CS5101: Design and Analysis of Algorithms

Sorting

Arijit Mondal

Dept of CSE

arijit@iitp.ac.in

https://www.iitp.ac.in/~arijit/

Tournament sort

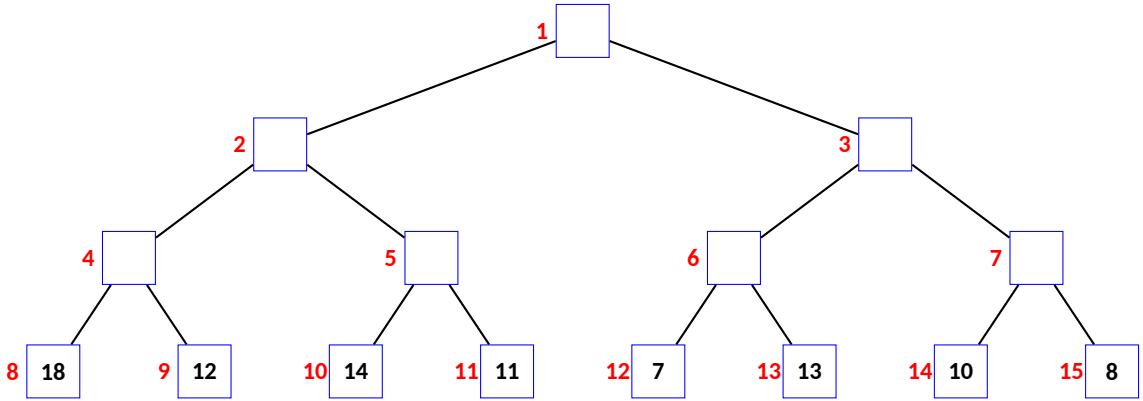
- We have seen how to find max and 2nd-max
- Can that be extended to find 3rd-max?

1	2	3	4	5	6	7	8
18	12	14	11	7	13	10	8

Tournament sort

- We have seen how to find max and 2nd-max
- Can that be extended to find 3rd-max?

1	2	3	4	5	6	7	8
18	12	14	11	7	13	10	8



Sorting-1

- sortS(S, n)
  1. if n=1 then return S_1
  2. $x = \max(S)$; $S' = \{S\} - x$;
  3. $S'' =$ sortS$(S', n-1)$
  4. return $\{S''\} * \{x\}$

Sorting-2

- sortI(S, n)
  1. if n=1 then return s_1
  2. Split S as follows: $S' = \{S\} - s_n$;
  3. $S'' =$ sortI$(S', n-1)$
  4. Insert s_n in S'' at appropriate location to obtain S'''
  5. return S'''

Sorting-3

- sortM(S, n)
  1. if n=1 then return S_1
  2. Split S into two disjoint sets S_1, S_2
  3. $S'_1 =$ sortM(S_1, n_1)
  4. $S'_2 =$ sortM(S_2, n_2)
  5. return merge(S_1, S_2)

Sorting-3

- sortM(S, n)
  1. if n=1 then return S_1
  2. Split S into two disjoint sets S_1, S_2
  3. $S'_1 = \text{sortM}(S_1, n_1)$
  4. $S'_2 = \text{sortM}(S_2, n_2)$
  5. return merge(S_1, S_2)
- merge($S_1[1 \ldots n_1], S_2[1 \ldots n_2]$)
  1. if $n_1 = 0$ return S_2
  2. if $n_2 = 0$ return S_1
  3. if $S_1[1] \leq S_2[1]$
  4. return merge($S_1[2 \ldots n_1], S_2[1 \ldots n_2]$)
  5. else
  6. return merge($S_1[1 \ldots n_1], S_2[2 \ldots n_2]$)

Sorting-4

- sortQ(S, n)
  1. if n=1 then return S_1
  2. Choose a pivot $p \in S$
  3. Split S in such a way that $S_1 = \{y | y \leq p\}$ and $S_2 = \{y | y > p\}$
  4. $S'_1 =$ sortQ(S_1, l)
  5. $S'_2 =$ sortQ$(S_2, n - l - 1)$
  6. return $\{S'_1\} * \{p\} * \{S'_2\}$

