Faculty of Information Technology, Monash University

FIT2004, S2/2016

Week 3: Sorting Algorithms

Lecturer: Muhammad Aamir Cheema

ACKNOWLEDGMENTS

The slides are based on the material developed by Arun Konagurthu and Lloyd Allison.

Overview

- In the previous week's lectures, we focused on precise reasoning to solve problems using loop invariants.
- We have also analysed them, proved them etc.
- In this lecture, we will discuss and analyze more efficient O(N log N) time sorting algorithms.
- Our discussion will cover
  - Stability of sorting and In-place algorithms
  - Heap data-structure and Heap sort
  - Merge sort
  - Quicksort

Recommended reading

- Priority Queue and Heap data structure reference: http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Priority-Q/
- Merge sort: http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Sort/Merge/
- Dutch National Flag partitioning problem: http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Sort/Flag/
- Quick sort: http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Sort/Quick/
- Weiss, Data structures and Algorithm analysis in Java, Chapters 6 & 7

Stable sorting algorithms

A sorting algorithm is called stable if it maintains the relative ordering of elements that have equal keys.

Input is sorted by names

Input

Marks	**80**	**75**	**70**	**90**	**85**	**75**
Name	Alice	Bill	John	Geoff	Leo	Maria

Sort on Marks using a stable algorithm

Output

Marks	**70**	**75**	**75**	**80**	**85**	**90**
Name	John	Bill	Maria	Alice	Leo	Geoff

Note: Output is sorted on marks then names.
Unstable sorting cannot guarantee this (e.g., Maria may appear before Bill)
Selection sort is unstable. Insertion sort is stable!

Priority Queue

- Priority Queue is an Abstract Data Type usually implemented with a heap
- The operations of a [Priority Queue] are:
  - create an empty Priority Queue
  - insert an element having a certain priority to the Priority Queue
  - remove the element having the highest priority.

Do not confuse Priority Queue with ordinary Queue

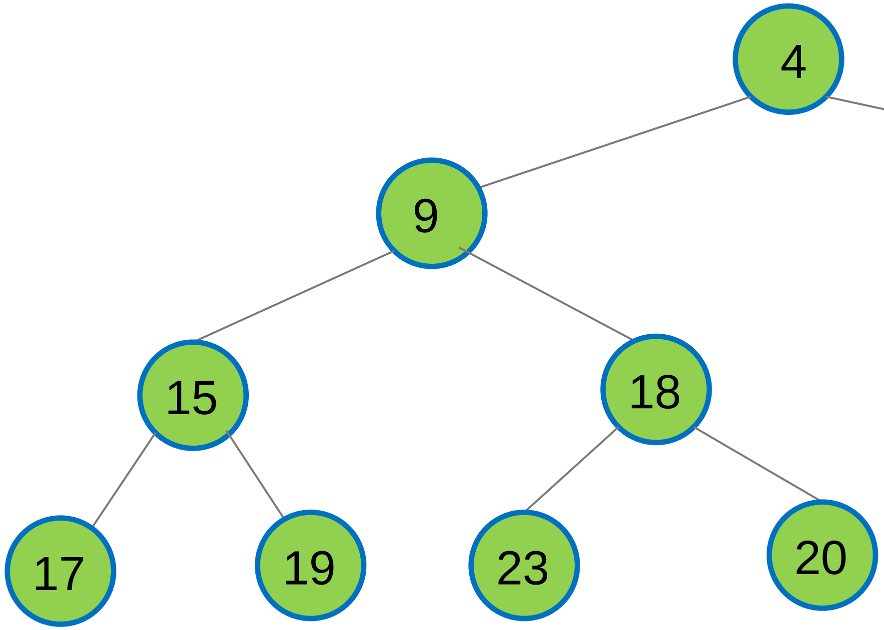
Heap data structure

- Heap is an implementation of a Priority Queue
- Heap is so common for Priority Queue implementation that, in context of Priority Queue, when the term Heap is used, it generally means an implementation of Priority Queue.
- In this unit, we may use Heap and Priority Queue interchangeably

Properties of Heap

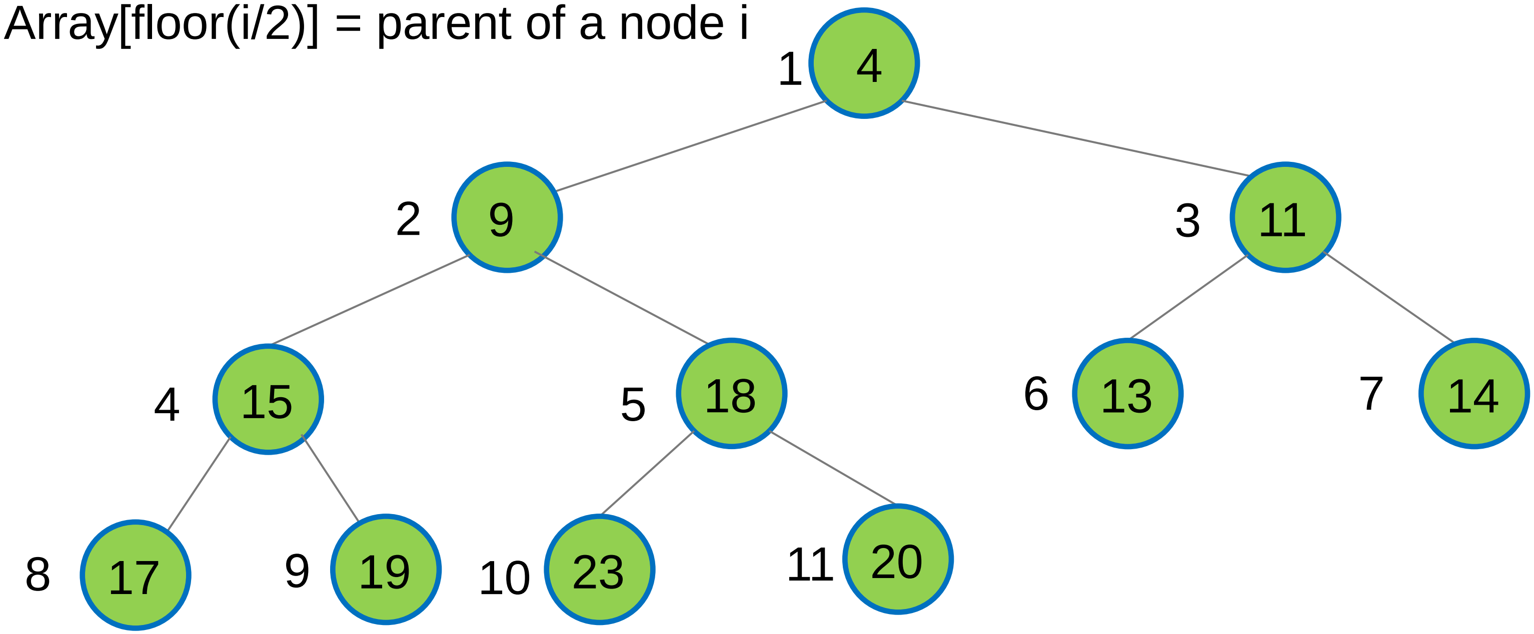
Heap is an implementation of the Priority Queue

- Heap is a balanced binary tree
- A parent is always smaller than or equal to its children (this implies that the root is the smallest element in the heap)



Heap can be represented as an array

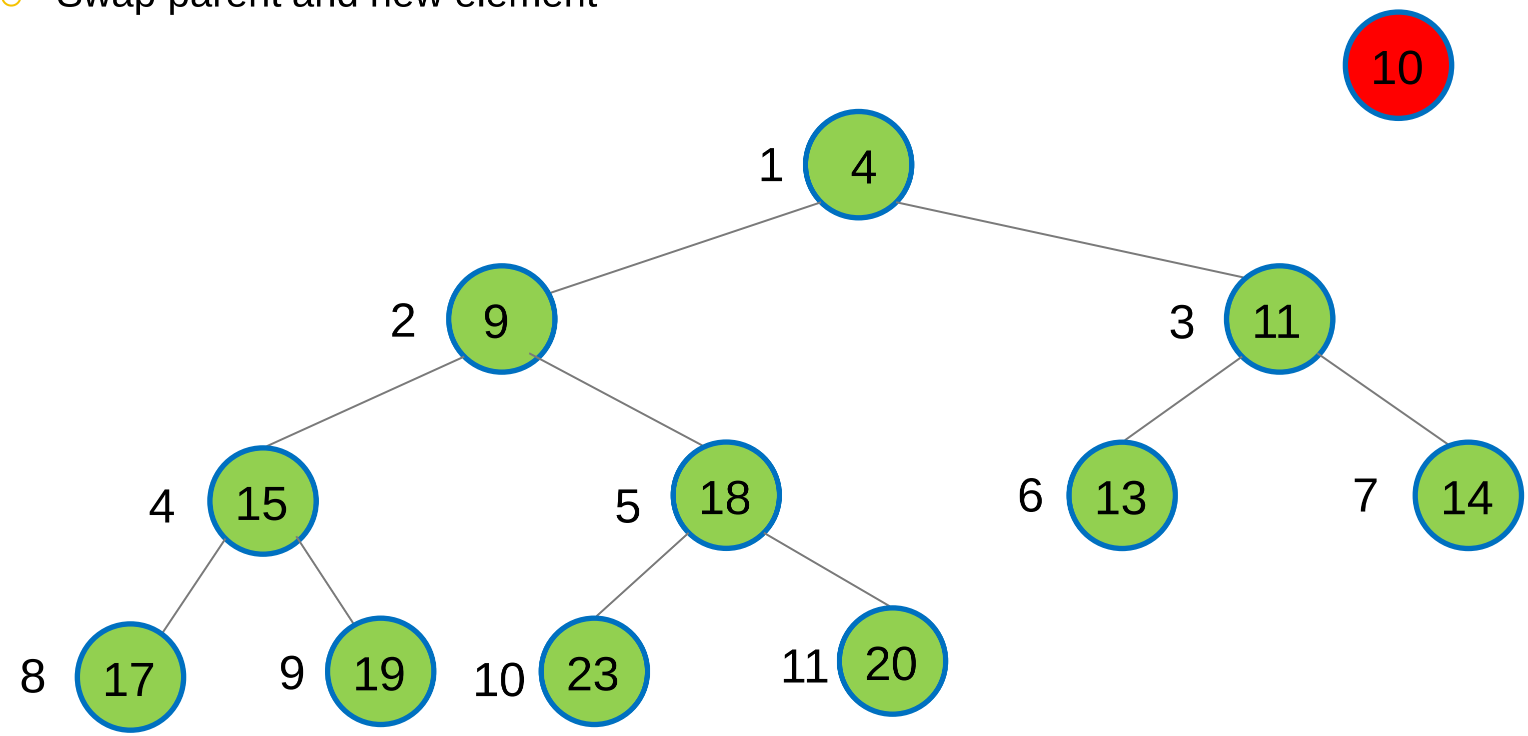
- Array[1] = root of the heap
- Array[i] = an arbitrary node i
- Array[2i] = left child of node i
- Array[2i + 1] = right child of node i
- Array[floor(i/2)] = parent of a node i



4	9	11	15	18	13	14	17	19	23	20
1	2	3	4	5	6	7	8	9	10	11

Insertion in Heap (up-Heap)

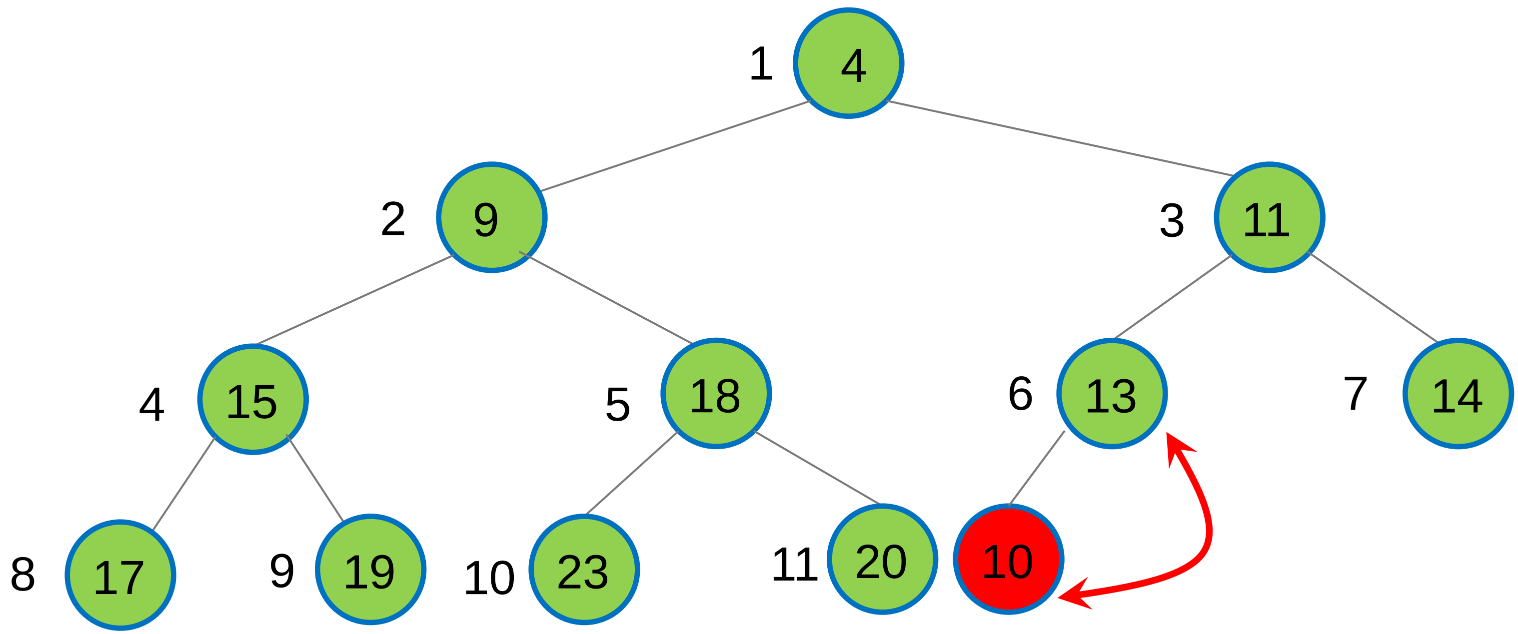
- Insert new element at Array[N+1]
- While parent(new) > new
  - Swap parent and new element



4	9	11	15	18	13	14	17	19	23	20	
1	2	3	4	5	6	7	8	9	10	11	12

Insertion in Heap (up-Heap)

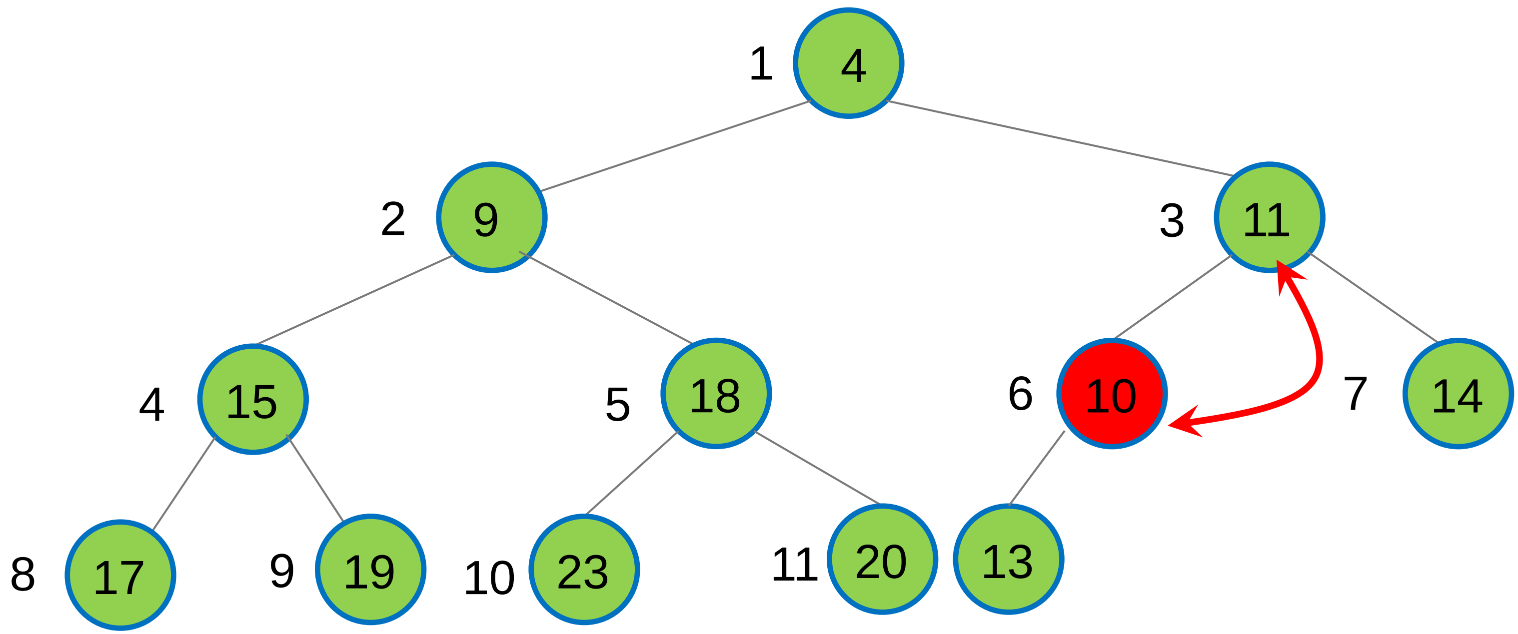
- Insert new element at Array[N+1]
- While parent(new) > new
    - Swap parent and new element



4	**9**	**11**	**15**	**18**	**13**	**14**	**17**	**19**	**23**	**20**	**10**
1	2	3	4	5	6	7	8	9	10	11	12

Insertion in Heap (up-Heap)

