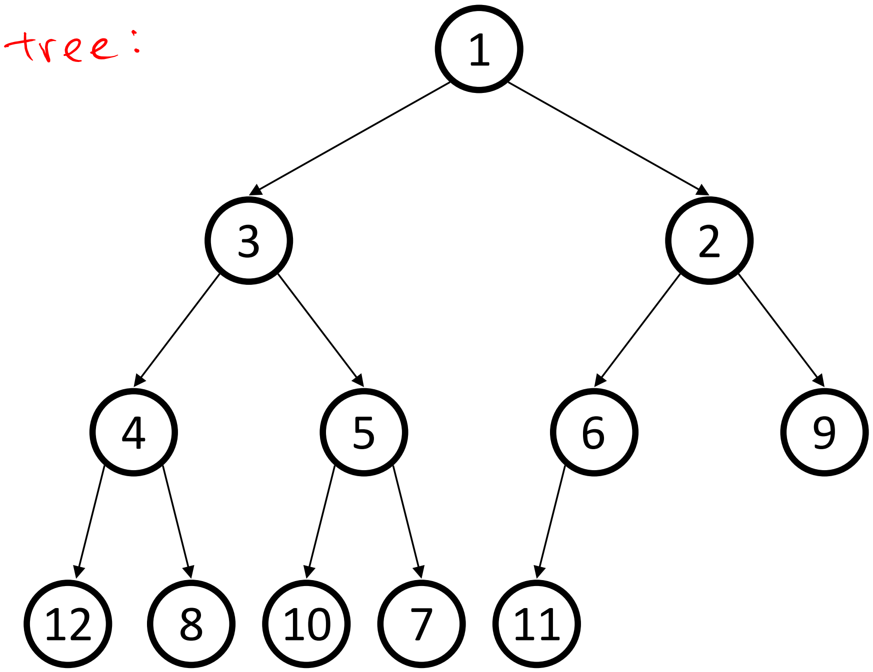


`heapify` (Floyd's Method): Example

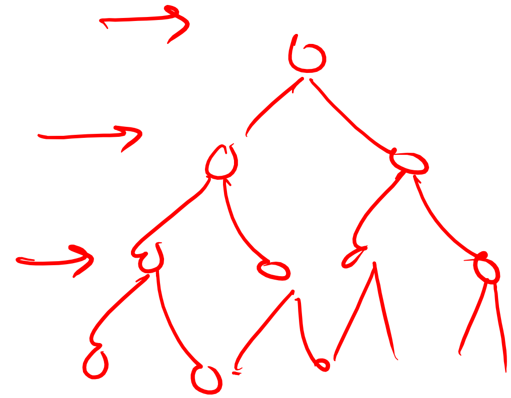


`heapify` (Floyd's Method): Example

The final tree:



`heapify` (Floyd's Method)



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

Start at parent of last leaf node

But is it right? … it "seems to work"

- Let's *prove* it restores the heap property ← (Correctness)
- Then let's *prove* its running time ← (Efficiency)

Correctness

Use Proof by Induction ☺

```
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;
  }
}
```

Remember, Asymptotic Analysis means we're looking at $n \to \infty$ (really REALLY large values of n) for the WORST CASE

Easier argument: `buildHeap` is $O(n \log n)$ where *n* is `size`

- $O(size/2)$ loop iterations $\leftarrow O(n)$
- Each iteration does one `percolateDown`, each is $O(\log n)$

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 $O(n)$ where *n* is `size`

- `size/2` total loop iterations: *O*(*n*)
- 1/2 the loop iterations `percolateDown` at most 1 step
- 1/4 the loop iterations `percolateDown` at most 2 steps
- 1/8 the loop iterations `percolateDown` at most 3 steps
- …
- ((1/2) + (2/4) + (3/8) + (4/16) + …) < 2 (page 4 of Weiss) $\sum_{i=1}^{\infty} \frac{i}{2^i} = 2$

$\Rightarrow$ At most 2*(size/2) total percolate steps. $\Rightarrow$ $\boxed{O(n)}$!

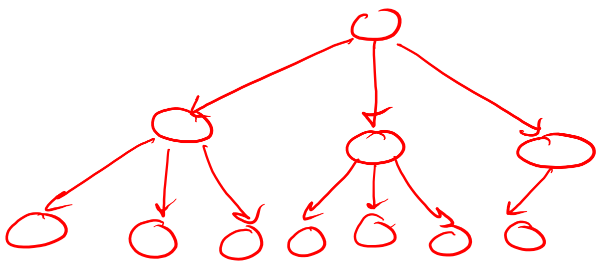
Lessons from `buildHeap`

- Without providing `buildHeap`, clients can implement their own that runs in $O(n \log n)$ worst case
- By providing a specialized operation (with access to the internal data), we can do $O(n)$ 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
2	5, 6, 7
3	8, 9, 10
4	11, 12, 13
5	14, 15, 16
…	…

Wrapping up Heaps

- What are heaps a data structure for? Priority Queue ADT
- What is it usually implemented with? Arrays
  Why? Space and time efficiency (fast shift operations)
- What are some example uses?
  (I'll leave that as an exercise for you :))