Sorting-4

- sortQ(S, n)
  1. if n=1 then return S_1
  2. Choose a pivot $p \in S$
  3. Split S in such a way that $S_1 = \{y | y \leq p\}$ and $S_2 = \{y | y > p\}$
  4. $S'_1 = \text{sortQ}(S_1, l)$
  5. $S'_2 = \text{sortQ}(S_2, n - l - 1)$
  6. return $\{S'_1\} * \{p\} * \{S'_2\}$
- Partition(S, p)
  1. swap $s_p \leftrightarrow s_n$
  2. $l = 0$
  3. for i in 1 to $(n - 1)$
  4. if $s_i < s_n$ then $\{\ l = l + 1$; swap $s_l \leftrightarrow s_i\ \}$
  3. swap $s_n \leftrightarrow s_{l+1}$
  6. return $l + 1$

Average case analysis-1

Lower bound for sorting

- Comparison based sorting has lower bound $\Omega(n \lg n)$

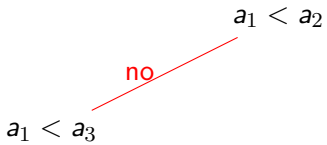
Lower bound for sorting

- Comparison based sorting has lower bound $\Omega(n \lg n)$

$$a_1 < a_2$$

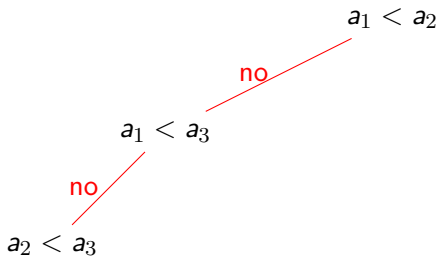
Lower bound for sorting

- Comparison based sorting has lower bound $\Omega(n \lg n)$