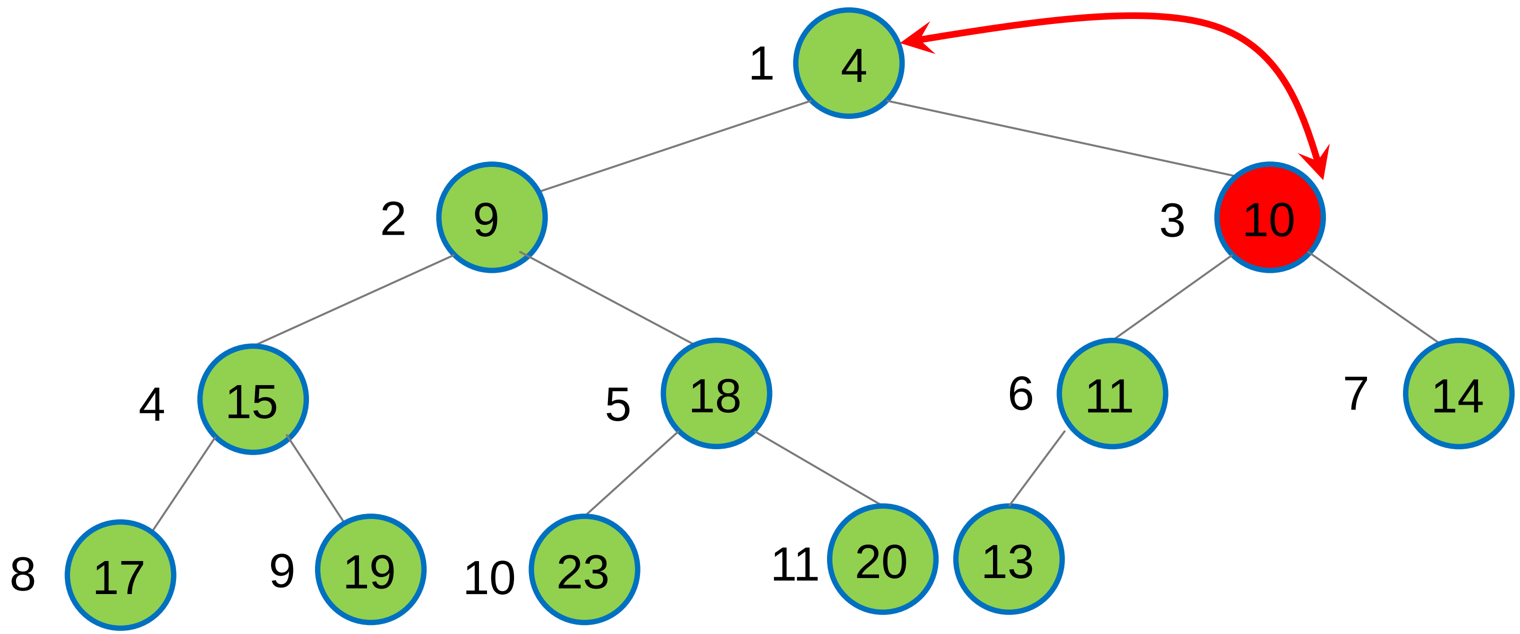
- Insert new element at Array[N+1]
- While parent(new) > new
  - Swap parent and new element



4	9	11	15	18	10	14	17	19	23	20	13
1	2	3	4	5	6	7	8	9	10	11	12

Insertion in Heap (up-Heap)

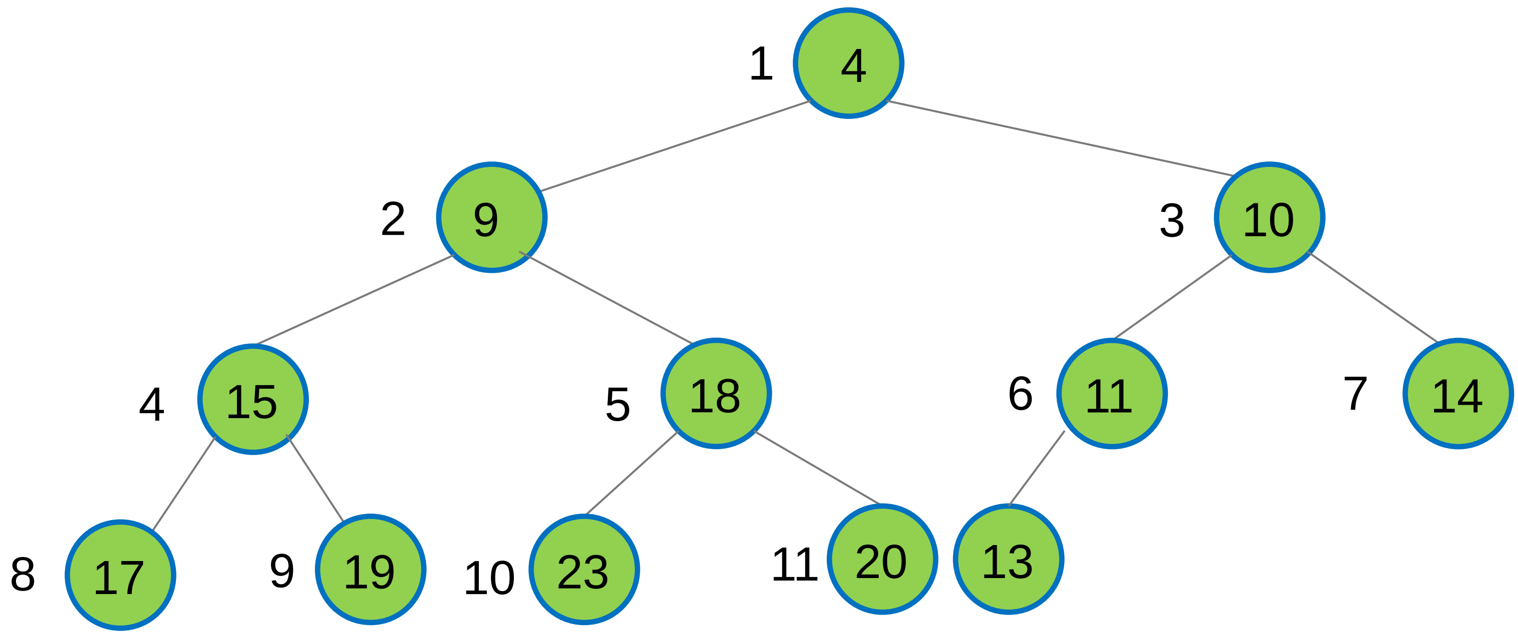
- Insert new element at Array[N+1]
- While parent(new) > new
  - Swap parent and new element



4	9	10	15	18	11	14	17	19	23	20	13
1	2	3	4	5	6	7	8	9	10	11	12

Insertion in Heap (up-Heap)

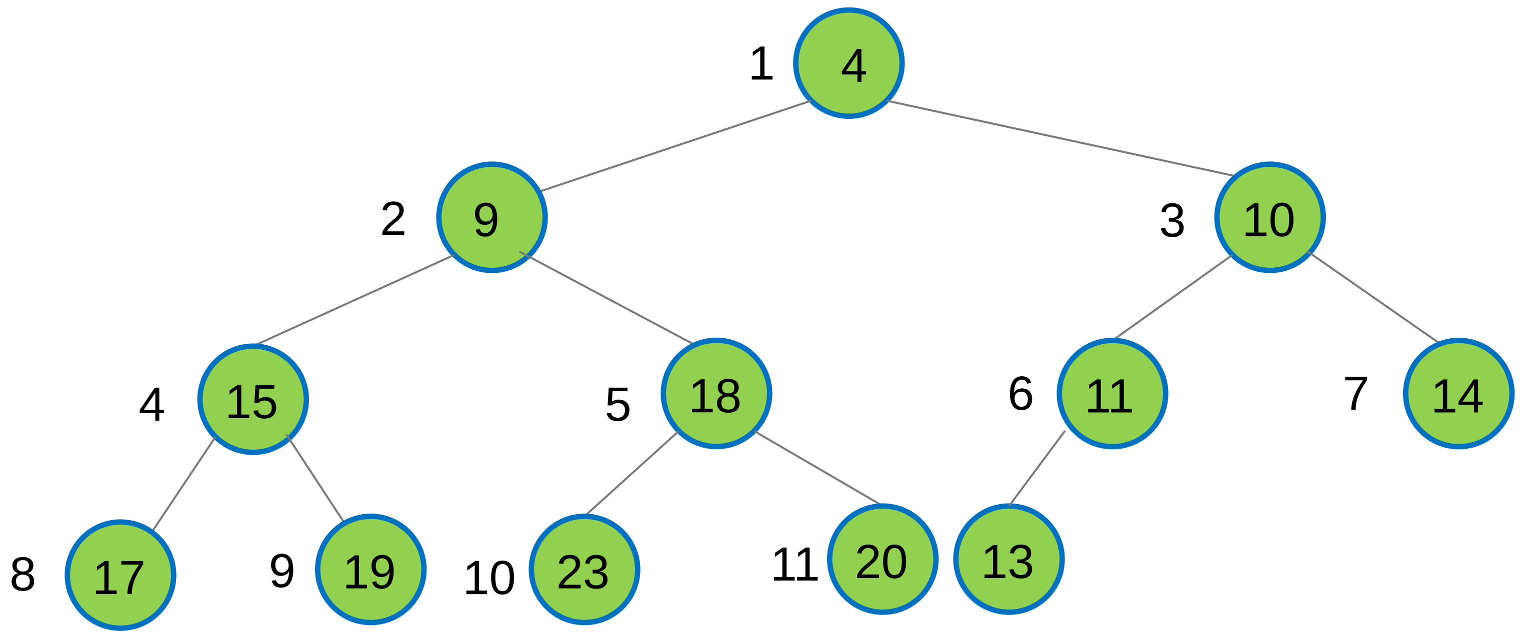
- Insert new element at Array[N+1]
- While parent(new) > new
  - Swap parent and new element



1	2	3	4	5	6	7	8	9	10	11	12
4	**9**	**10**	**15**	**18**	**11**	**14**	**17**	**19**	**23**	**20**	**13**

Insertion in Heap (up-Heap)

- Insert new element at Array[N+1]
- While parent(new) > new and new is not the root node
  - Swap parent and new element



4	**9**	**10**	**15**	**18**	**11**	**14**	**17**	**19**	**23**	**20**	**13**
1	2	3	4	5	6	7	8	9	10	11	12

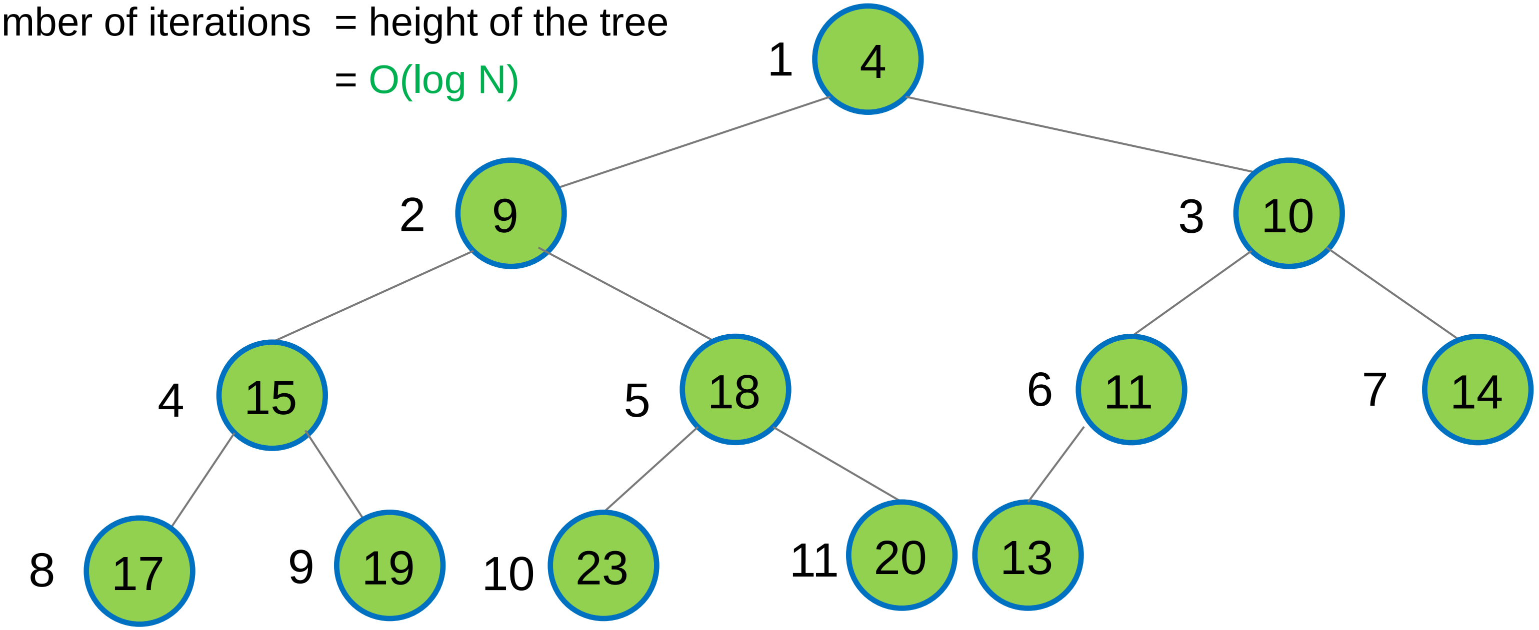
Complexity of up-heap

- Insert new element at Array[N+1]
- While parent(new) > new and new is not the root node
  - Swap parent and new element

Worst-case time complexity:

Number of iterations = height of the tree

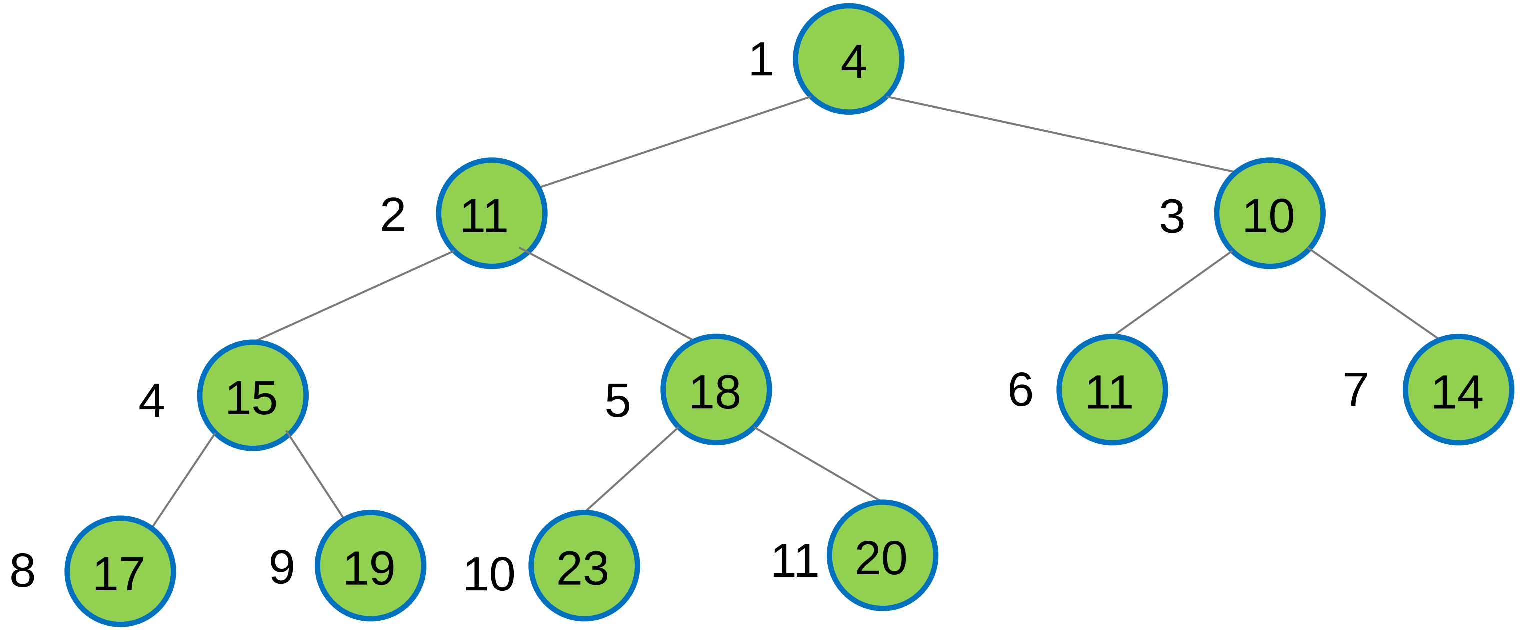
= O(log N)



4	**9**	**10**	**15**	**18**	**11**	**14**	**17**	**19**	**23**	**20**	**13**
1	2	3	4	5	6	7	8	9	10	11	12

Deletion from Heap (downHeap)

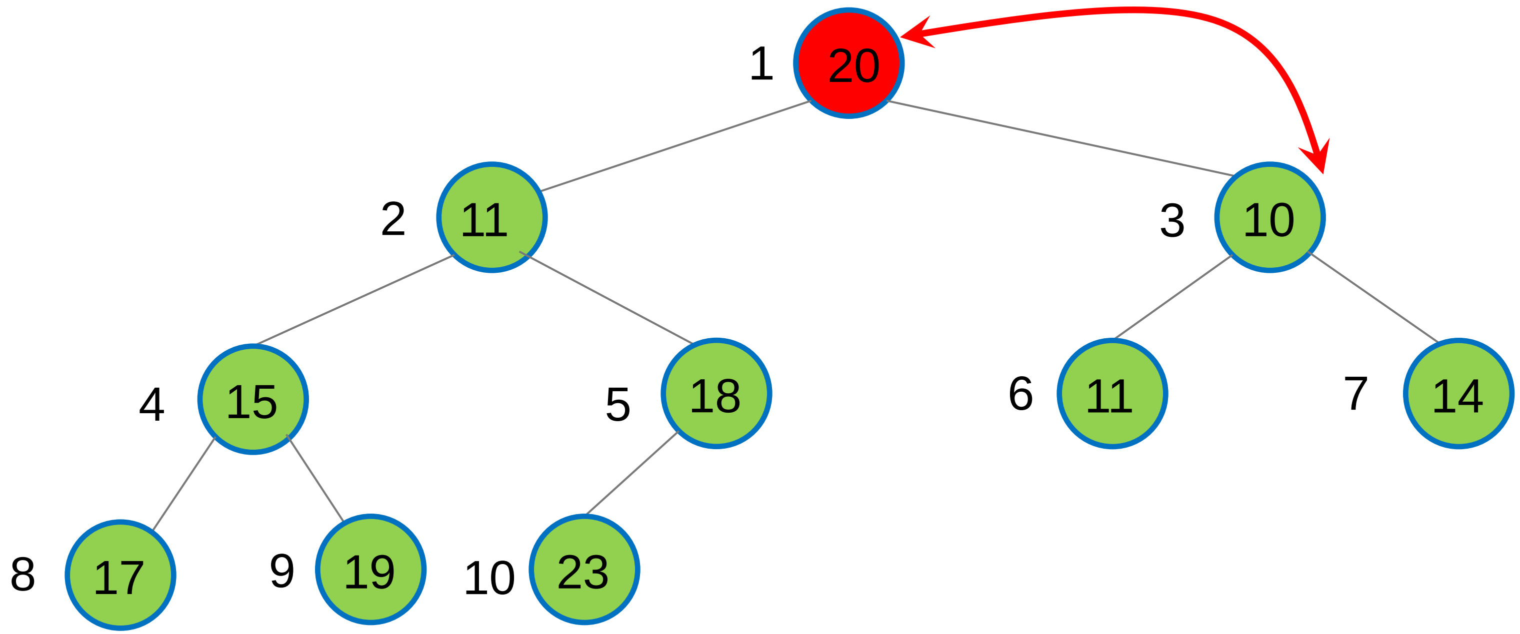
- Delete Array[1]
- Move Array[N] (called last) to Array[1]



