Heaps - II

- In a max-heap, the max-heap property is satisfied at every node i (except at the root)

  $A[\text{parent}(i)] \geq A[i]$

  ↳ The largest value in a max-heap resides at the root

- A min-heap is organized in the opposite fashion:

  Min-heap property

  $A[\text{parent}(i)] \leq A[i]$, for any node except the root

  ↳ the smallest element in a min-heap is at the root.

- Heap-sort algorithm uses max-heap for sorting in $O(n \lg n)$
- Min-heaps / max-heaps are used to implement priority queues
  eg. processes in the Operating System
- Height of a node in a heap is defined similar to trees:

  [# of edges on the longest simple downward Path from the node to a leaf.

  ↳ Height of heap = height of the root node
- Since a heap of n elements is a complete binary tree, its height is $\Theta(\lg n)$

  ↳ ⇒ Basic operations on a heap are $O(\lg n)$

Max-Heapify — $O(\lg n)$ Maintains Max-heap property

Build-Max-Heap — $O(n)$ Create a Max-heap from a set of unordered values

Heap-Sort — $O(n \lg n)$ Sort an array in-place

Max-Heap-Insert
Max-Heap-Extract-Max
Max-Heap-Increase-key
Max-Heap-Maximum

— Used to implement priority queues

— run in $O(\lg n)$

— Maintaing heap property

Max-Heapify (A, i) : Assumes that subtrees Left (i) & Right (i) are max-heaps

However $A[i]$ might be smaller than its children, thus violate the max-heap property.

— Max-Heapify lets the value $A[i]$ "flow down" in the max-heap so that the tree rooted at i obeys max-heap property

Max-Heapify (A, i)

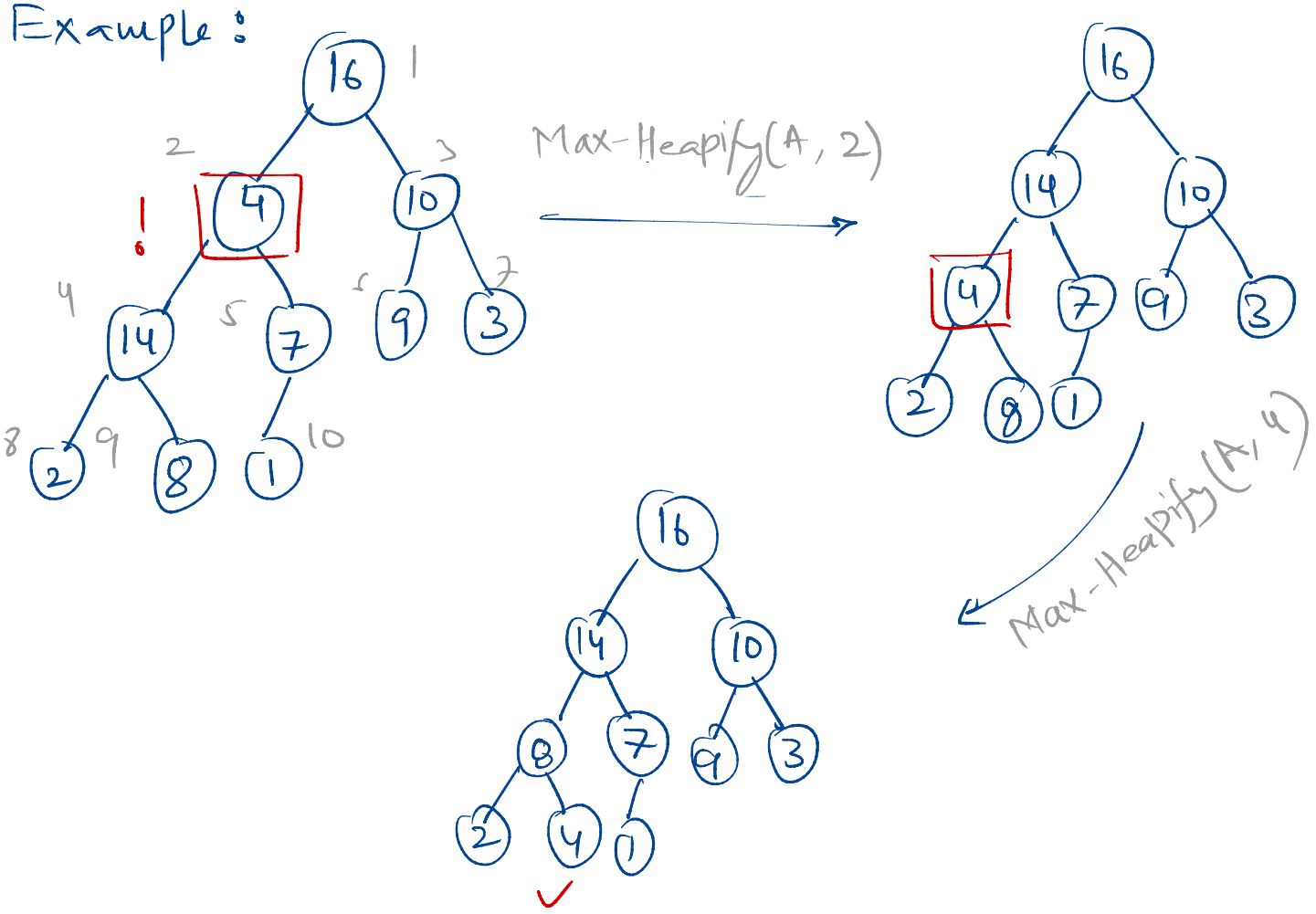
```
l = Left(i)
r = Right(i)
if l ≤ A.heap-size  &  A[l] > A[i]
        largest = l
else    largest = i
if r ≤ A.heap-size  &  A[r] > A[largest]
        largest = r
if largest ≠ i
        exchange A[i] with A[largest]
        Max-Heapify(A, largest)
```