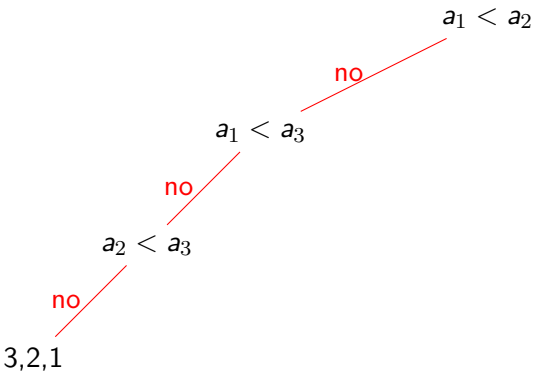
Lower bound for sorting

- Comparison based sorting has lower bound $\Omega(n \lg n)$



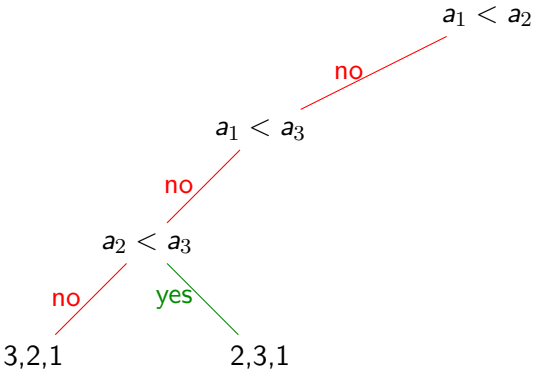
Lower bound for sorting

- Comparison based sorting has lower bound $\Omega(n \lg n)$



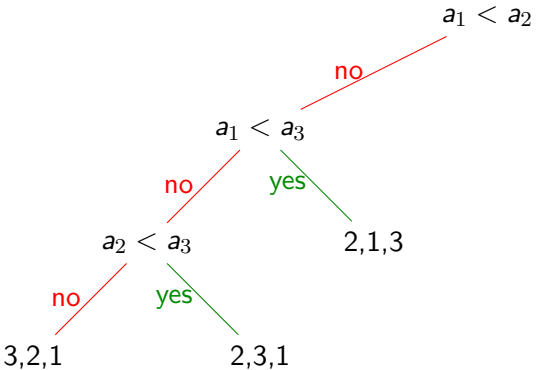
Lower bound for sorting

- Comparison based sorting has lower bound $\Omega(n \lg n)$



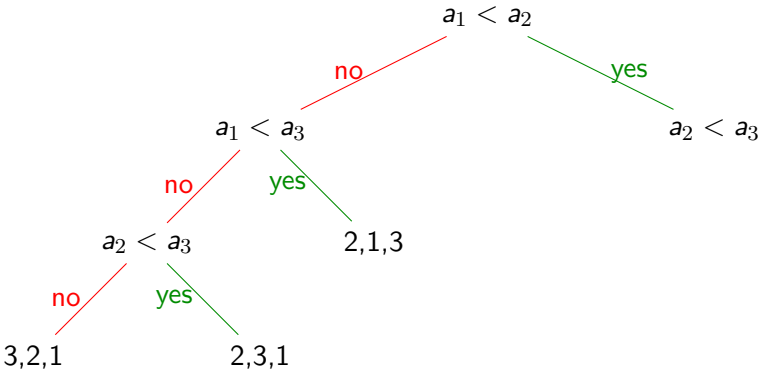
Lower bound for sorting

- Comparison based sorting has lower bound $\Omega(n \lg n)$



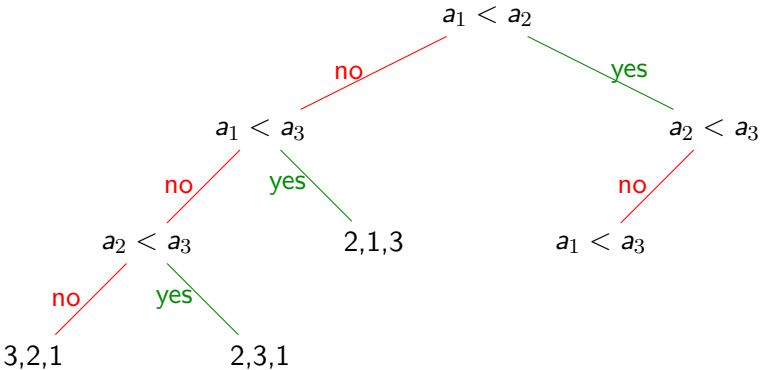
Lower bound for sorting

- Comparison based sorting has lower bound $\Omega(n \lg n)$



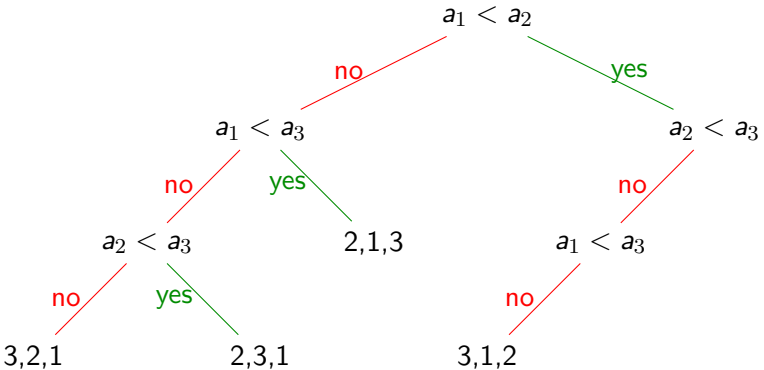
Lower bound for sorting

- Comparison based sorting has lower bound $\Omega(n \lg n)$



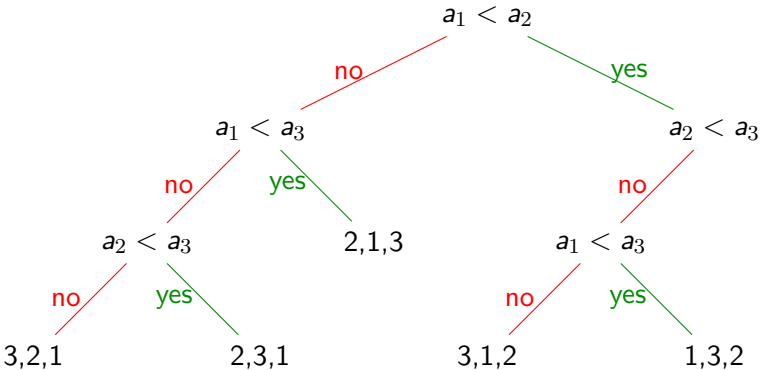
Lower bound for sorting