1	2	3	4	5	6	7	8	9	10	11
4	**11**	**10**	**15**	**18**	**11**	**14**	**17**	**19**	**23**	**20**

Deletion from Heap (downHeap)

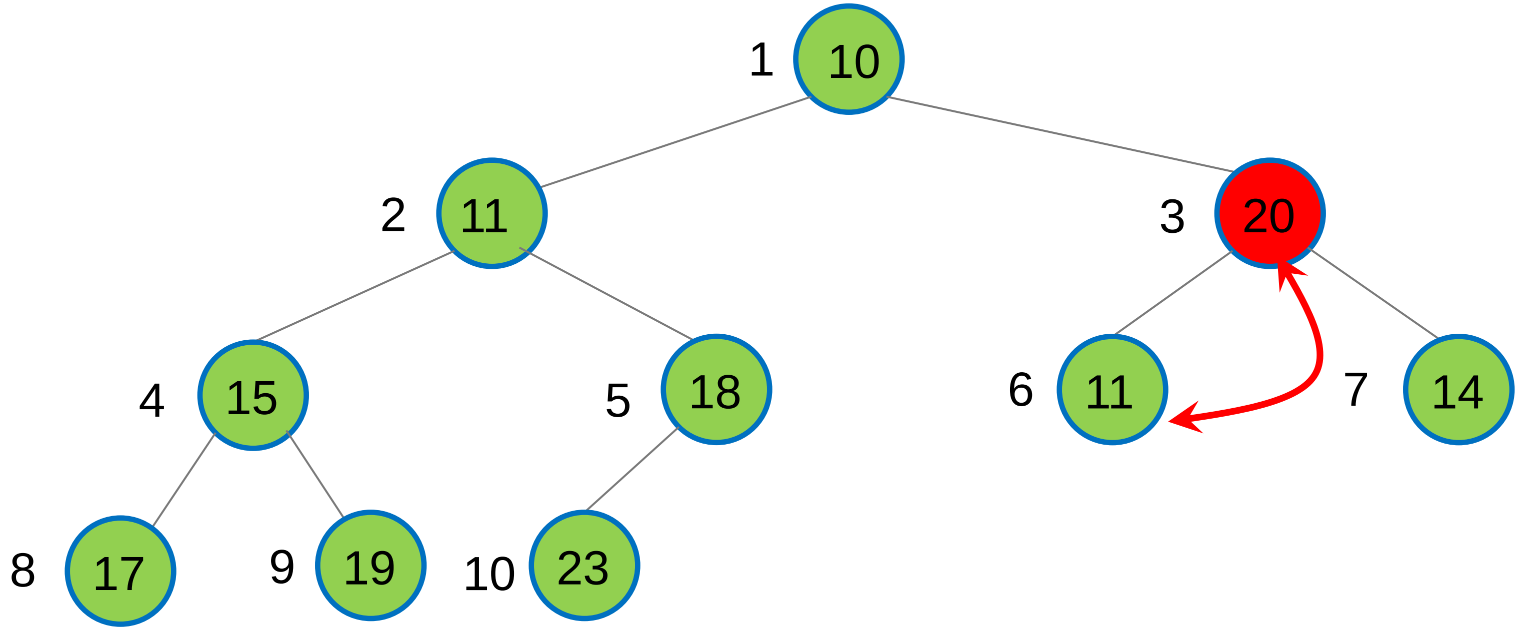
- Delete Array[1]
- Move Array[N] (called last) to Array[1]
- While leftchild(last) < last or rightchild(last) < last
  - Swap last with minimum(leftchild,rightchild)



20	11	10	15	18	11	14	17	19	23	
1	2	3	4	5	6	7	8	9	10	11

Deletion from Heap (downHeap)

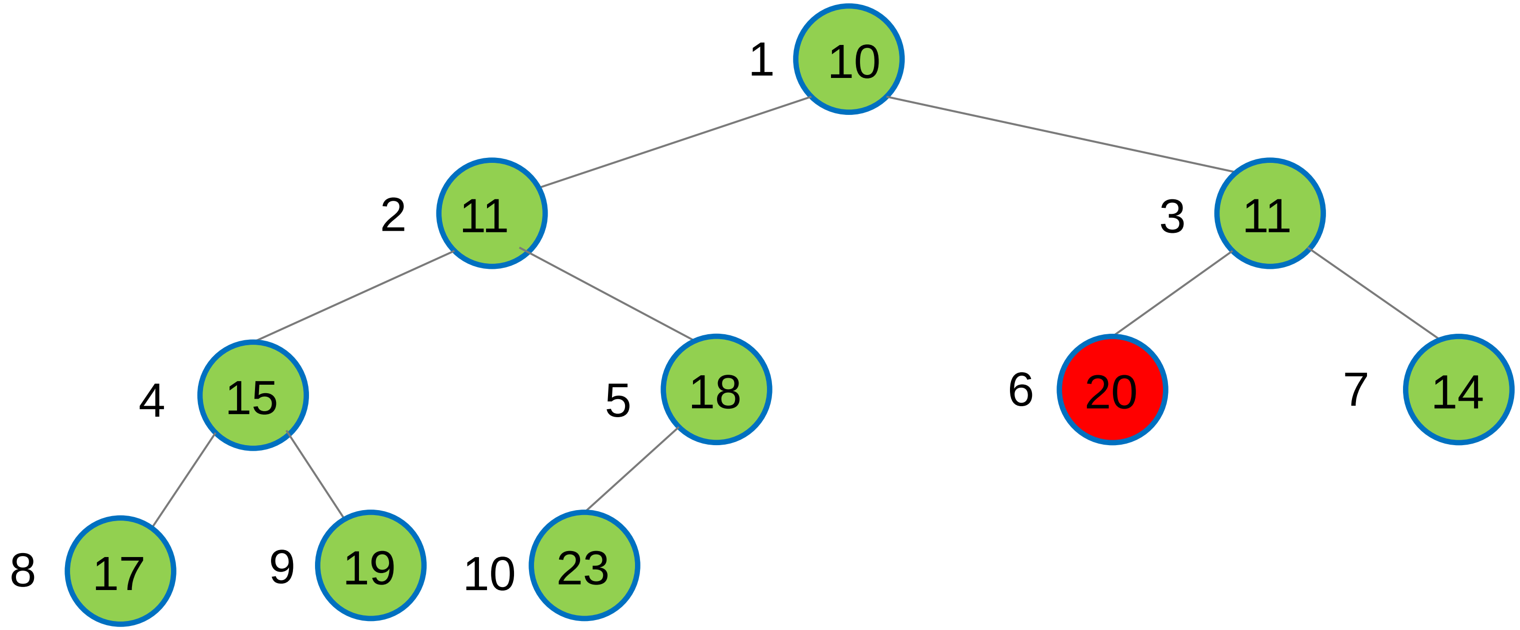
- Delete Array[1]
- Move Array[N] (called last) to Array[1]
- While leftchild(last) < last or rightchild(last) < last
  - Swap last with minimum(leftchild,rightchild)



10	11	20	15	18	11	14	17	19	23	
1	2	3	4	5	6	7	8	9	10	11

Deletion from Heap (downHeap)

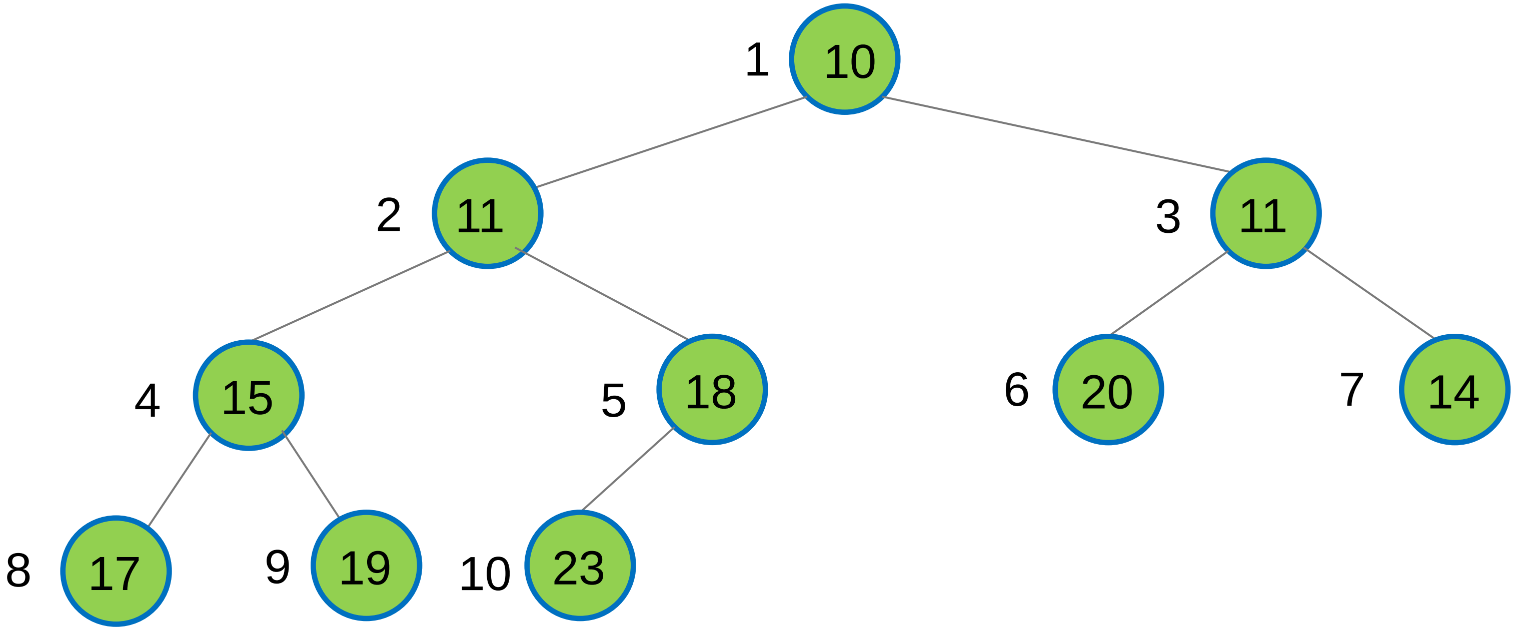
- Delete Array[1]
- Move Array[N] (called last) to Array[1]
- While leftchild(last) < last or rightchild(last) < last
  - Swap last with minimum(leftchild,rightchild)



10	11	11	15	18	20	14	17	19	23	
1	2	3	4	5	6	7	8	9	10	11

Deletion from Heap (downHeap)

- Delete Array[1]
- Move Array[N] (called last) to Array[1]
- While leftchild(last) < last or rightchild(last) < last
  - Swap last with minimum(leftchild,rightchild)

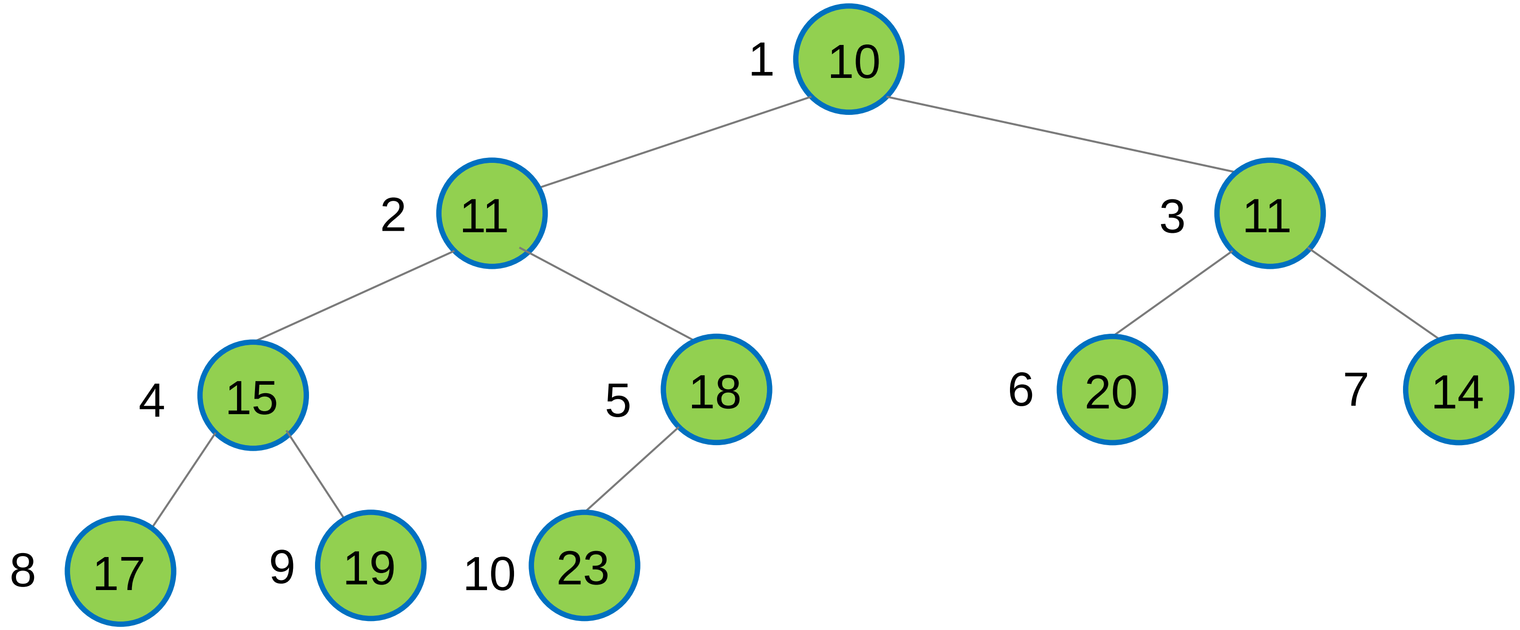


10	11	11	15	18	20	14	17	19	23	
1	2	3	4	5	6	7	8	9	10	11

Complexity of downHeap?

- Delete Array[1]
- Move Array[N] (called last) to Array[1]
- While leftchild(last) < last or rightchild(last) < last
  - Swap last with minimum(leftchild,rightchild)

O(log N)



10	11	11	15	18	20	14	17	19	23	
1	2	3	4	5	6	7	8	9	10	11

Heapify

The heapify procedure builds a heap from an unsorted array

A straightforward approach:

- Initialize an empty heap
- **for** i **=** 1**;** i **<=** N**;** i**++**
- insert array**[** i **]** **in** heap

Time Complexity

log(1) + log(2) + log(3) + … + log(N)

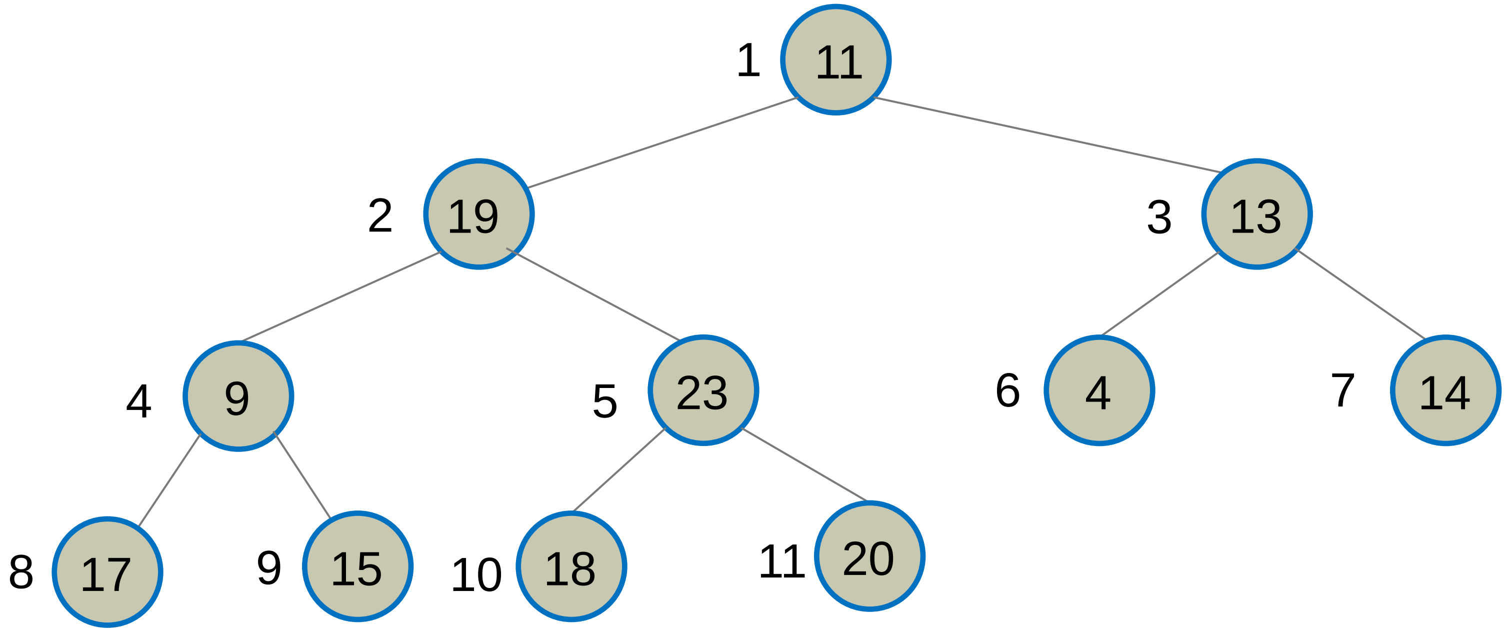
= log(1x2x3x…xN) = log(N!)

