Priority Queues & Heaps

CSC263 Week 2

Announcements

- Tutorial 1 Quiz due Thursday 9am
- New to course this week? Check out Quercus
  - Syllabus
  - Unannotated Lecture notes
  - Annotated lecture notes from last week
- Extra office hours – Wednesday 3-4pm BA 2270

Designing a Data Structure for Priority Queue ADT

Data

A collection of items which each have a priority

Operations

Insert(PQ, x, priority)
FindMax(PQ)
ExtractMax(PQ)

Example sequence used across various implementations:

IN(8), IN(5), IN(10), IN(3), FM(), IN(16), EM(), EM(), IN(7)

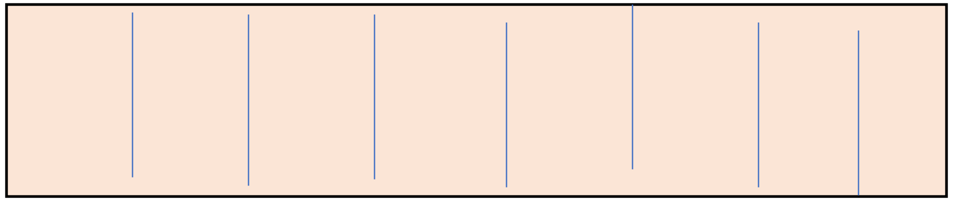
Approach 0: An unsorted linked list

Discussed in DISCOVER module

Insert in O(1)
FindMax in O(n)
ExtractMax in O(n)

Approach 1: An unsorted Array

nothing here



IN(8), IN(5), IN (10), IN (3), FM(), IN(16), EM(), EM(), IN(7)

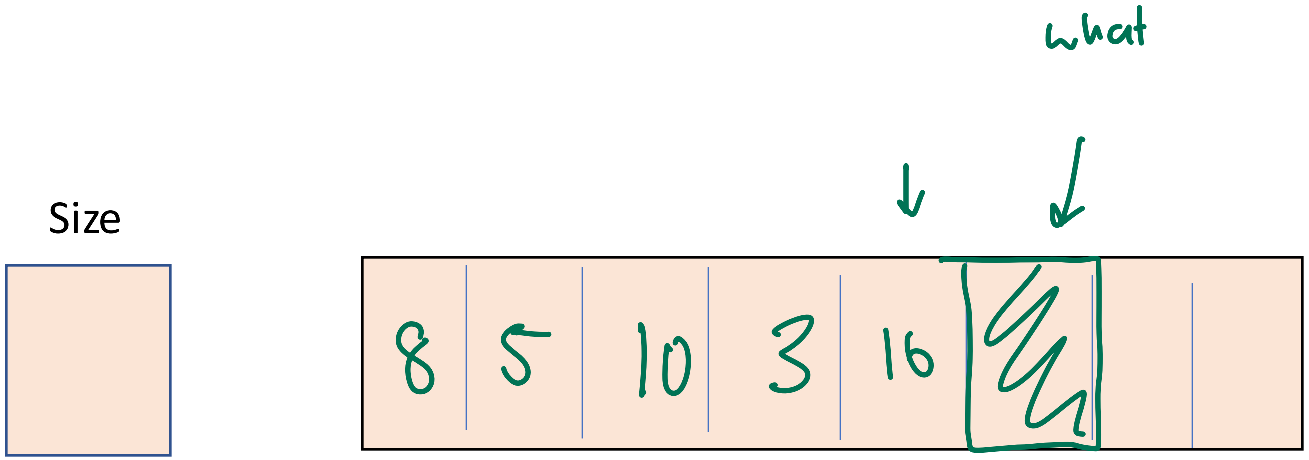
Important Side Note

- Both Python and Java often hide the complexity of operations
- Appending an element into a Python list

```
L.append(x)
```

In this course we want to write and analyze algorithms from the simple operations that don't depend on hidden complexity.

Approach 1: An unsorted Array



insert O(1)

Find-max O(n)

O(n) to find max

O(1) to possibly replace it.

extract max

IN(8), IN(5), IN(10), IN (3), FM(), IN(16), EM(), EM(), IN(7)

do nothing

Approach 2: A Sorted Array

What if we kept the items in the array in sorted order?