↳ check max of A[i], A[Left(i)], and A[Right(i)]
& swap with A[i]

↳ call max-heapify recursively on the subtree whose root was swapped.

Max-Heapify (A, root) ⟶ $\Theta(\lg n)$

Example:



Complexity analysis: for a tree rooted at i:

$\Theta(1)$: fix up relationship among $A[i]$, $A[\text{left}(i)]$, and $A[\text{Right}(i)]$

+

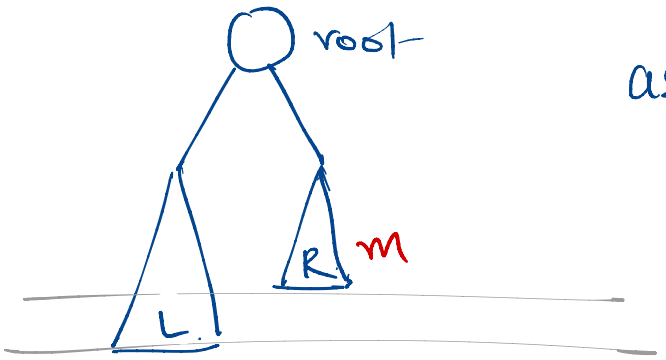
time to max heapify on remaining.

Use result: for a complete binary tree of n elements, the left or right subtree each can have size at most $2n/3$

$\Rightarrow T(n) \leq T(2n/3) + \Theta(1) \Rightarrow T(n) = \lg(n)$

Proof for "$\frac{2n}{3}$"

Extreme case for a heap is when the left subtree of root is completly full to the last level, and the right subtree is completely empty in the last level



assume: total # of nodes in the tree = n

let total # of nodes in the right subtree = m

$\Rightarrow$ Size of left subtree $= 2m + 1$

Therefore: $n = 1 + (2m+1) + m$

$= 3m + 2$

$\Rightarrow m = \frac{(n-2)}{3}$

$\Rightarrow$ Size of left subtree $= 2m+1 = \frac{2(n-2)}{3} + 1 = \frac{(2n-1)}{3}$

$\Rightarrow$ Size of lef subtree $< 2n/3$

Building a max-heap:

Build-Max-Heap converts an array $A[1:n]$ into a max-heap by calling Max-Heapify in a bottom-up manner.

→ Observe: $A[\lfloor n/2 \rfloor + 1 : n]$ are all leaves and 1-element max-heaps

---

Build-Max-Heap (A, n)

A.heap-size = n

for $i = \lfloor n/2 \rfloor$ down to 1

  Max-Heapify (A, i)

Complexity: $T(n) = \frac{n}{2} \cdot O(\lg n) = O(n \lg n)$

→ loose upper bound!

Better bound: $O(n)$!

Heapsort algorithm :

Given input array $A[1:n]$

1. Call Build-Max-Heap to build a max-heap
2. Move the max element to the end by exchanging $A[1]$ with $A[n]$
3. reduce heap size by 1
4. call max-heapify at the root to fix

Heapsort(A, n)

Build-Max-Heap(A, n) ⟶ $O(n)$

for $i = n$ down to 2 — $(n-1)$ $(\lg n)$

- Exchange $A[1]$ with $A[i]$
- $A.heapsize = A.heapsize - 1$
- Max-Heapify(A, 1)

Complexity : $O(n) + (n-1)\lg(n)$

$$T(n) = O(n \lg n)$$