= O(N log N)

11	**19**	**13**	**9**	**23**	**4**	**14**	**17**	**15**	**18**	**20**
1	2	3	4	5	6	7	8	9	10	11

Heapify in O(N)

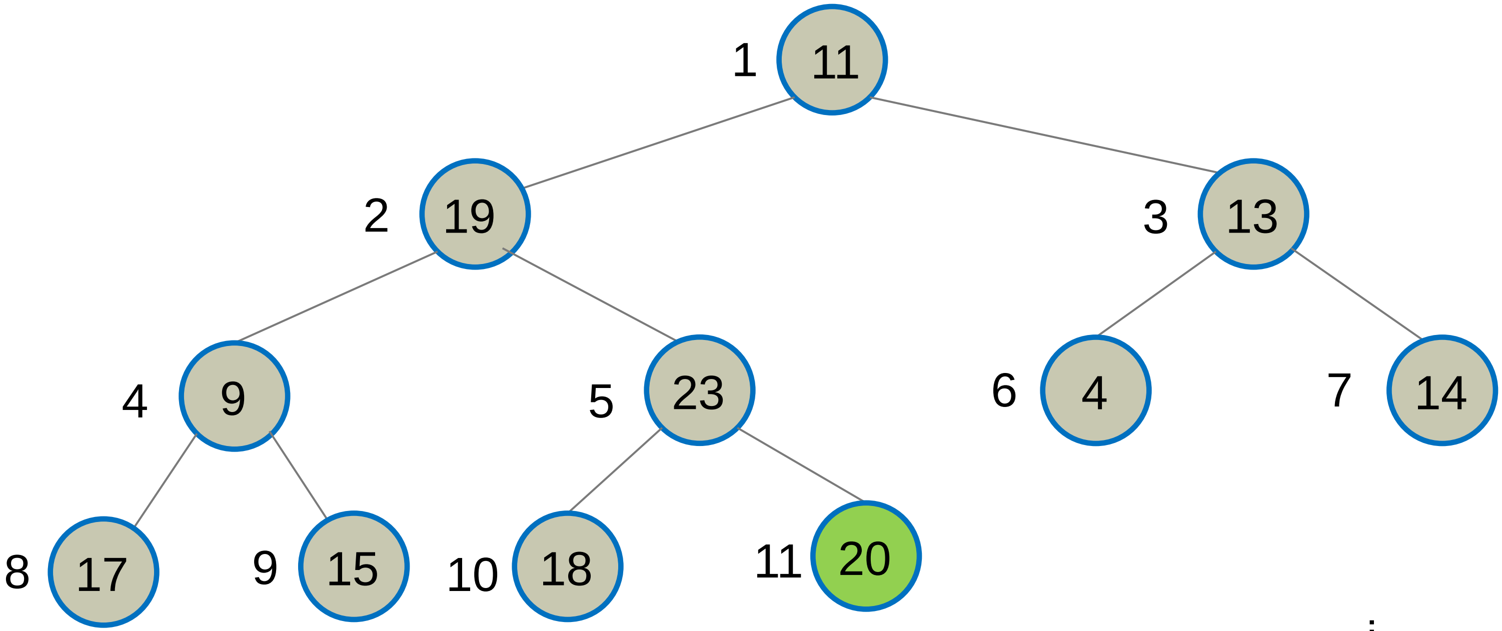
- **for** i **=** N**;** i **>=** 1**;** i**--**
-   downHeap array**[**i**]** **in** the heap rooted at array**[**i**]**



11	19	13	9	23	4	14	17	15	18	20
1	2	3	4	5	6	7	8	9	10	11

Heapify in O(N)

- **for** i **=** N**;** i **>=** 1**;** i**--**
- downHeap array**[**i**]** **in** the heap rooted at array**[**i**]**

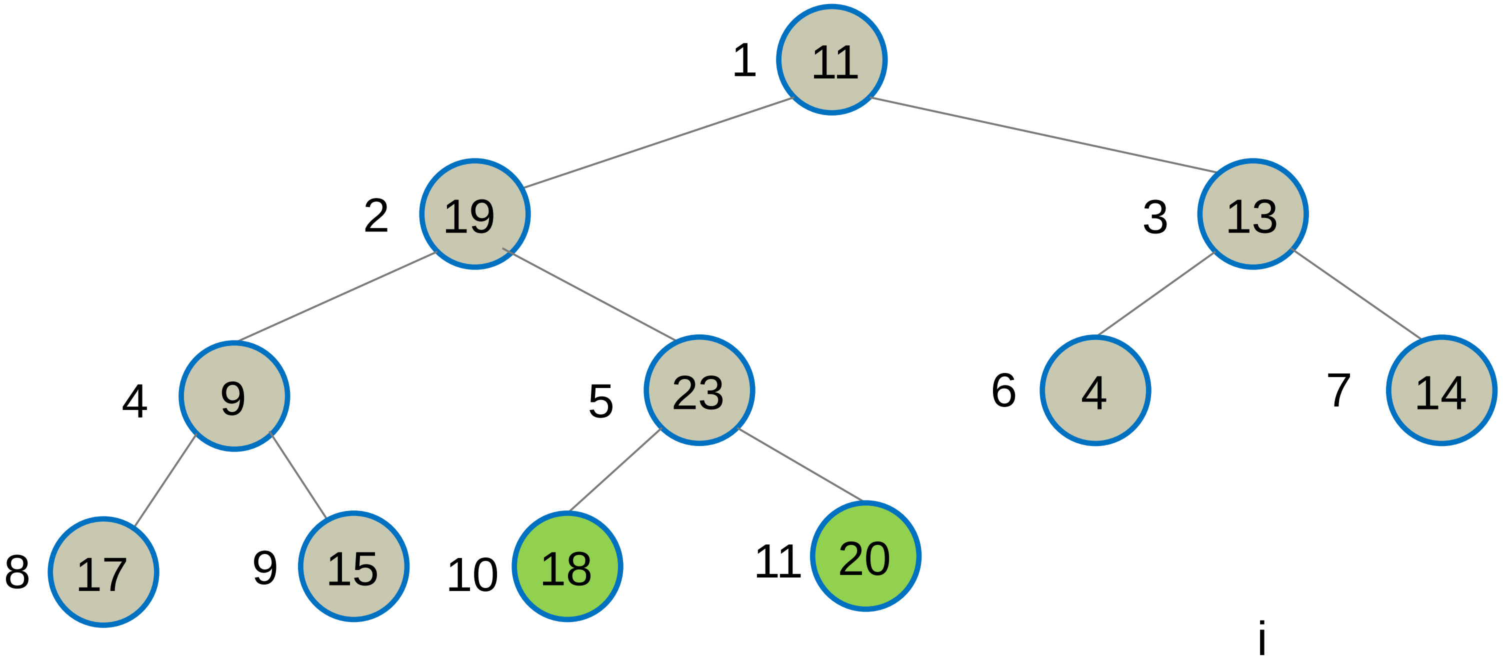


i

11	**19**	**13**	**9**	**23**	**4**	**14**	**17**	**15**	**18**	**20**
1	2	3	4	5	6	7	8	9	10	11

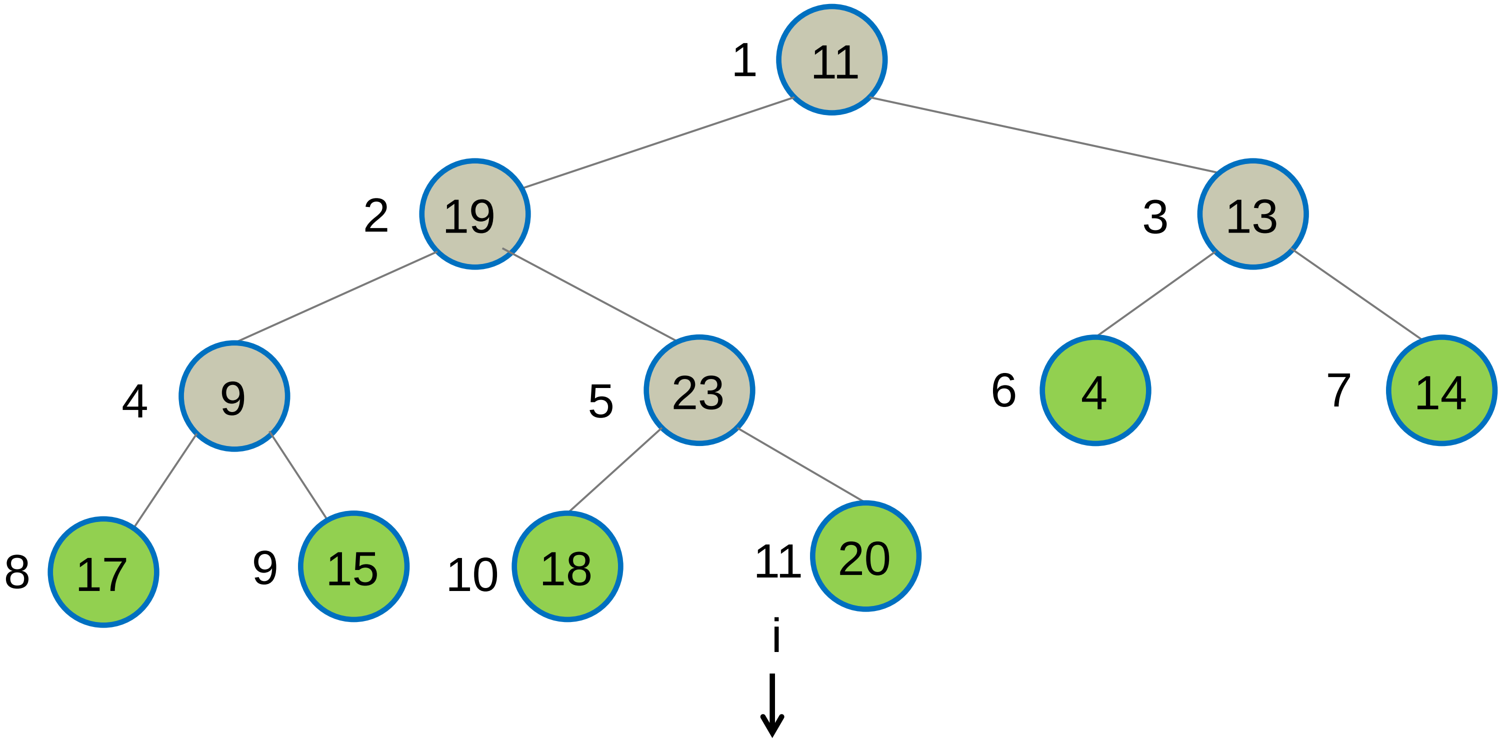
Heapify in O(N)

- **for** i **=** N**;** i **>=** 1**;** i**--**
- downHeap array**[**i**]** **in** the heap rooted at array**[**i**]**



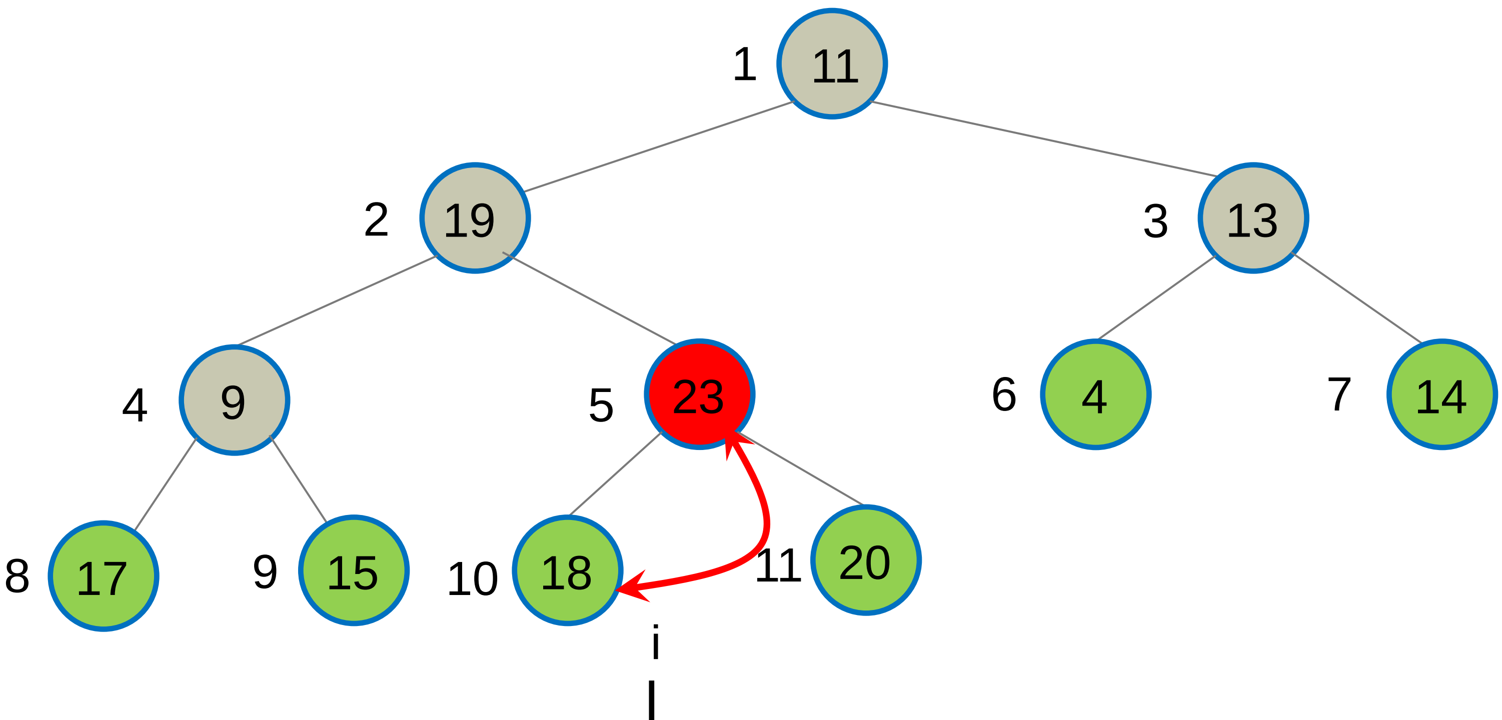
11	**19**	**13**	**9**	**23**	**4**	**14**	**17**	**15**	**18**	**20**
1	2	3	4	5	6	7	8	9	10	11

Heapify in O(N)



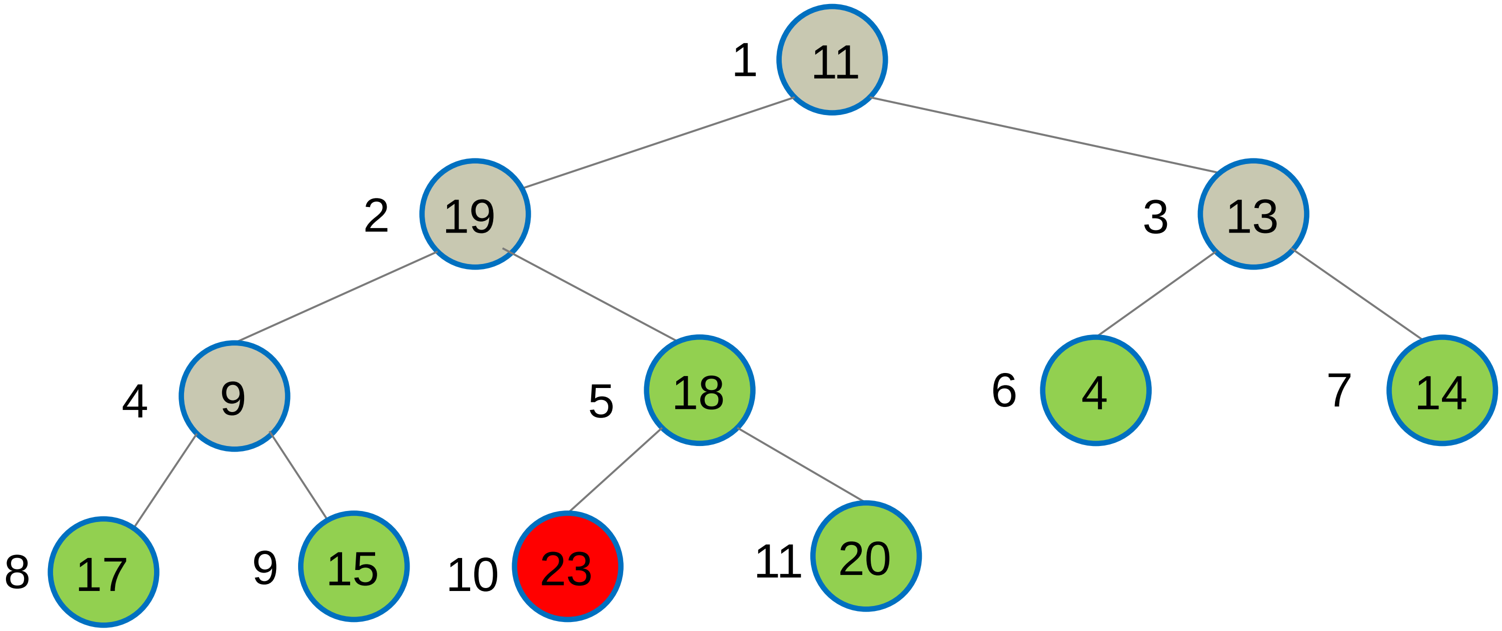
11	19	13	9	23	4	14	17	15	18	20
1	2	3	4	5	6	7	8	9	10	11

Heapify in O(N)



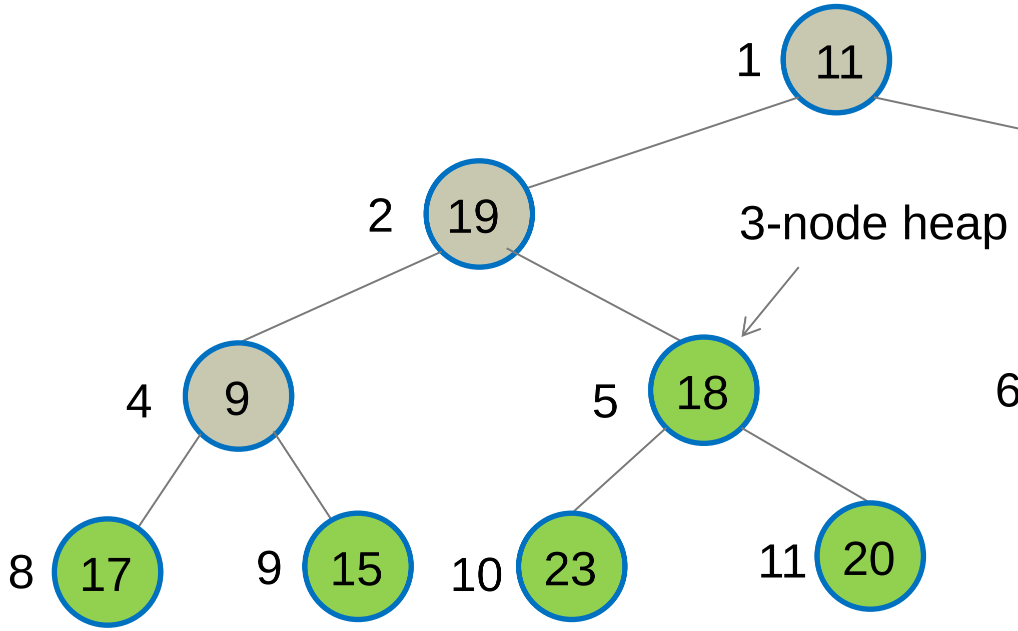
11	19	13	9	23	4	14	17	15	18	20
1	2	3	4	5	6	7	8	9	10	11

Heapify in O(N)