Size

4

10	8	5	3				

O(1) O(n)

IN(8), IN(5), IN(10), IN(3), || FM(), IN(16), EM(), EM(), IN(7)

everything has to be shifted O(n)

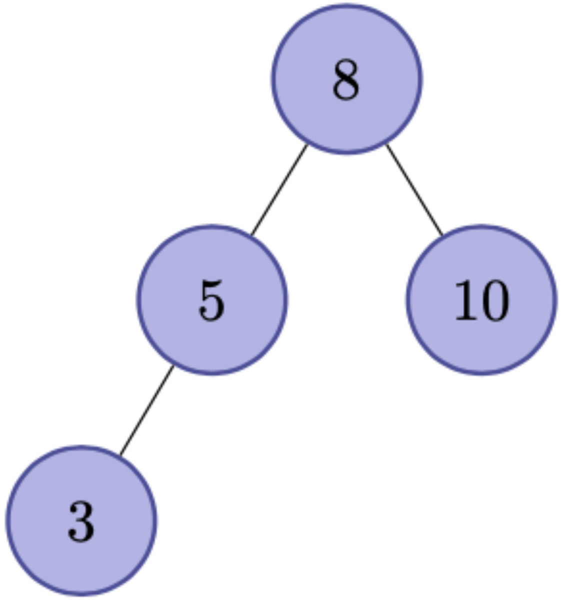
Approach 3: An Ordered Linked List

What if we kept the items in an ordered linked list?

IN(8), IN(5), IN(10), IN(3), FM(), IN(16), EM(), EM(), IN(7)

Approach 4: A Binary Search Tree

What if we kept the items in a binary search tree?



Insert → worst case still O(n)

Extract max for balanced O(log n)

IN(8), IN(5), IN(10), IN(3), FM(), IN(16), EM(), EM(), IN(7)

Heaps

shape

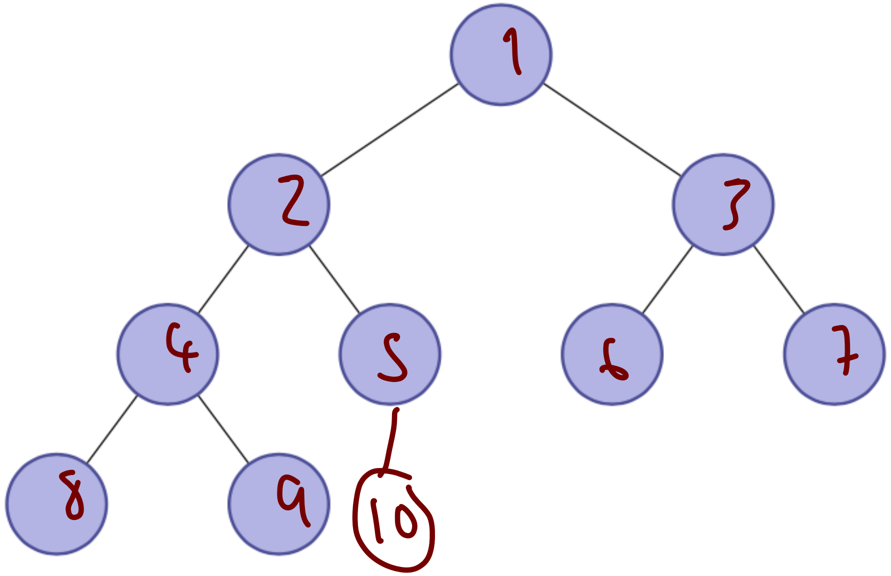
- Based on a nearly complete binary tree
- Heap Property determines relationship between values of parents and children
- Kind of sorted: enough to make query operations fast while not requiring full sorting after each update.
- Stored in an array

She said it is wow! very cool!

Nearly Complete Binary Tree

- Binary tree
- Every row is completely filled except possibly the lowest row
- The lowest row is filled from the left

not the node values, just the order of the node

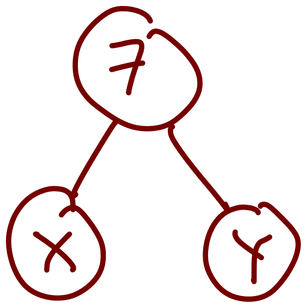


for a "complete" binary tree the last row is always filled up, so this is nearly complete

Heap Property

7 always $\geq X$

$\geq Y$



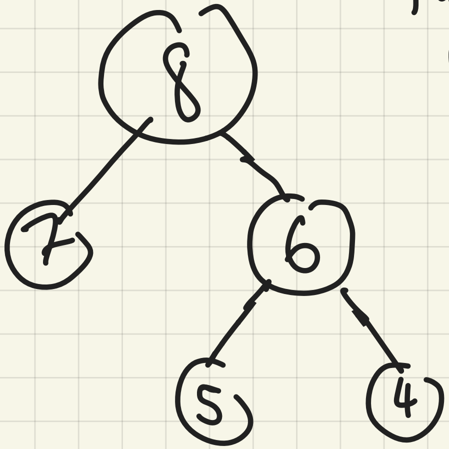
- Max Heap

    The value at every node is equal to or greater than the value of its immediate children

- Min Heap (you fill this one in)

The value at every node is equal to or lesser than the value of its immediate children.

