Department of
Computer Engineering

# Data Structure & Algorithms

Heap

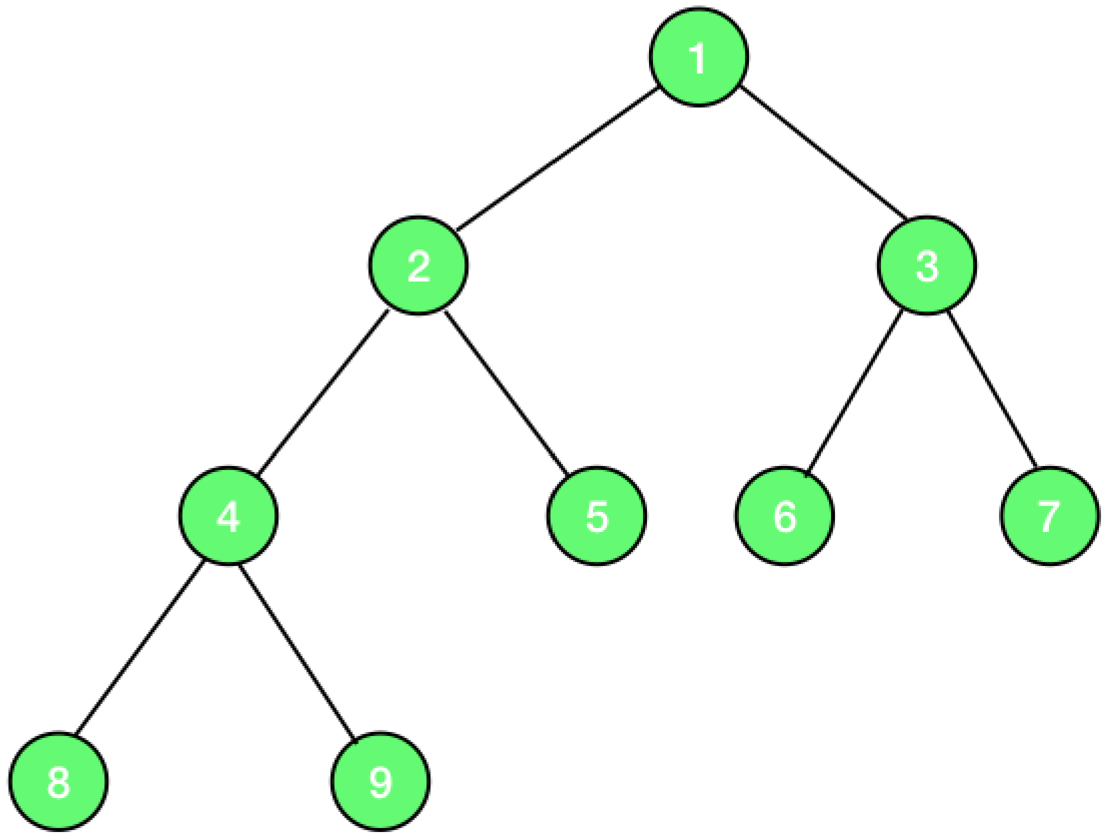
## Heap

- A heap is a data structure which uses a binary tree for its implementation. It is the base of the algorithm heapsort and is also used to implement priority queue. It is basically a complete binary tree and generally implemented using an array.
- The root of the tree is the first element of the array.