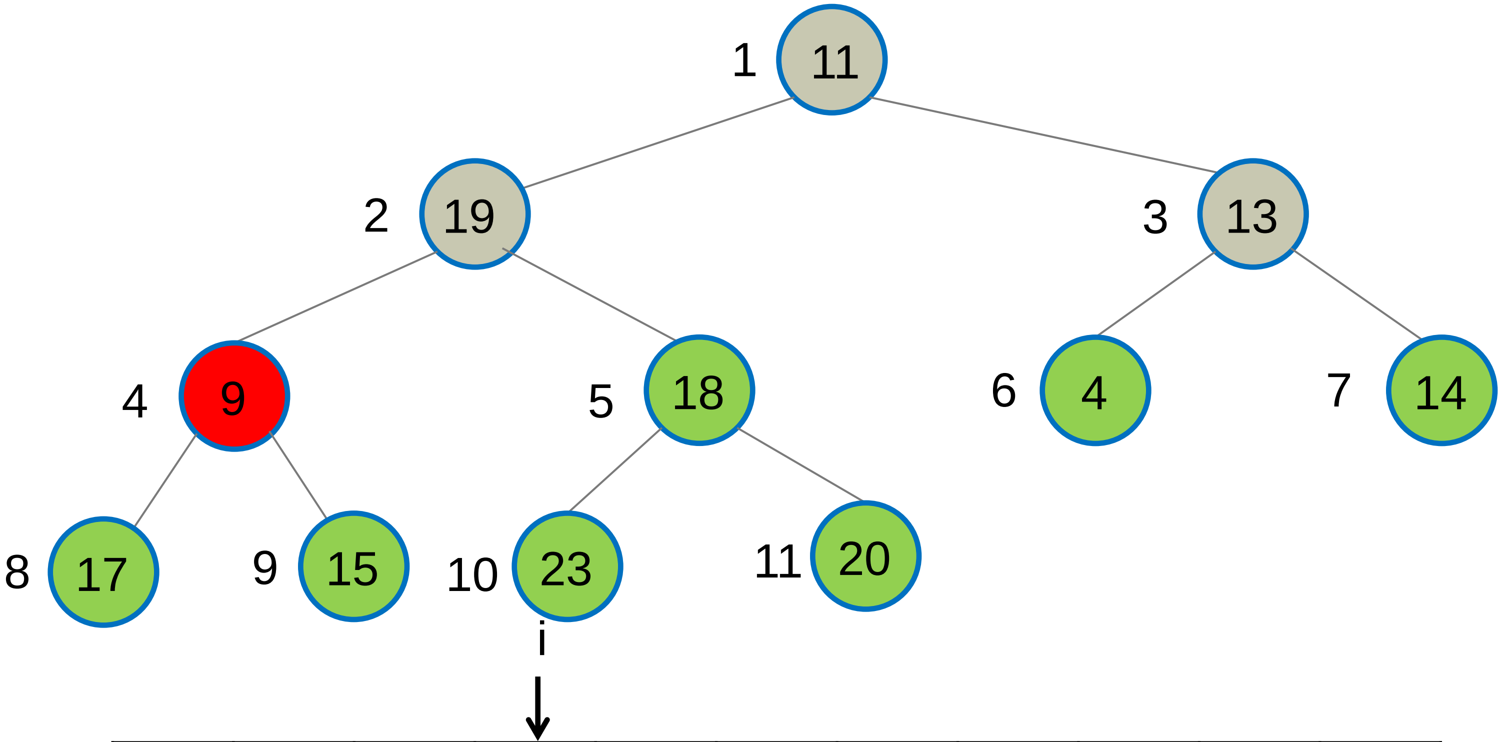
11	19	13	9	18	4	14	17	15	23	20
1	2	3	4	5	6	7	8	9	10	11

Heapify in O(N)



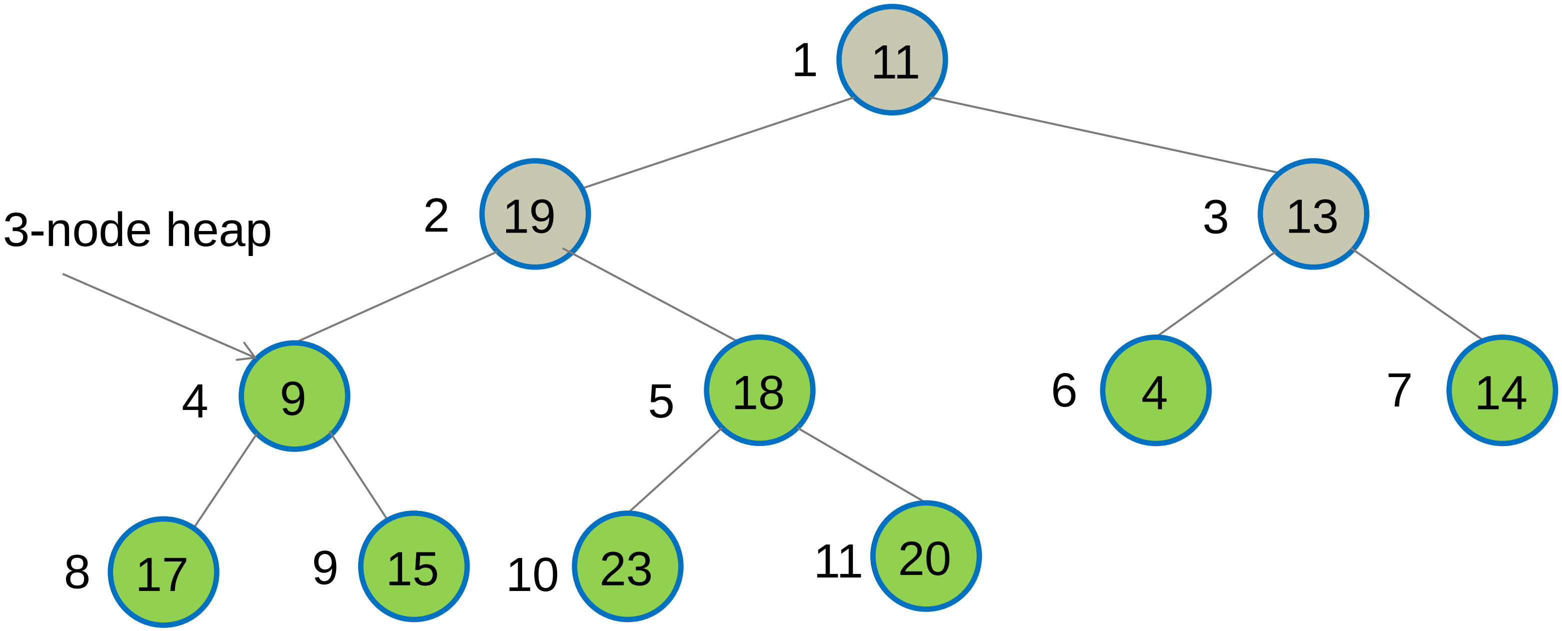
11	19	13	9	18	4	14	17	15	23	20
1	2	3	4	5	6	7	8	9	10	11

Heapify in O(N)



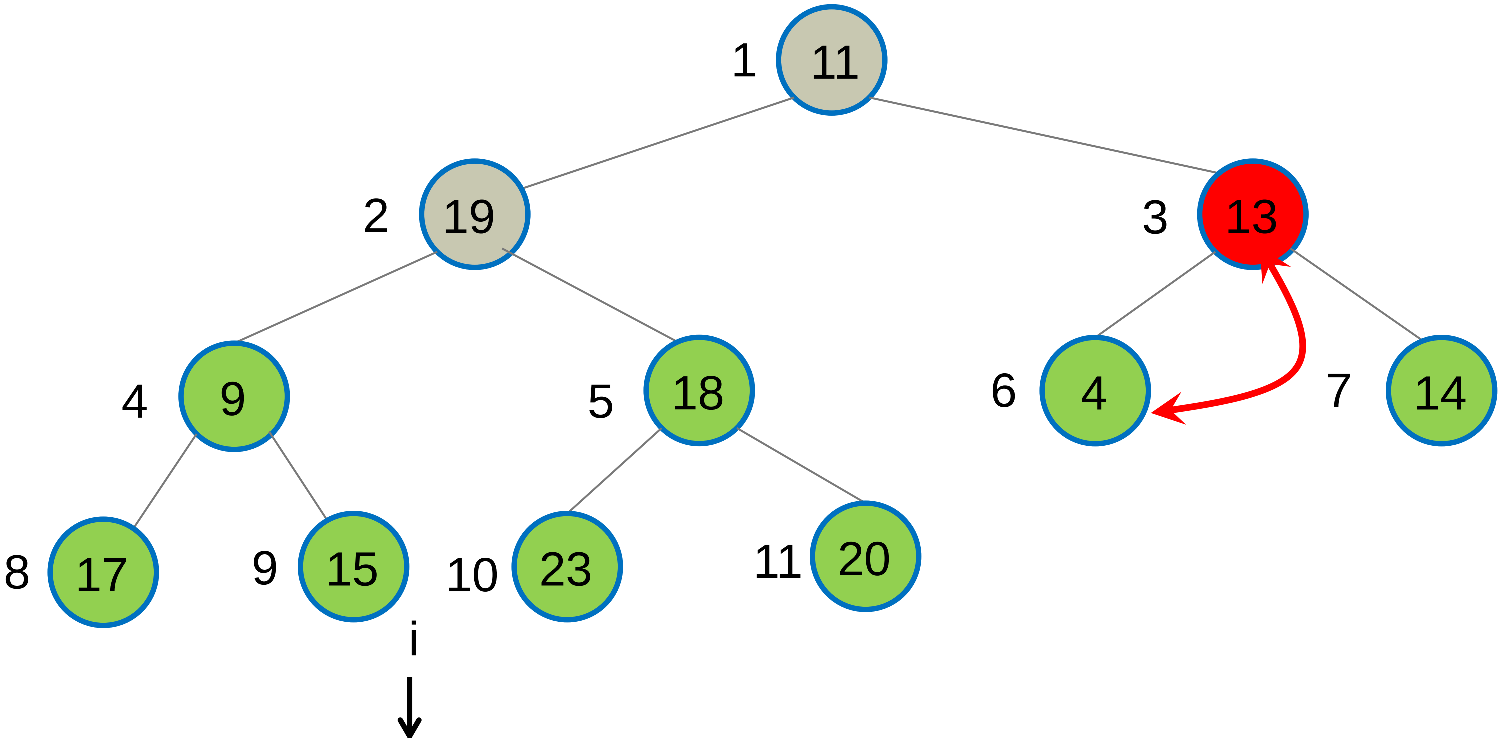
11	19	13	9	18	4	14	17	15	23	20
1	2	3	4	5	6	7	8	9	10	11

Heapify in O(N)



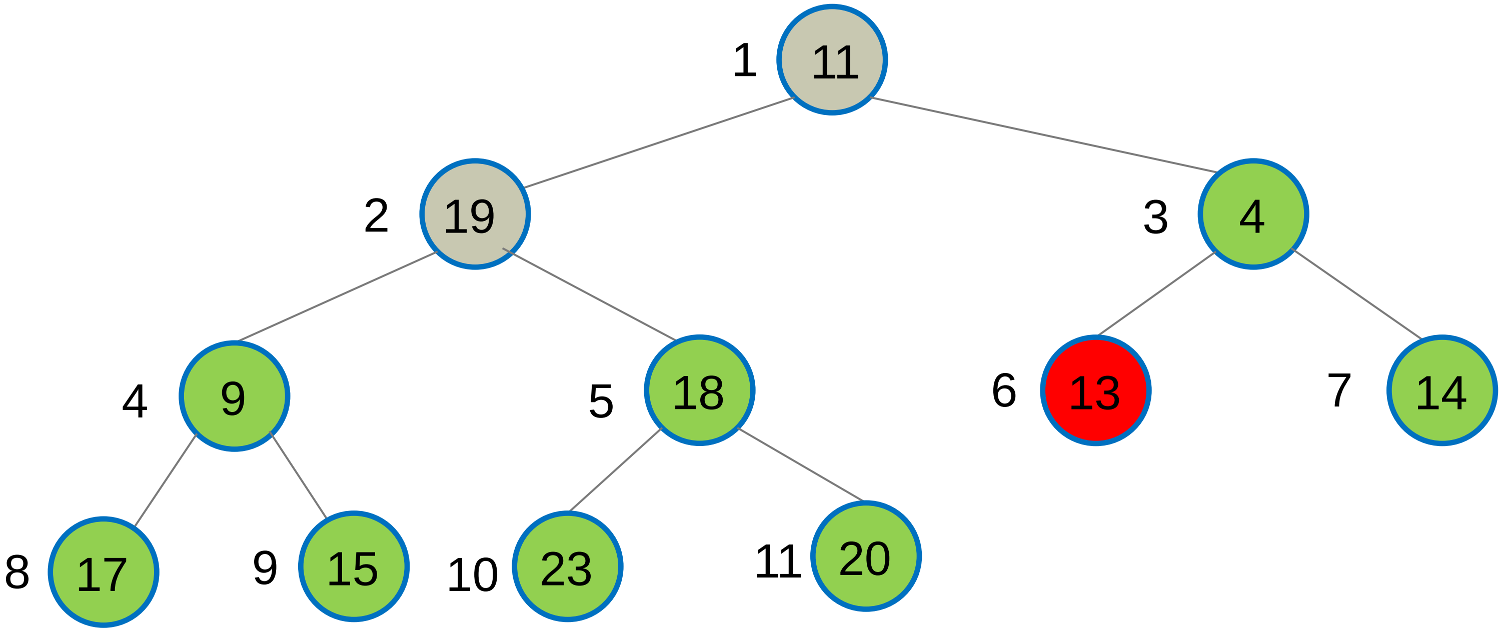
11	**19**	**13**	**9**	**18**	**4**	**14**	**17**	**15**	**23**	**20**
1	2	3	4	5	6	7	8	9	10	11

Heapify in O(N)



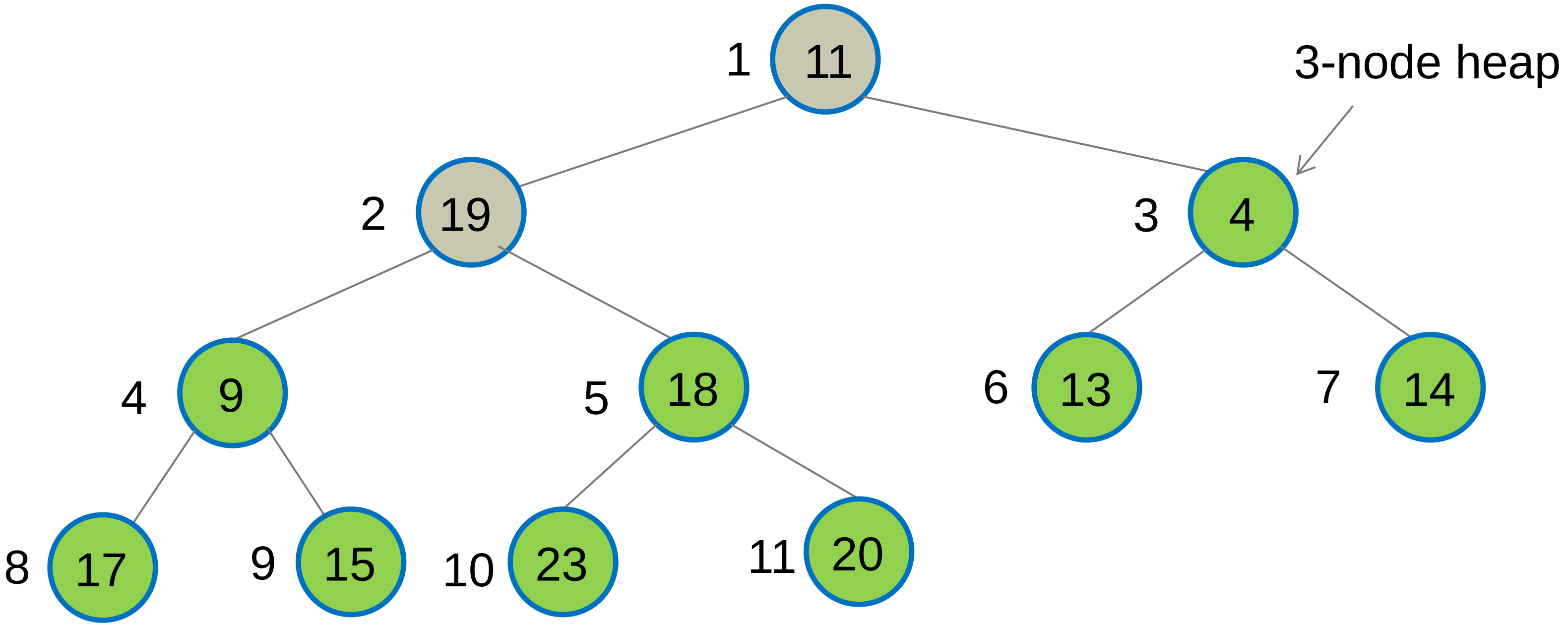
11	19	13	9	18	4	14	17	15	23	20
1	2	3	4	5	6	7	8	9	10	11

Heapify in O(N)



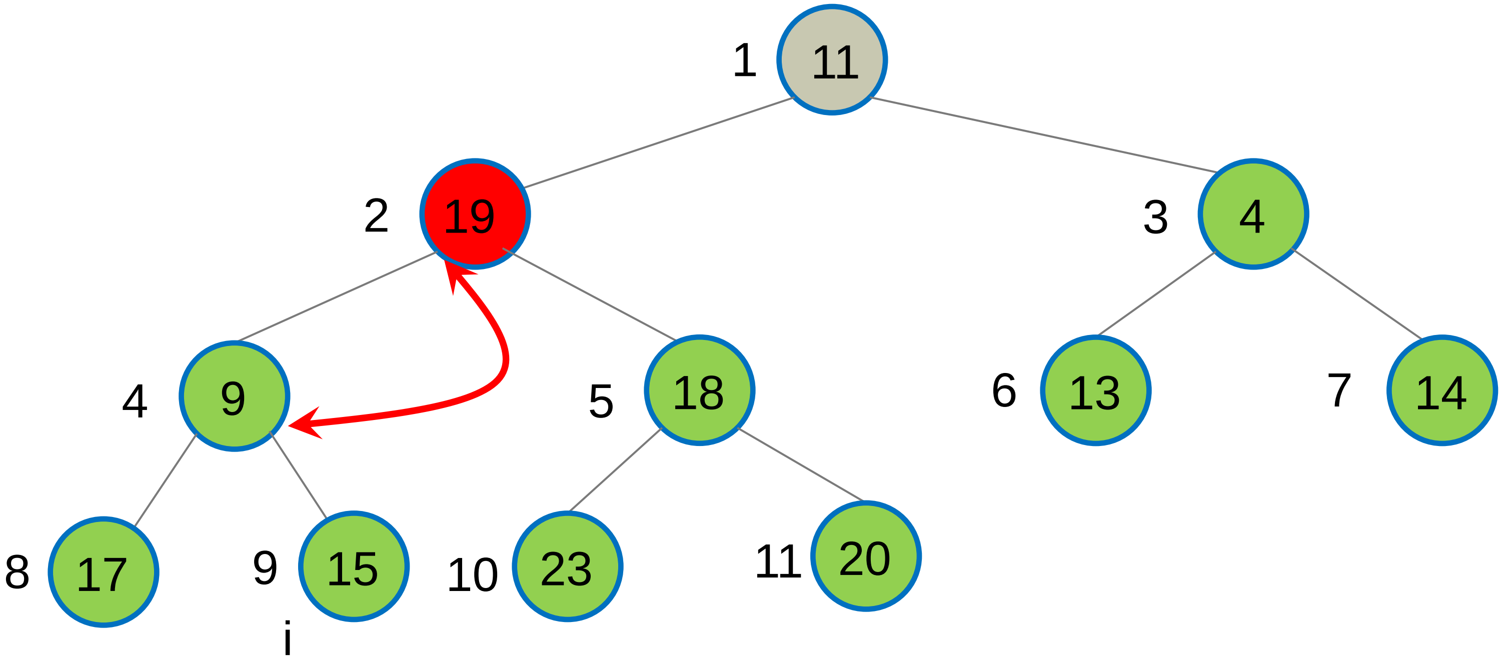
11	19	4	9	18	13	14	17	15	23	20
1	2	3	4	5	6	7	8	9	10	11