valid heap

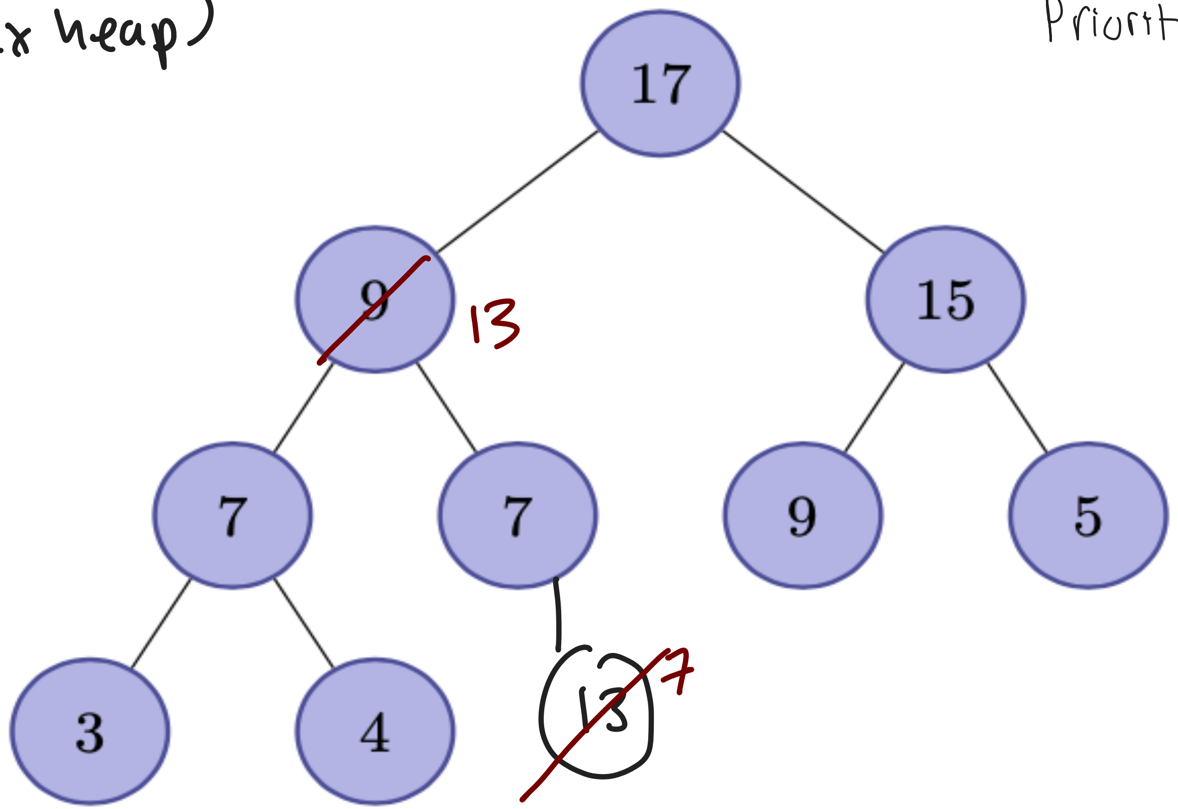


the shape is wrong even if the numbers are right !!

Implementing the PQ Operations: Insert

(max heap)

Priority Que



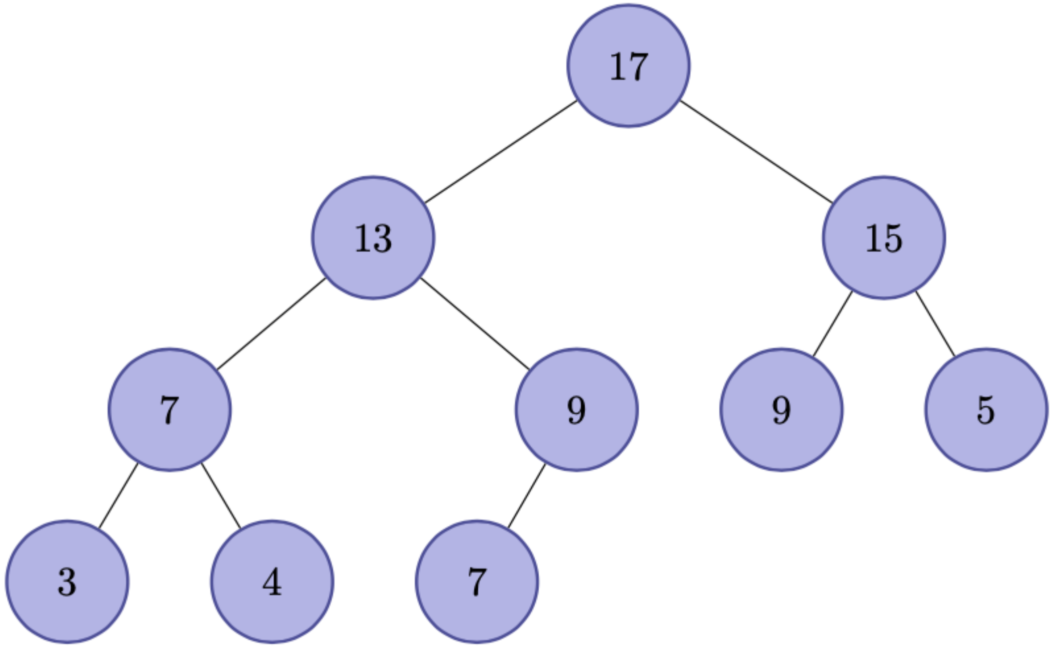
Insert(13)

no example can be worse than this

- worst case → $O(\log n)$
- instance to come up with

- Increment heapsize and add element at next position
- Result might violate heap property so bubble up
- Running time? Θ(height of tree) $\Theta(\log n)$