- Comparison based sorting has lower bound $\Omega(n \lg n)$



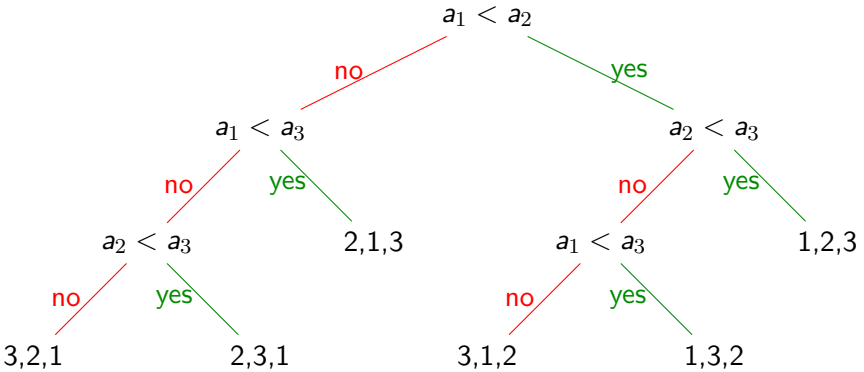
Lower bound for sorting

- Comparison based sorting has lower bound $\Omega(n \lg n)$



Lower bound for sorting

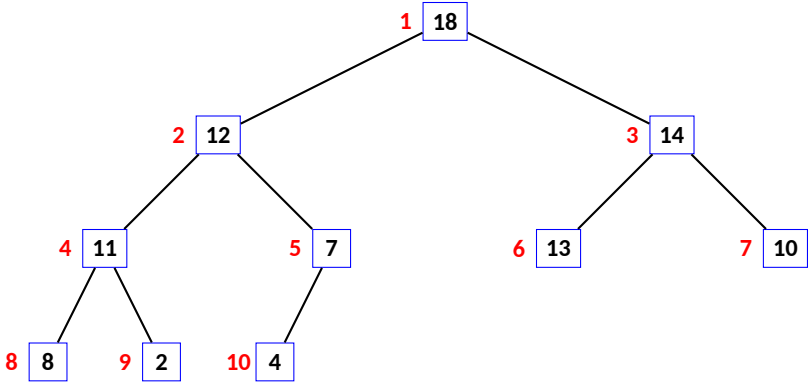
- Comparison based sorting has lower bound $\Omega(n \lg n)$



Heaps-1

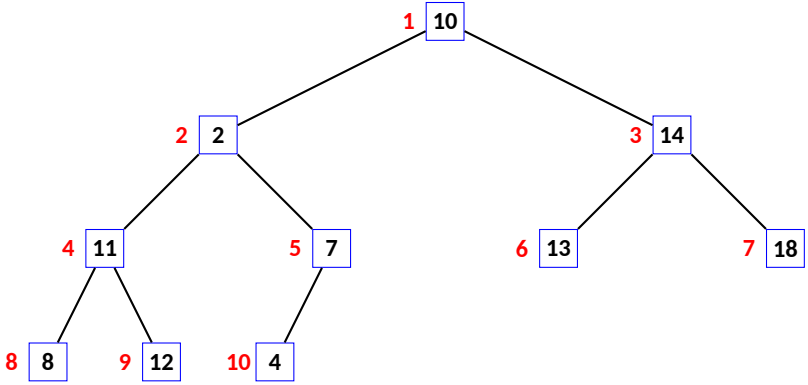
- A heap is a binary tree whose keys satisfy the following (max) heap property: The key of every node is greater than or equal to the key of any of its children.
- The tree is nearly complete binary tree
- Heaps are useful in implementing a priority queue. It maintains a partial order among the elements
- Typical operations - insert, extract-max, heapify
- Two scenario may be looked into–
  - Online: always maintain a heap as we receive the inputs
  - Offline: all numbers are available in an array, we need to create a heap using those