Heapify in O(N)



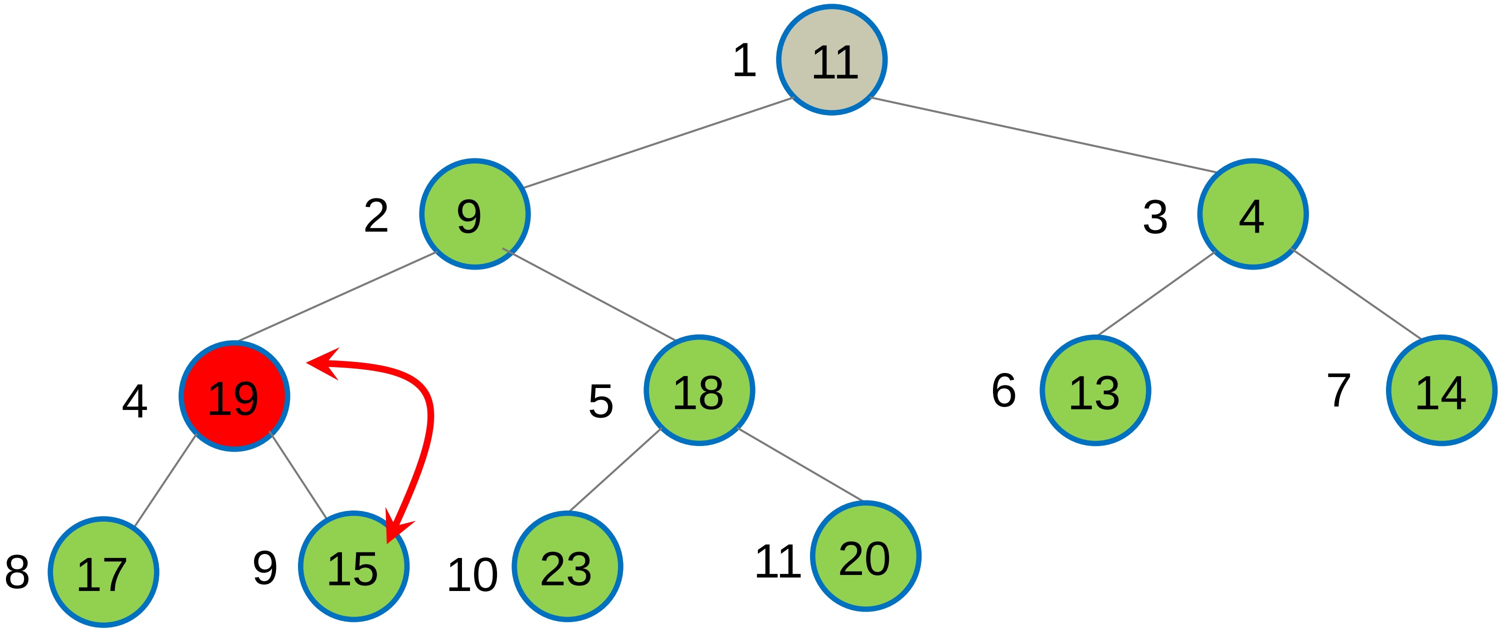
11	19	4	9	18	13	14	17	15	23	20
1	2	3	4	5	6	7	8	9	10	11

Heapify in O(N)



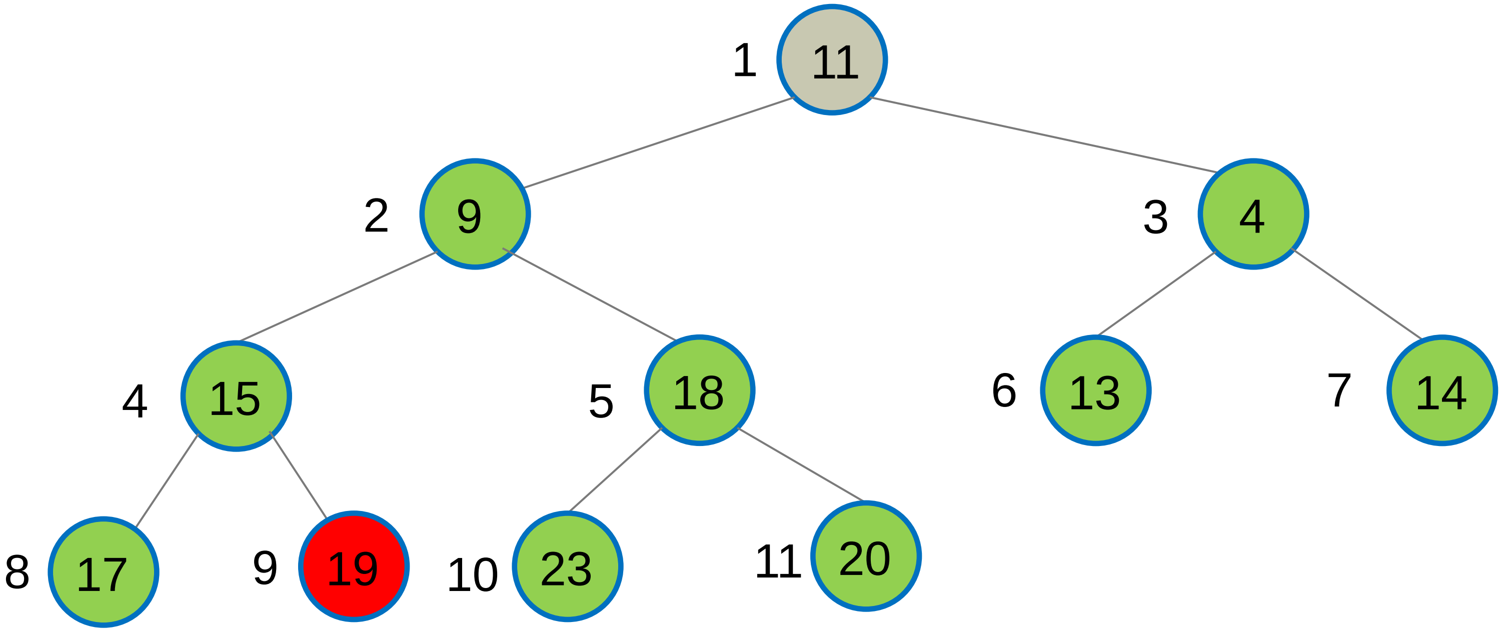
11	19	4	9	18	13	14	17	15	23	20
1	2	3	4	5	6	7	8	9	10	11

Heapify in O(N)



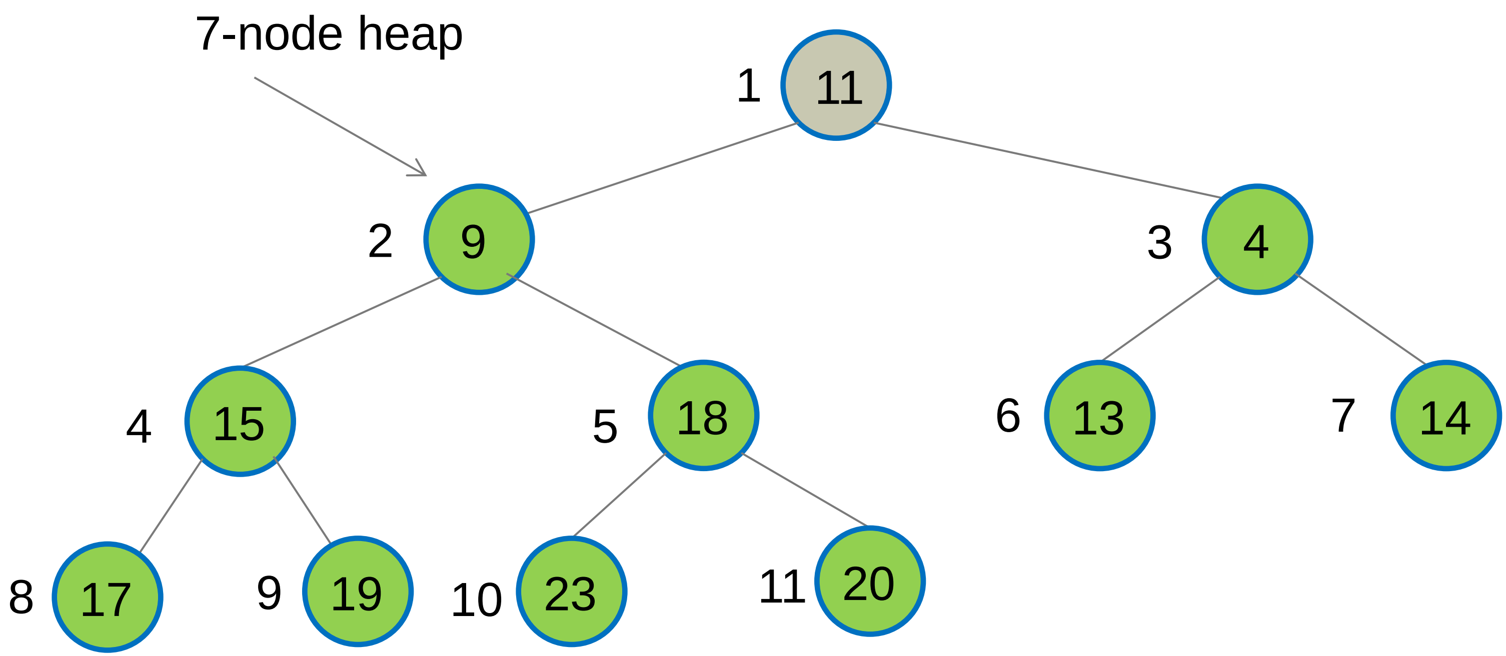
11	9	4	19	18	13	14	17	15	23	20
1	2	3	4	5	6	7	8	9	10	11

Heapify in O(N)



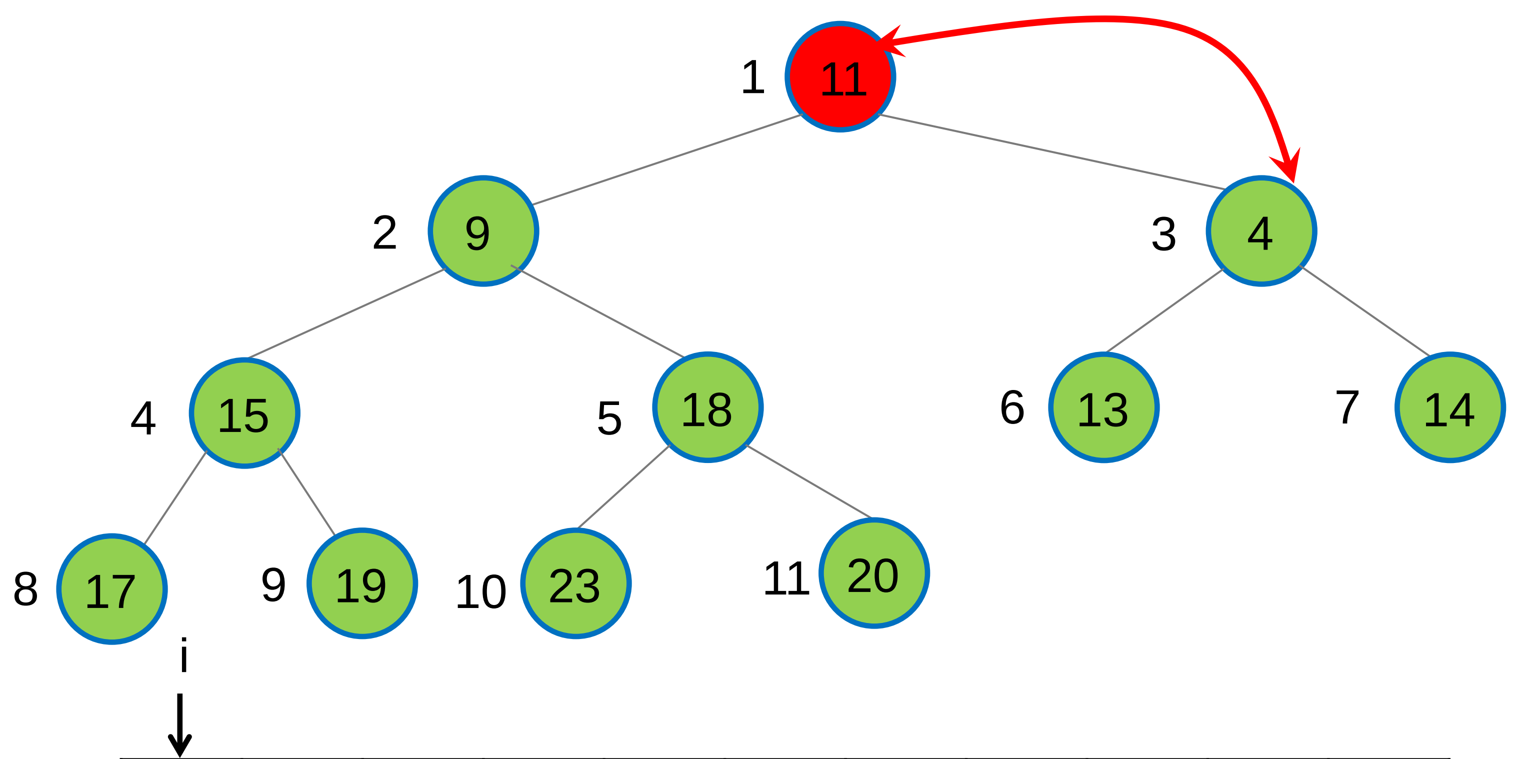
11	9	4	15	18	13	14	17	19	23	20
1	2	3	4	5	6	7	8	9	10	11

Heapify in O(N)



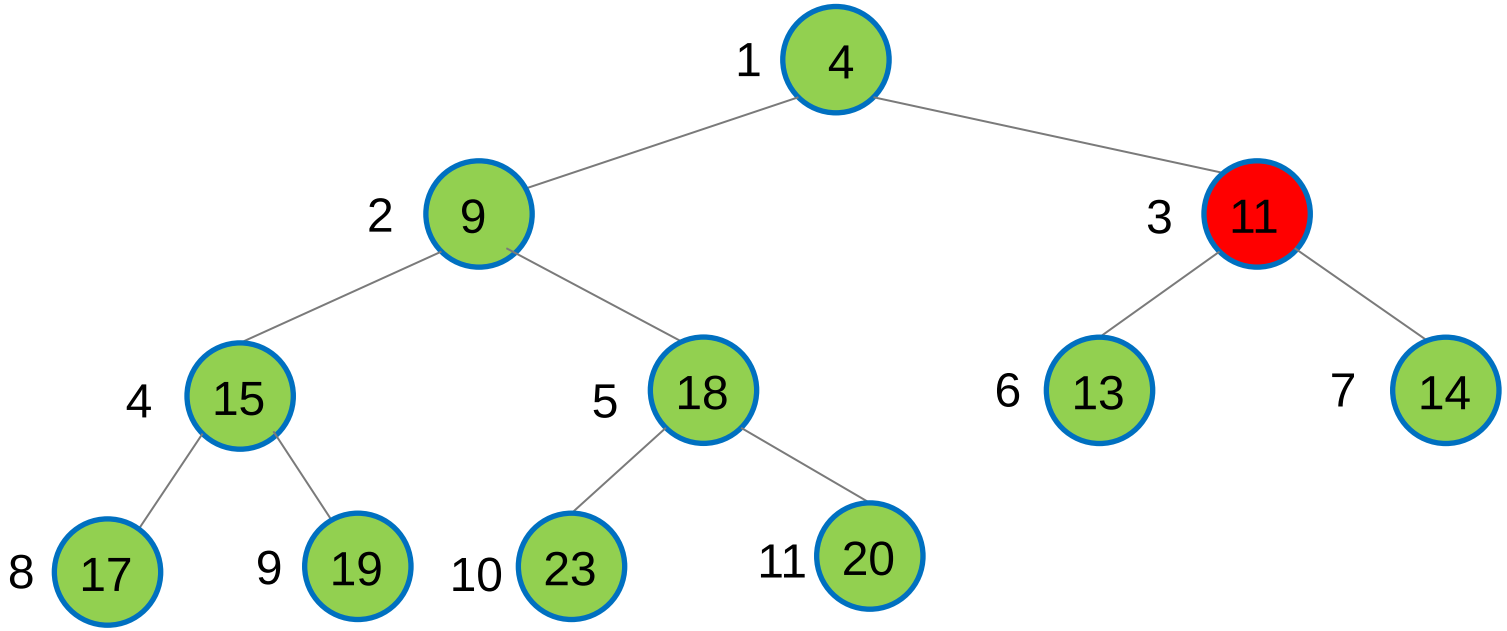
11	9	4	15	18	13	14	17	19	23	20
1	2	3	4	5	6	7	8	9	10	11

Heapify in O(N)