Implementing the PQ Operations: FindMax

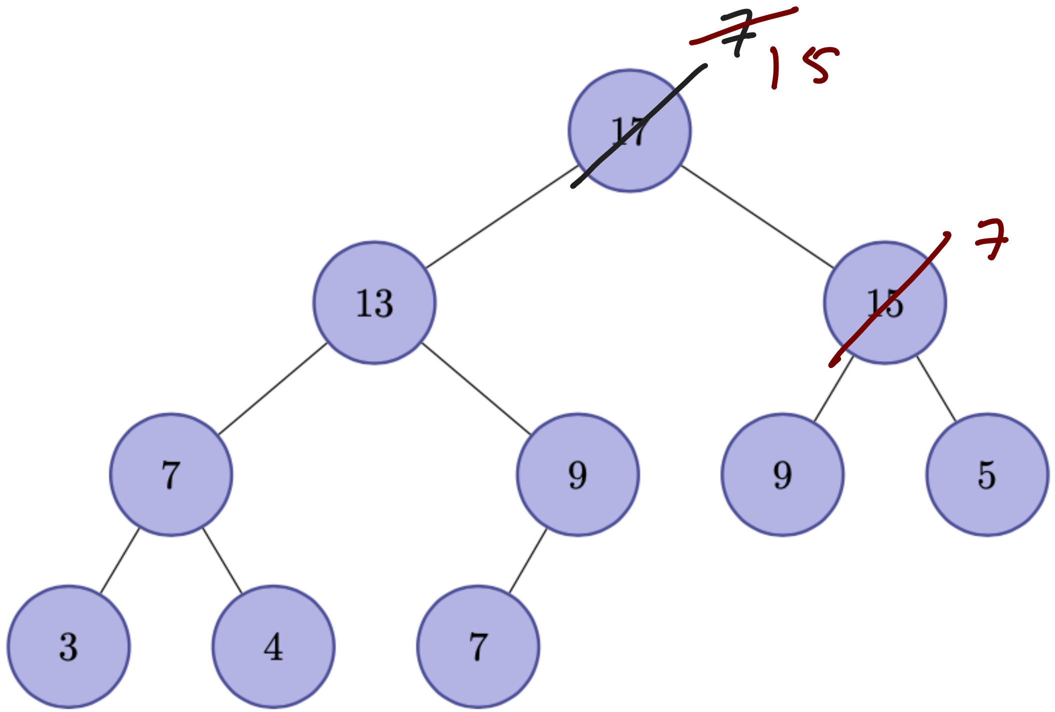


Findmax()

- Doesn't change heap
- Running time?

Implementing the PQ Operations: ExtractMax

① make shape correct first. so have to choose 7



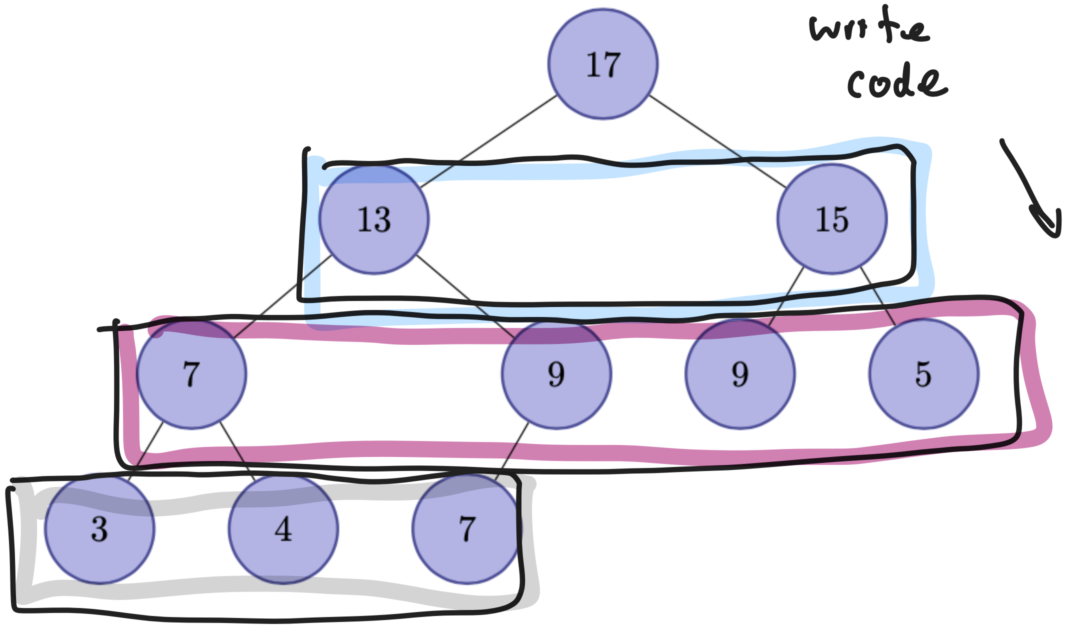
precondition to bubble down is the subtrees are heaps but the whole thing does not have to be heaps.