## Complete Binary Tree

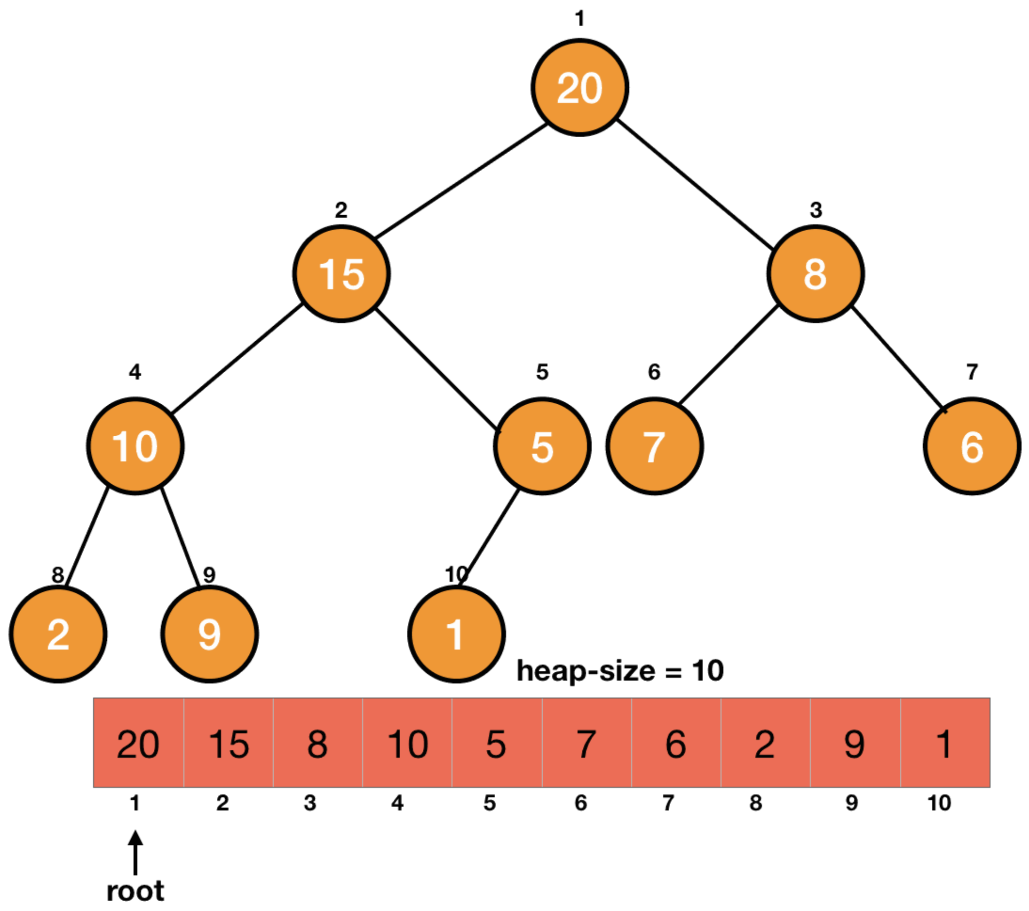
- A binary tree which is completely filled with a possible exception at the bottom level i.e., the last level may not be completely filled and the bottom level is filled from left to right.

# Complete Binary Tree – Example



A Complete Binary Tree

# Heap – Example



## Heap – Cont.

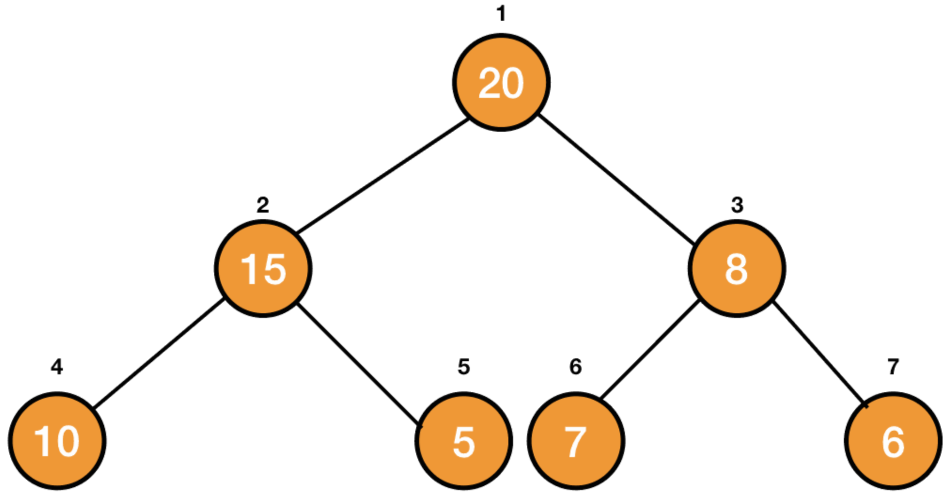
- Since a heap is a binary tree, we can also use the properties of a binary tree for a heap i.e.,