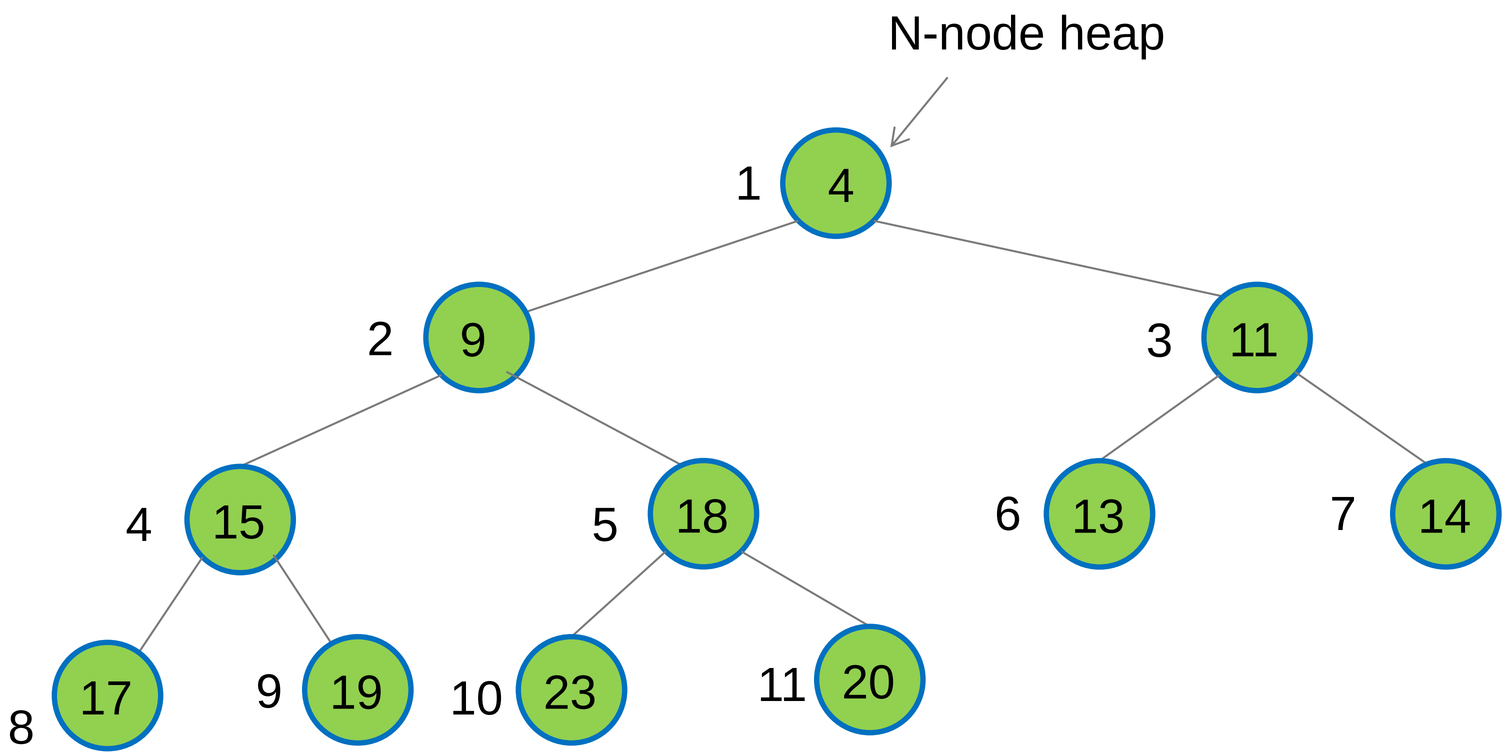
11	9	4	15	18	13	14	17	19	23	20
1	2	3	4	5	6	7	8	9	10	11

Heapify in O(N)



4	9	11	15	18	13	14	17	19	23	20
1	2	3	4	5	6	7	8	9	10	11

Heapify in O(N)



4	9	11	15	18	13	14	17	19	23	20
1	2	3	4	5	6	7	8	9	10	11

Heapify in O(N)

Complexity Analysis

Swaps for level 1 nodes (i.e., leaf nodes):	0	number of nodes: N/2
Swaps for level 2 nodes:	1	number of nodes: N/4
Swaps for level 3 nodes:	2	number of nodes: N/8
Swaps for level 4 nodes:	3	number of nodes: N/16
…		
Swap for root node:	h	number of nodes: 1

Total cost: 1*N/4 + 2*N/8 + 3*N/16 + …. + h*1

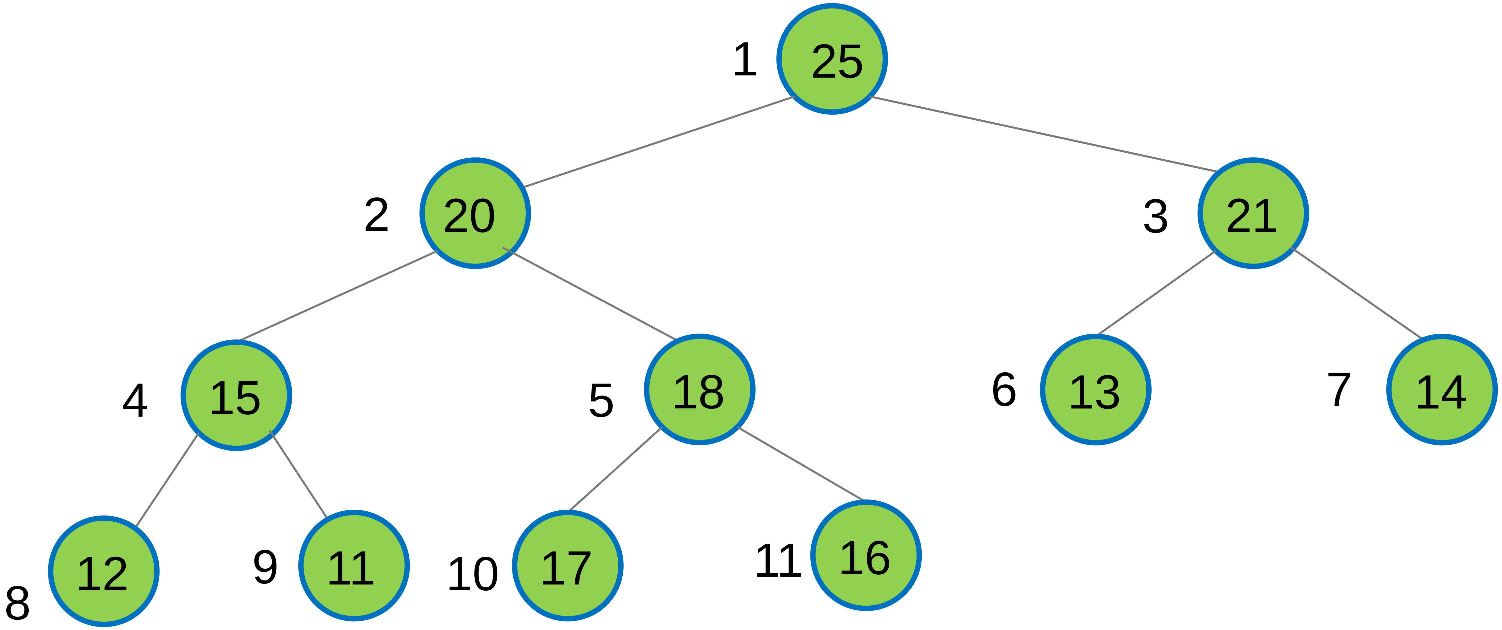
= N(1/4 + 2/8 + 3/16 + … + h/N)

<= N(1/4 + 2/8 + 3/16 + ….) // (1/4 + 2/8 + 3/16 + …) converges to a constant)

= O(N)

Min-Heap and Max-Heap

- The examples we saw earlier prefer smaller elements (i.e., smaller element is at the top of the heap). Such heap is called a min-heap
- A max-heap prefers larger elements and can be implemented similarly



Heap Sort

- Heapify the input array A in a min-heap
- Initilalize an empty array B of size N
- For i = 1 to N
  - Remove the top element from the heap (e.g., A[1]) and put it at B[i]
- Copy array B to array A

Time complexity:

Space complexity:

Is the above sorting algorithm stable?

In-place Algorithm: An algorithm is called in-place if it uses only constant space (i.e., O(1)) additional to the space used by input.

Is the above version of Heap Sort In-place?

The above version of Heap Sort is not in-place as it requires an additional O(N) space for array B.

In-Place Heap Sort

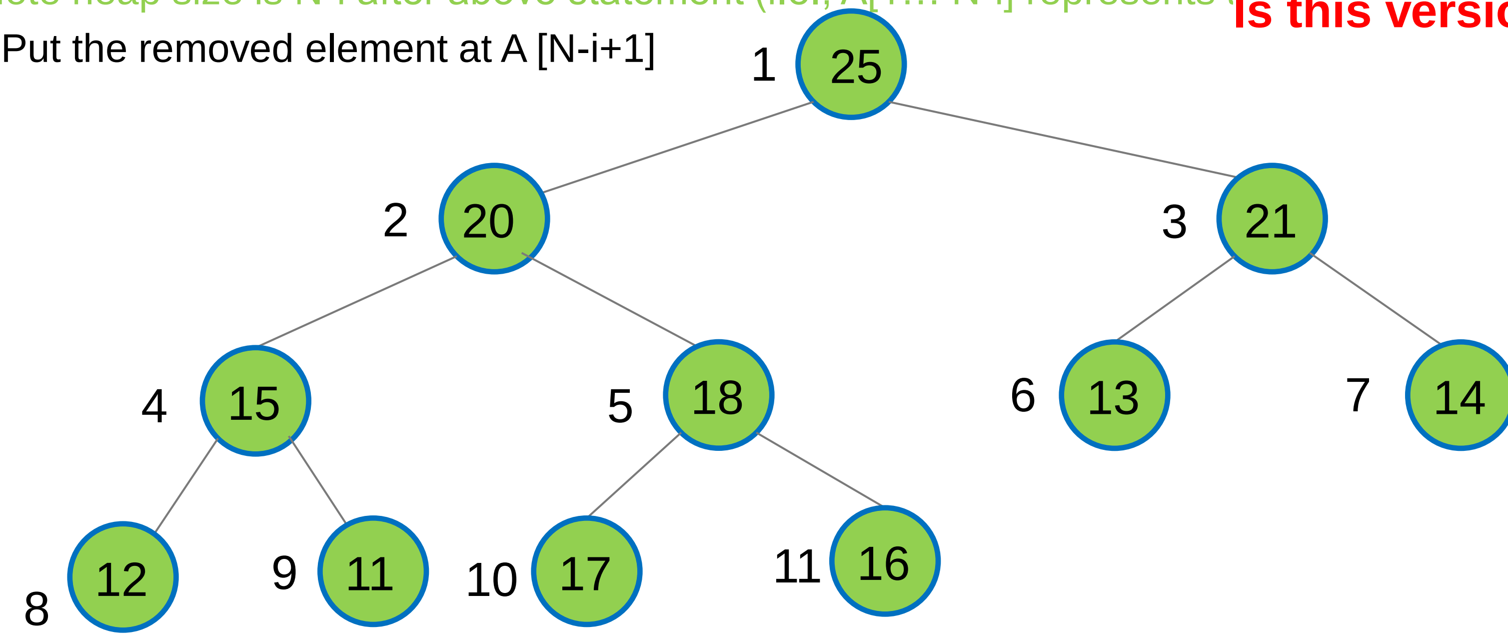
- Heapify the input array A using a max-heap
- For i = 1 to N
  - Remove biggest element from the heap (i.e., A[1])

  // Note heap size is N-i after above statement (i.e., A[1… N-i] represents the heap)
  - Put the removed element at A [N-i+1]

Time complexity?

Space Complexity?

Is this version stable?



20	18	16	15	17	13	14	12	11	21	25
1	2	3	4	5	6	7	8	9	10	11

Divide and Conquer Sorting

Divide and Conquer Paradigm

- Divide the problem into smaller sub-problems
- Conquer (solve) each sub-problem
- Combine the results

Divide and Conquer Sorting Algorithms

- Merge Sort
- Quick Sort

Merge Sort

```
// Merge Function: Merging two sorted arrays
i = 1, j = 1;
while i <= a.length and j <= b.length
        if a [ i ] < b [  j ]
                answer.append(a[ i ])
                i++;
        else
                answer.append(b[ j ])
                j++;
while i <= a.length
        answer.append(a[ i ])
        i++;
while j <= b.length
        answer.append(b[ j ]);
        j++;
```

Time Complexity:

= O(|a| + |b|)

Is this In-place?

Is this version of merging stable?

a

1	5	6

b

3	4	7

1	3	4	5	6	7

answer

Merge Sort

```
// Merge Function: Merging two sorted arrays
i = 1, j = 1;
while i <= a.length and j <= b.length
        if a [ i ] <= b [  j ]
                answer.append(a[ i ])
                i++;
        else
                answer.append(b[ j ])
                j++;
while i <= a.length
        answer.append(a[ i ])
        i++;
while j <= b.length
        answer.append(b[ j ]);
        j++;
```

Time Complexity:

= O(|a| + |b|)

Note: Merge Function (hence Merge Sort) is not in-place

Is this version of merging stable?

a

1	5	6

b

3	4	7

1	3	4	5	6	7

answer

Merge Sort

Divide Step

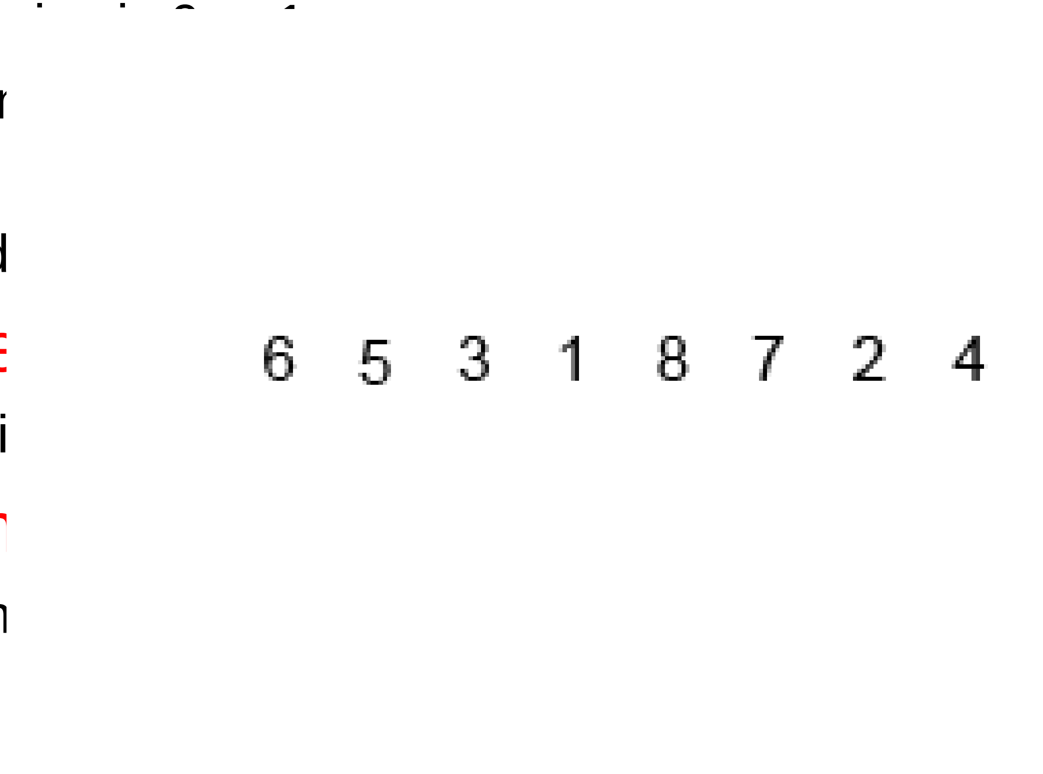
- If array
  - returr
- Else
  - divid

Conque

- Recursi

Combir

- Combin

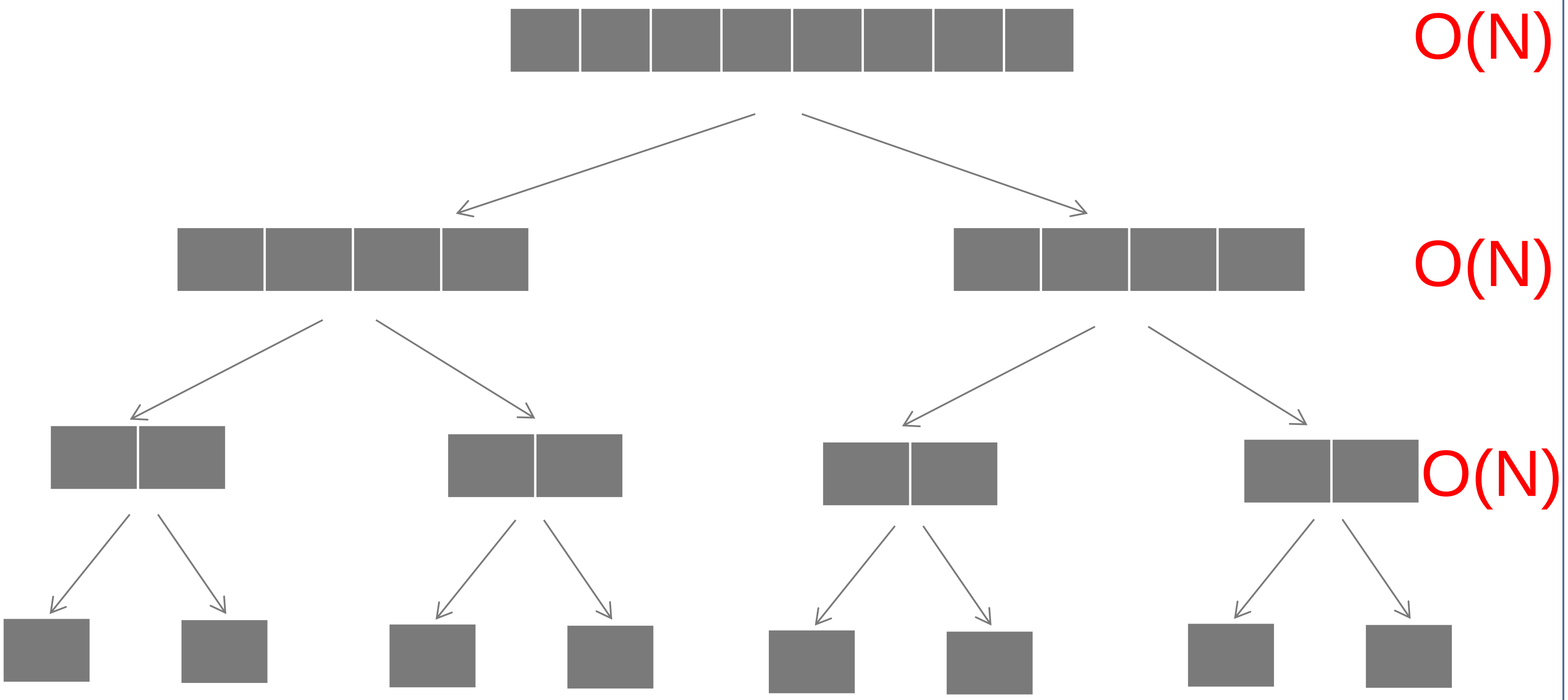


Source: wikimedia.org

Is Merge Sort stable?

Is Merge Sort in-place?