- Remove and return root element
- Strategy: restore shape first then fix heap property
- Bubble down also called ____max heapify____
- Running time? ____Θ(height) $\Theta(\log n)$____

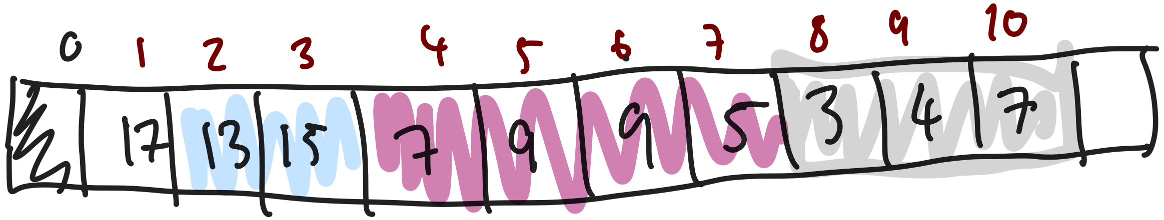
Now the really cool bit!!!

not efficient if we have to keep this "node" structure.

write code



① can keep as array because it is almost complete BST → no holes in array.



only one

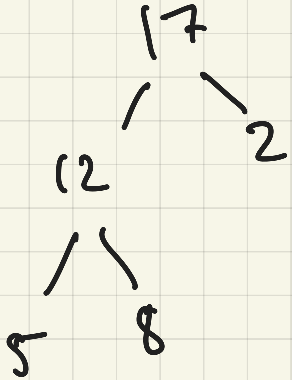
convention is to put whatever to 0

- Use an array to store the heap (not linked nodes and references)
- Convention: use 1-based indexing so root is at element 1
- For node at position i, left child is at $2i$, right child is at $2i+1$, and parent is at $\lfloor i/2 \rfloor$. or i // 2

Is this a valid heap?

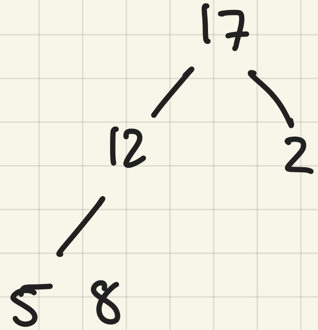
[17, 12, 2, 5, 8, || 0, 0, 0] → not part

yes



heapsize = 5

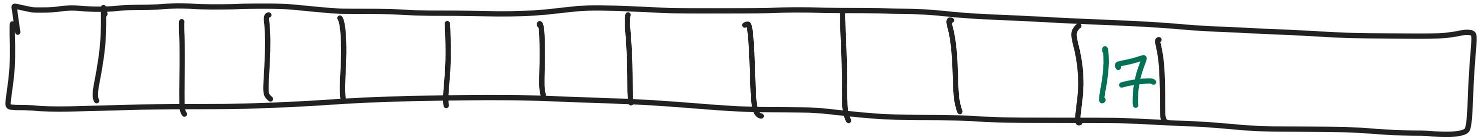
[17, 12, 2, 5, 8, 16, 0 72] → junk



yes

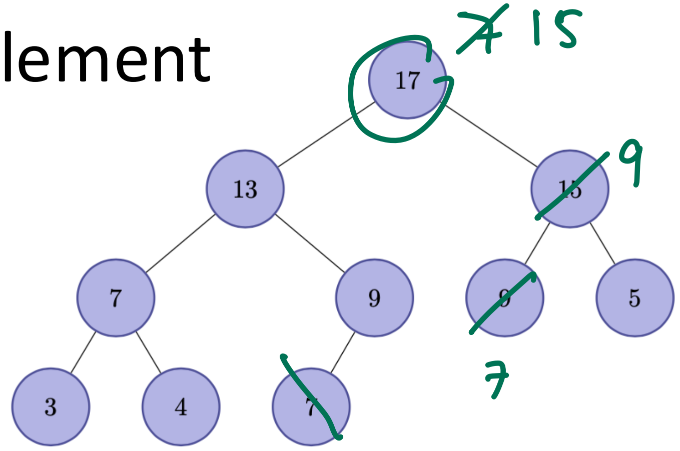
heap to sorted → consider to heap?