Heaps-2

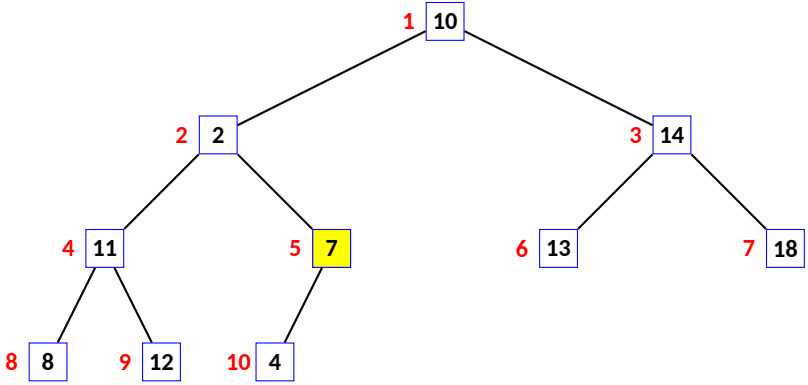


1	2	3	4	5	6	7	8	9	10
18	12	14	11	7	13	10	8	2	4

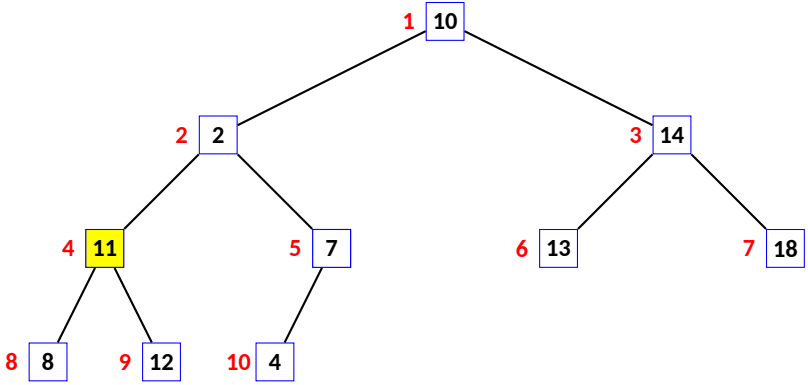
Heapify



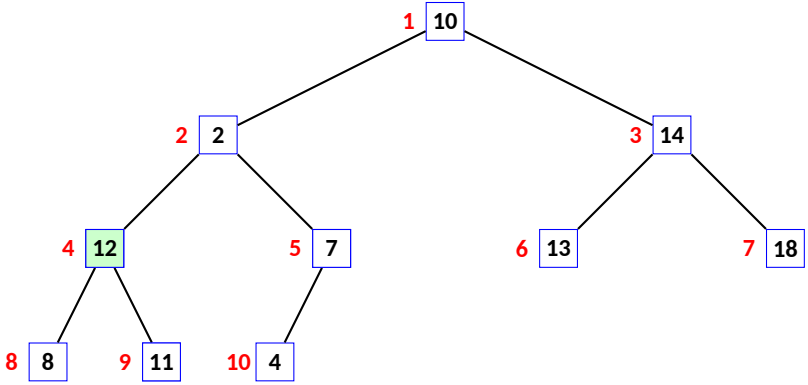
Heapify



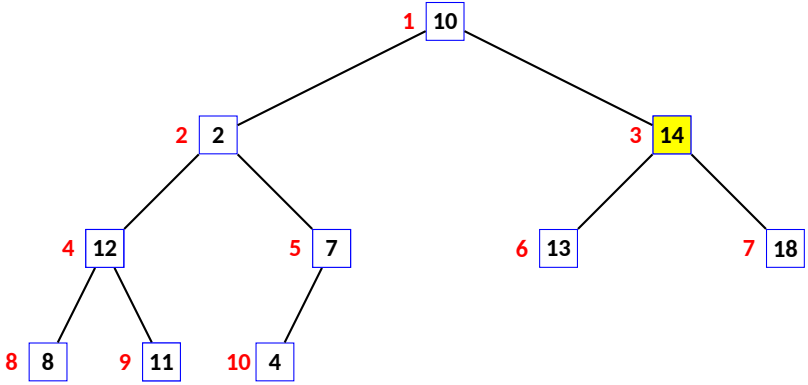
Heapify



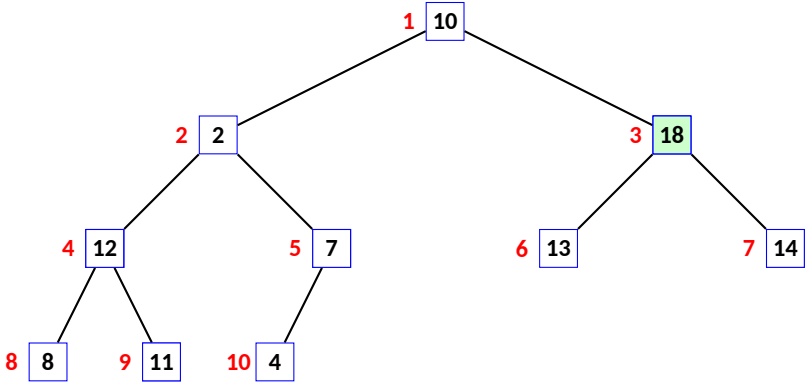
Heapify



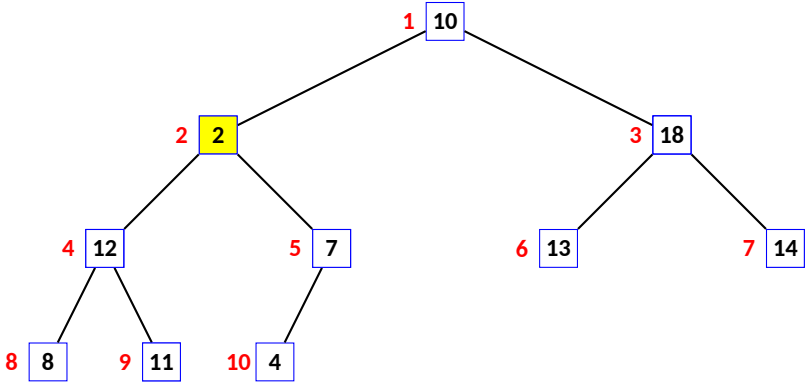
Heapify



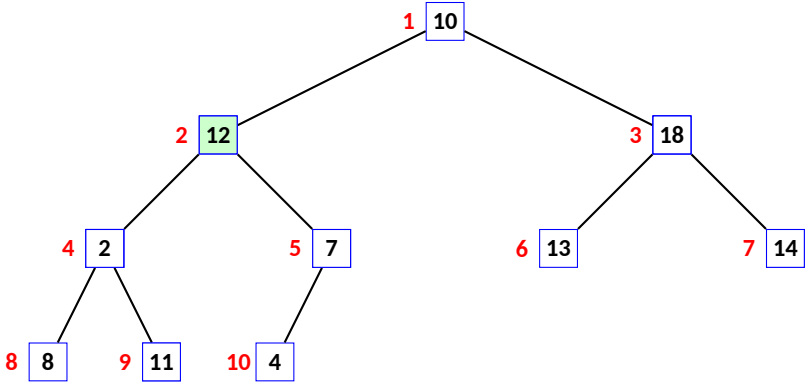
Heapify



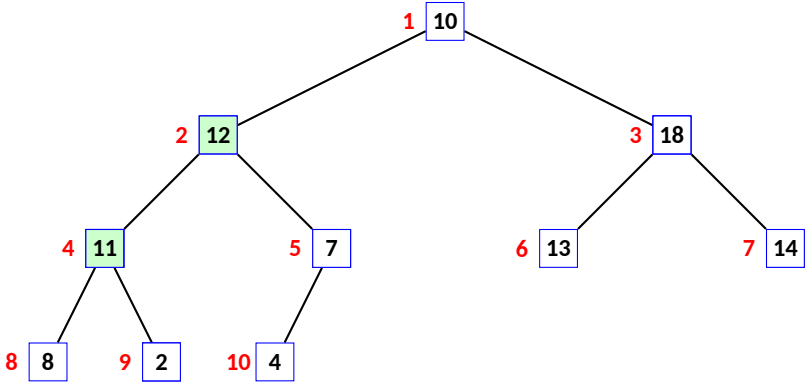
Heapify



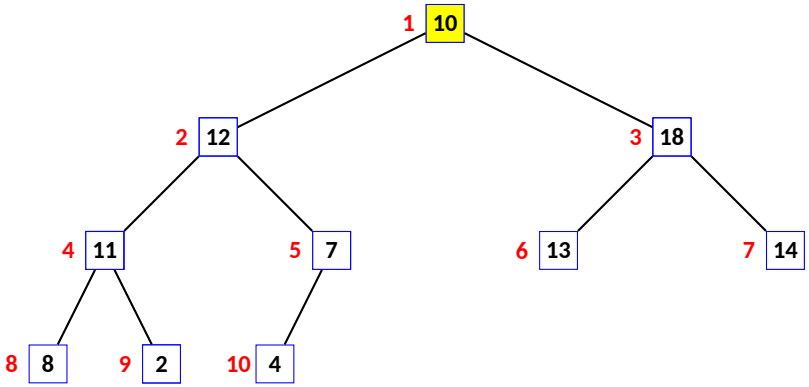
Heapify



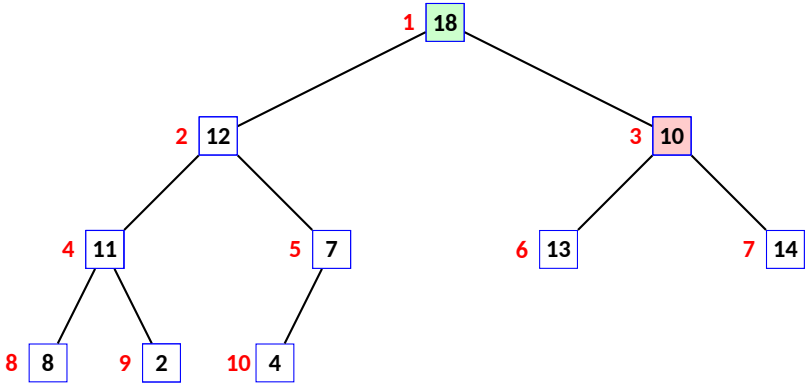
Heapify



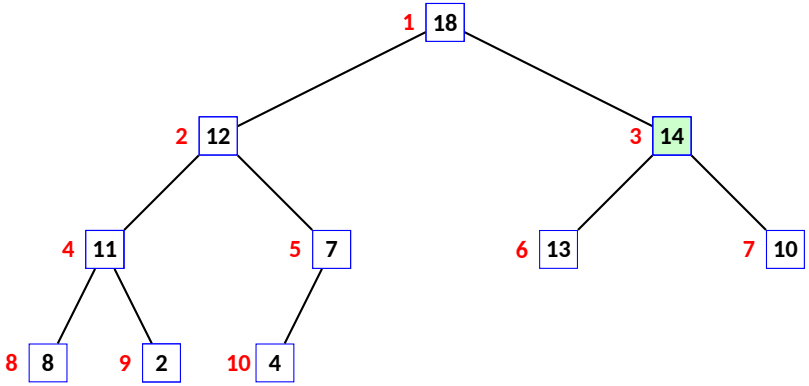
Heapify



Heapify