Complexity of Merge Sort



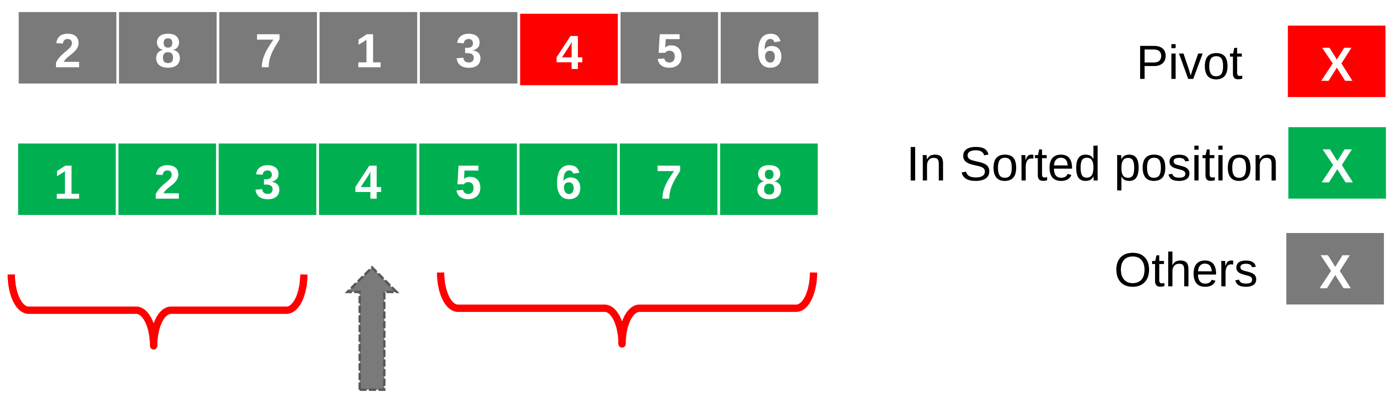
Height: O(log N)
Worst-case complexity: O(N log N)

Best-case time complexity?
Average-case time complexity?

Quicksort

Partitioning

- Choose a pivot p
- Partition the array in two sub-arrays w.r.t. p
  - LEFT ← elements smaller than or equal to p
  - RIGHT ← elements greater than p
- QuickSort(LEFT)
- QuickSort(RIGHT)

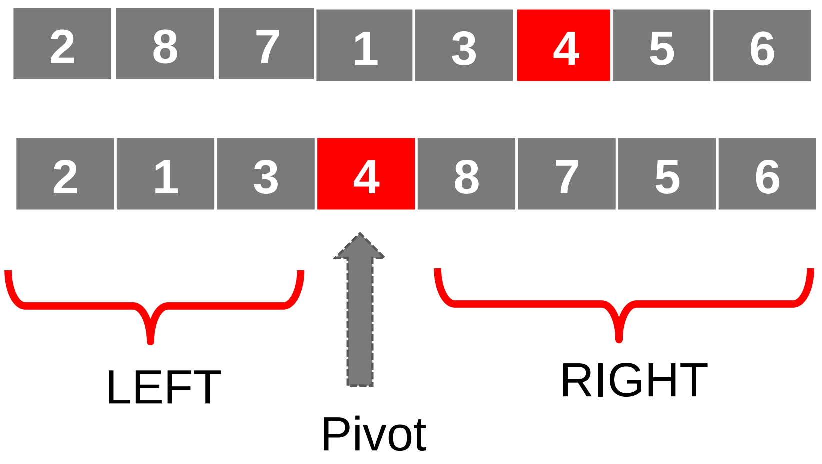


In sorted position

Partitioning: An out-of-place version

- Initialize two lists LEFT and RIGHT
- For each element e (except pivot)
  - If e ≤ pivot
    - Insert e in LEFT
  - If e > pivot
    - Insert e in RIGHT
- Copy {LEFT, pivot, RIGHT} to the array

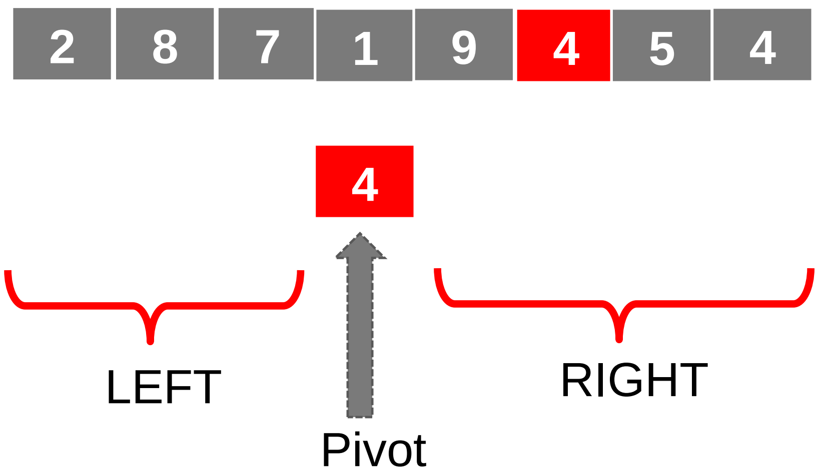
This is clearly not in-place.
Will this result in stable sorting?



Partitioning: An out-of-place version

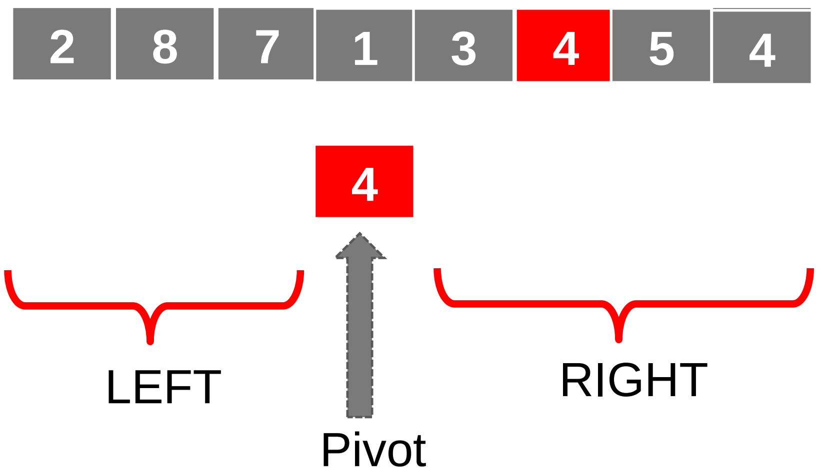
- Initialize two lists LEFT and RIGHT
- For each element e (except pivot)
  - If e ≤ pivot
    - Insert e in LEFT
  - If e > pivot
    - Insert e in RIGHT
- Copy {LEFT, pivot, RIGHT} to the array

This version is unstable but it can be made stable!



Partitioning: A stable version

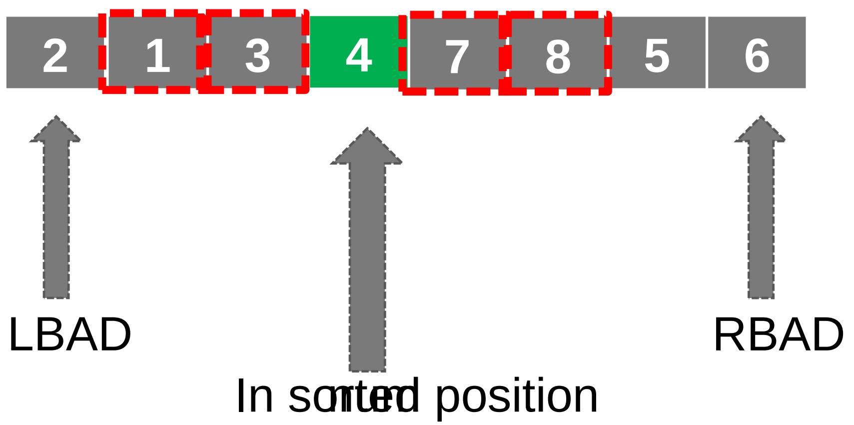
- Initialize two lists LEFT and RIGHT
- For each element e (except pivot)
  - If e ≤ pivot
    - If e == pivot and e.index > pivot.index
      - Insert e in RIGHT
    - Else
      - Insert e in LEFT
  - If e > pivot
    - Insert e in RIGHT
- Copy {LEFT, pivot, RIGHT} to the array



In-Place Partitioning

- num ← the number of elements smaller than or equal to pivot O(N)
- Swap pivot with element at num
- Repeat until no bad element is found
  - Find a bad element (LBAD) on the L.H.S. of pivot
  - Find a bad element (RBAD) on the R.H.S. of pivot
  - Swap LBAD and RBAD

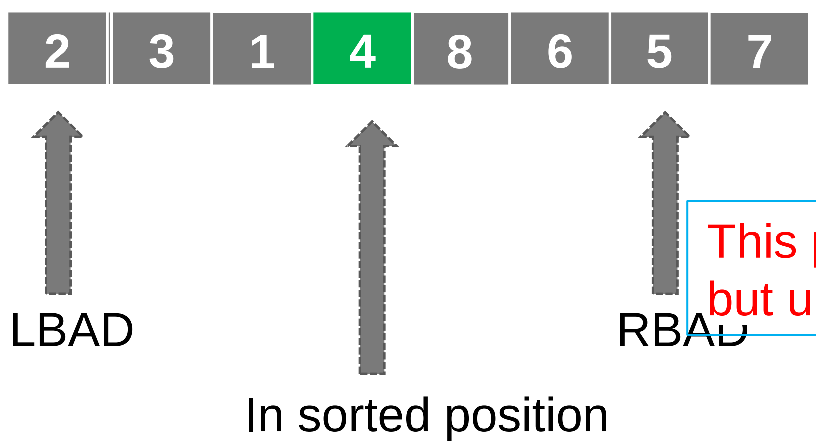
O(N)



This partitioning algorithm is in-place but results in unstable sorting.

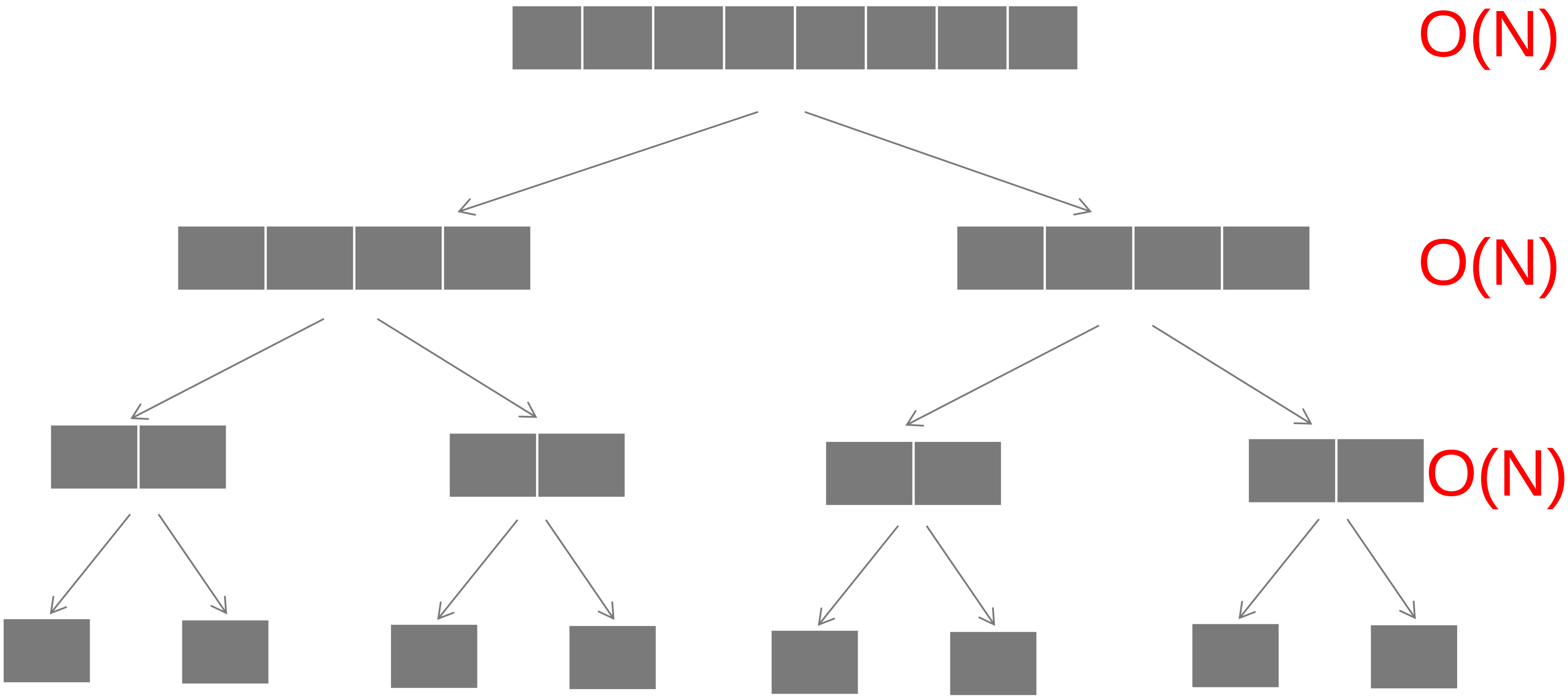
In-Place Partitioning (Improved)

- Swap pivot with the right most element
- LBAD points to the left most element
- RBAD points to the second right most element
- Repeat until LBAD and RBAD point to the same element
  - Move LBAD towards right until it points to an element e > pivot
  - Move RBAD towards left until it points to an element e ≤ pivot
  - Swap elements pointed by LBAD and RBAD
- Swap pivot with the element pointed by LBAD/RBAD



This partitioning is in-place but unstable.

Best-case complexity of Quicksort

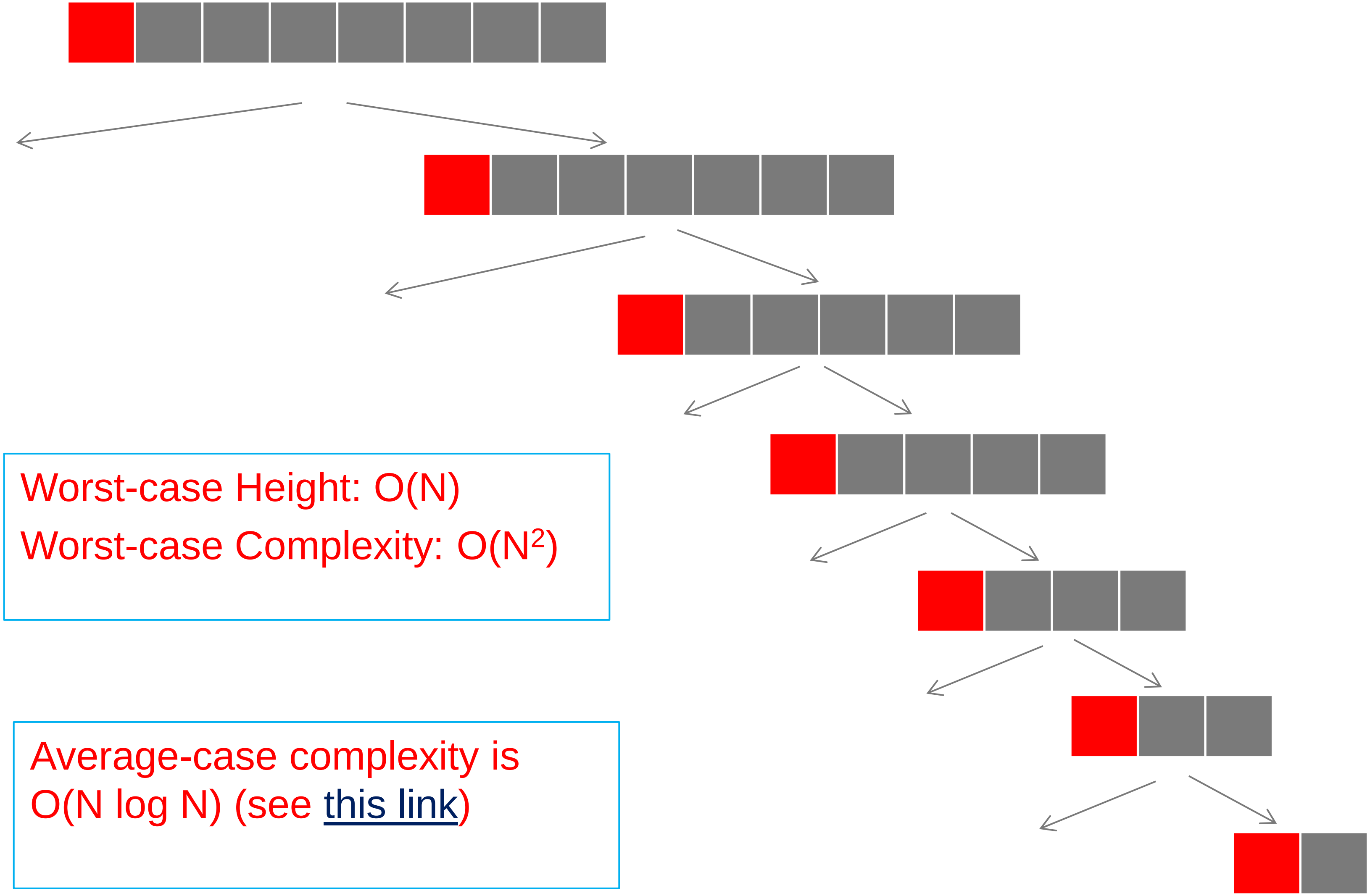


Best-case Height: O(log N)
Best-case complexity: O(N log N)

Important: Quicksort is not in-place even when in-place partitioning is used. Why?

Requires O(log N) space for recursion

Worst-case Complexity of Quicksort



Worst-case Height: O(N)

Worst-case Complexity: $O(N^2)$

Average-case complexity is O(N log N) (see this link)

Summary of complexities

	Best	Worst	Average	Stable?	In-place?
Selection Sort	$O(N^2)$	$O(N^2)$	$O(N^2)$	No	Yes
Insertion Sort	$O(N)$	$O(N^2)$	$O(N^2)$	Yes	Yes
Heap Sort	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	No	Yes
Merge Sort	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	Yes	No
Quick Sort	$O(N \log N)$	$O(N^2)$	$O(N \log N)$	Depends	No

Is it possible to develop a sorting algorithm with worst-case time complexity better than $O(N \log N)$?

Lower Bound Complexity

- Lower bound complexity for a **problem** is the lowest possible complexity **any** algorithm (known or unknown) can achieve to solve the problem
- What is the lower bound complexity of finding the minimum element in an array of N elements
  - **Ans:** O(N)
  - Since the algorithm we saw earlier has O(N) complexity, it is optimal
- What is the lower bound complexity for sorting?
  - For comparison-based algorithm, lower bound complexity is O(N log N).
  - Read https://www.cs.cmu.edu/~avrim/451f11/lectures/lect0913.pdf to see why the lower bound is O(N log N)

See you next week ☺