Heap Sort



Assuming we start with a valid heap, how do we get a sorted list?

Notice that the root of the heap stores the maximum element



KEY IDEA

- Remove the root and put it in an array at the end
- Decrement heapsize
- Restore the heap property

① remove root

②

2nd KEY IDEA

- Do this in place since replacement item for root was in the position where we want to put the root anyway.

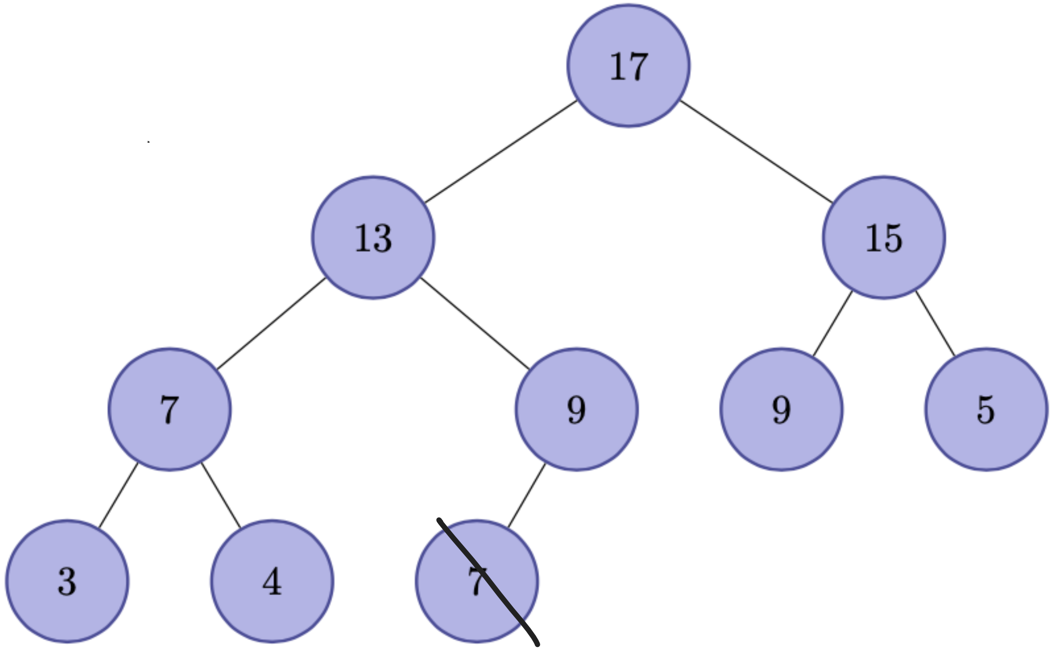
Heap Sort

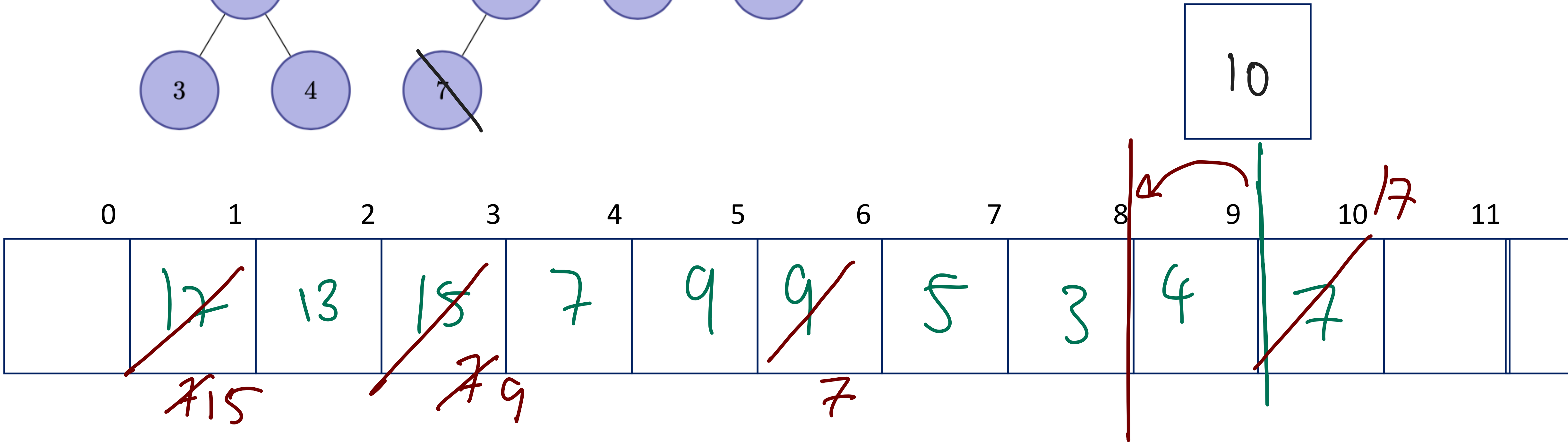
Going from a Max Heap to a listed sorted in non-decreasing order:

at least $O(n(\log n))$

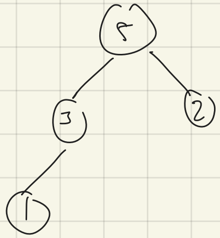
when we're heapifying in array comparing