Heapify



Max-Heapify

- Max-Heapify(A, i)
  1. $l =$ left(i), $r =$ right(i)
  2. if $l \leq A$.size and $A[l] > A[i]$ then largest $= l$ else largest $= i$
  3. if $r \leq A$.size and $A[r] > A[largest]$ then largest $= r$
  4. if largest $\neq$ i
  5. exchange $A[i]$ with $A[largest]$
  6. Max-Heapify($A, largest$)
- Build-Max-Heap(A, n)
  1. A.size $= n$
  2. for $i = \lceil n/2 \rceil$ downto 1
  3. Max-Heapify(A, i)

Build heap

Build heap

- Online: build time $O(n \lg n)$
- Offline: build time $O(n)$

Exercise-1

- **Input**: An array A of distinct integers. **Output**: The number of inversions of A — the number of pairs (i, j) of array indices with $i < j$ and $A[i] > A[j]$
- Example: Consider the permutation — $5, 9, 1, 8, 2, 6, 4, 7, 3$. The number for inversions for 1-9 are as follows: $2, 3, 6, 4, 0, 2, 2, 1, 0$ (aka. inversion table). Hence total number of inversion is 20.
- Given an inversion table, is it possible to find original permutation?
- Given an integer array, count the total number of inversion in $O(n \lg n)$ time
- Can *inversion* count be used for collaborative filtering?

Exercise-2

- Can **four** elements be sorted using 5 comparison?
- Can **five** elements be sorted using 7 comparison?

Thank you!