· root to children → $2i$ & $2i+1$



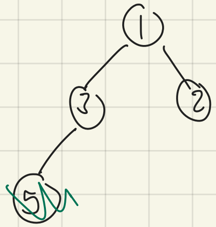


heapify
from
video 4



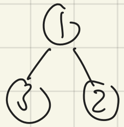
5	3	2	1

swap →

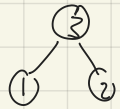


1	3	2	~~5~~

sorted

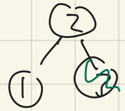


heapify →



3	1	2	~

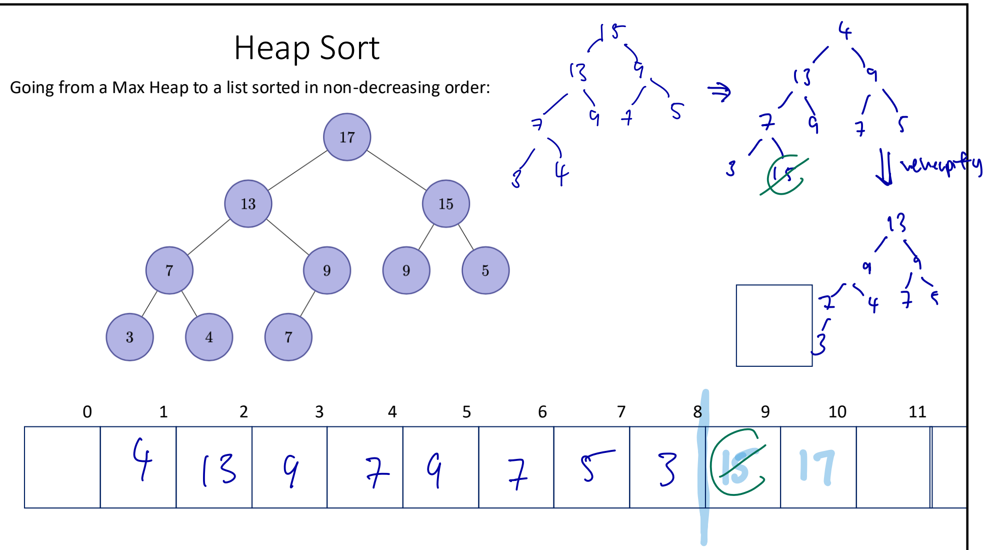
swap →



2	1	~~3~~	~

Heap Sort

Going from a Max Heap to a list sorted in non-decreasing order:

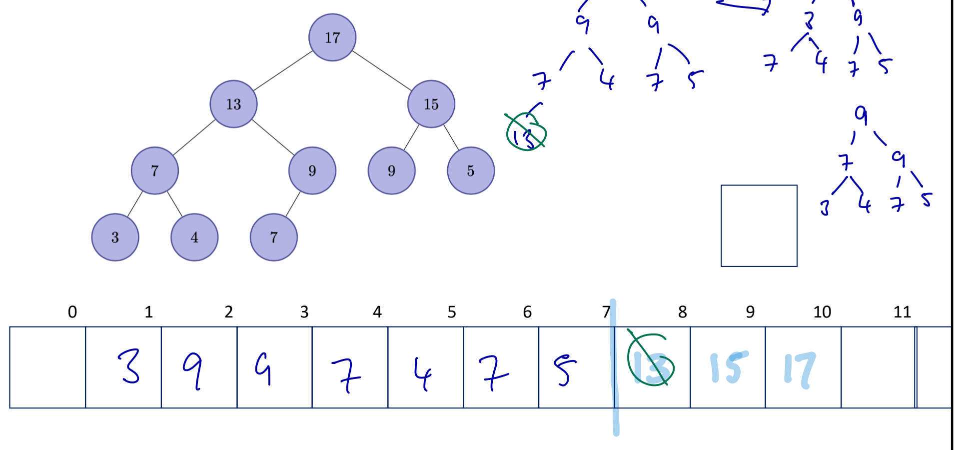


Heap Sort

swap

reheapify

Going from a Max Heap to a list sorted in non-decreasing order:



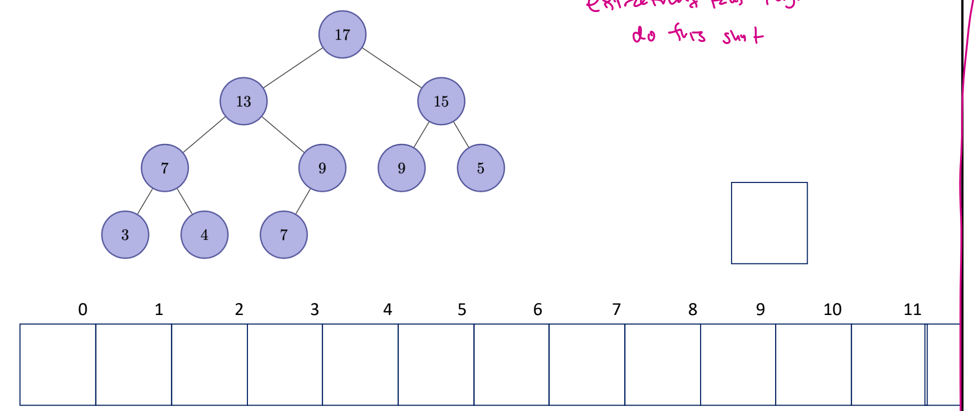
Analyzing

Heap Sort

bounded by $O(n \log n)$

Going from a Max Heap to a list sorted in non-decreasing order:

extraction takes $\log n$ but we do this shit



Bubble-down or max_heapify

```
def max_heapify(L, i):
    l = left(i)
    r = right(i)
    if l <= L.heapsize and L[l] > L[i]:
        largest = l
    else:
        largest = i
    if r <= L.heapsize and L[r] > L[largest]:
        largest = r
    if largest != i:
        exchange L[i] with L[largest]
        max_heapify(L, largest)
```

Building a Heap in the First Place

- Our discussion of heap sort only talked about going from a valid heap to a sorted list.
- But typically, we want sorting algorithms to go from an unsorted list to a sorted list.
- So, how do we efficiently go from an unsorted list to a valid heap?

Building a Heap in the First Place

Suppose we have unsorted array A of length n. How do we turn this into a heap?

- Approach 1: Start with an empty (i.e., garbage-filled) array B and size 0 (an empty heap) and INSERT each item from A into heap B
- Approach 2: Start with the same array A and call max_heapify(A, i) on each item in A working backwards (i goes from n to 1)

A = [2, 7, 26, 25, 19, 17, 1, 90, 3, 36]

Building a Heap in the First Place

Check your understanding by going to https://visualgo.net/en/heap

1. When the site loads, you are in E-lecture mode. Press ESC to get to the visualizations.
2. From the bottom left, select Create A – O(N log N)
   - Confirm your own building of the heap for approach 1 was correct
3. From the bottom left, select Create A – O(N)
   - Check your approach 2

Building a Heap in the First Place: Approach 1

0 Start with empty heap

1 insert into correct shape

2 make values correct !!

insert - $O(\log n)$

do it n times + size of |B|

$O(n \log n)$

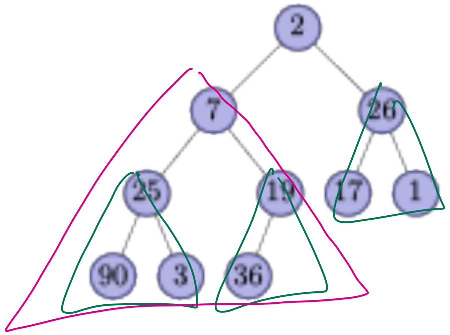
find lower bound? : worst case scenario ....

- sorted in ascending order $\Omega(n \log n)$

$\Theta(n \log n)$

Building a Heap in the First Place: Approach 2

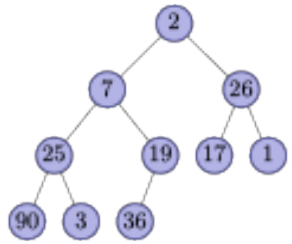
valid_heap all the subtrees



Runtime of Building a Heap

- Using the second approach, we make O(n) calls to max_heapify, and each one takes O(log n), so we immediately get a bound of O(n log n) but we can do better!

Start i at $\lfloor \frac{n}{2} \rfloor$ and move back to $i = 1$



A [2,7,26,25,19,17,1,90,3,36]

Runtime of Building a Heap

Running time of max_heapify(L,i) is proportional to $\log n$ = height

So how many subtrees of each height do we have?

$\frac{n}{2}$ leaves at height 0 — 0 swap

$\frac{n}{4}$ height at 1 — 1 swap

$\frac{n}{8}$ nodes at height 2 — 2 swap

⋮

1 node at height $\log n$ — $\log n$ swaps

Runtime of Building a Heap

$$= \sum_{h=1}^{\lfloor \log n \rfloor} h \times \#\text{ node at height } h$$

$$= \sum_{h=1}^{\log n} h \left\lceil \frac{n}{2^{h+1}} \right\rceil = O(n)$$

Gabriel's Staircase Series:

$$\sum_{k=1}^{\infty} k r^k = \frac{r}{(1-r)^2}, \quad \text{for } 0 < r < 1$$