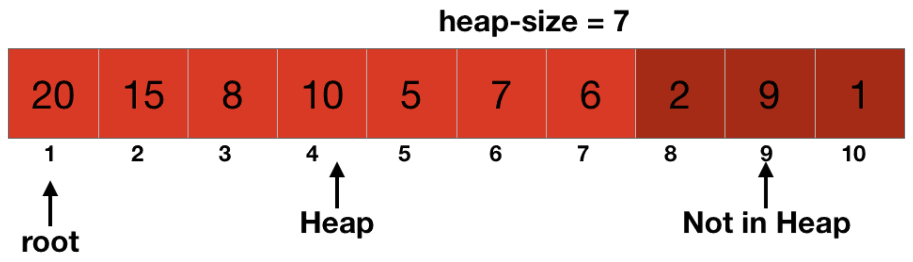
$$Parent(i) = \left\lfloor \frac{i}{2} \right\rfloor$$
$$Left(i) = 2 * i$$
$$Right(i) = 2 * i + 1$$

## Heap – Cont.

- We declare the size of the heap explicitly and it may differ from the size of the array. For example, for an array with a size of Array.length, a heap will only contain the elements which are within the declared size of the heap.

## Heap – Example





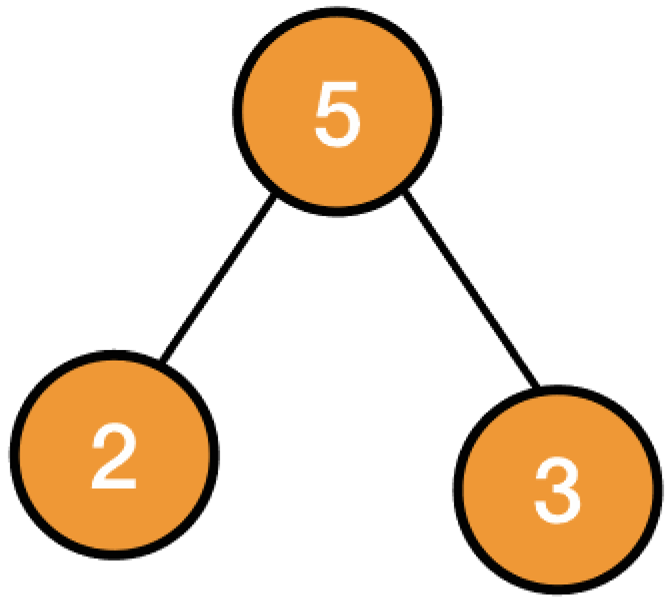
## Properties of a Heap

- Basically, we implement two kinds of heaps:
  - Max Heap
  - Min Heap

## Max Heap

- In a max-heap, the value of a node is either greater than or equal to the value of its children.
- $A\left[Parent[i]\right] \geq A[i]$ for all nodes $i\ >\ 1$

## Max Heap – Cont.

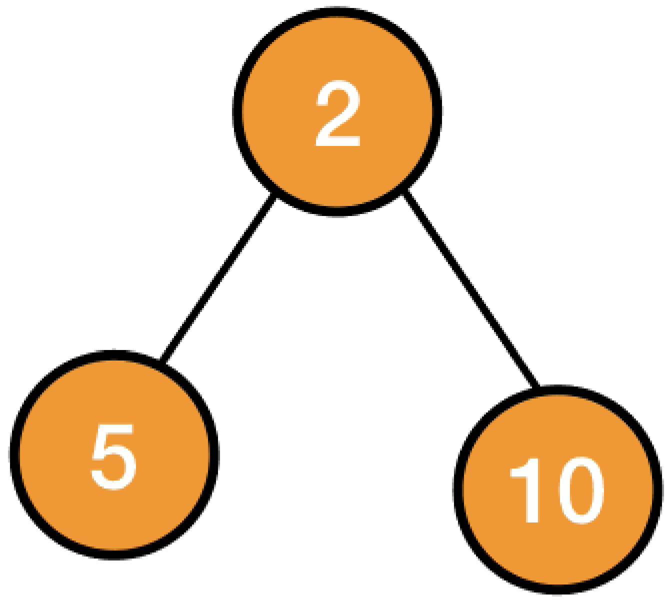


**Max Heap**

## Min Heap

- The value of a node is either smaller than or equal to the value of its children.
- $A[Parent[i]] \leq A[i]$ for all $nodes\ i > 1$

## Min Heap – Cont.



Min Heap

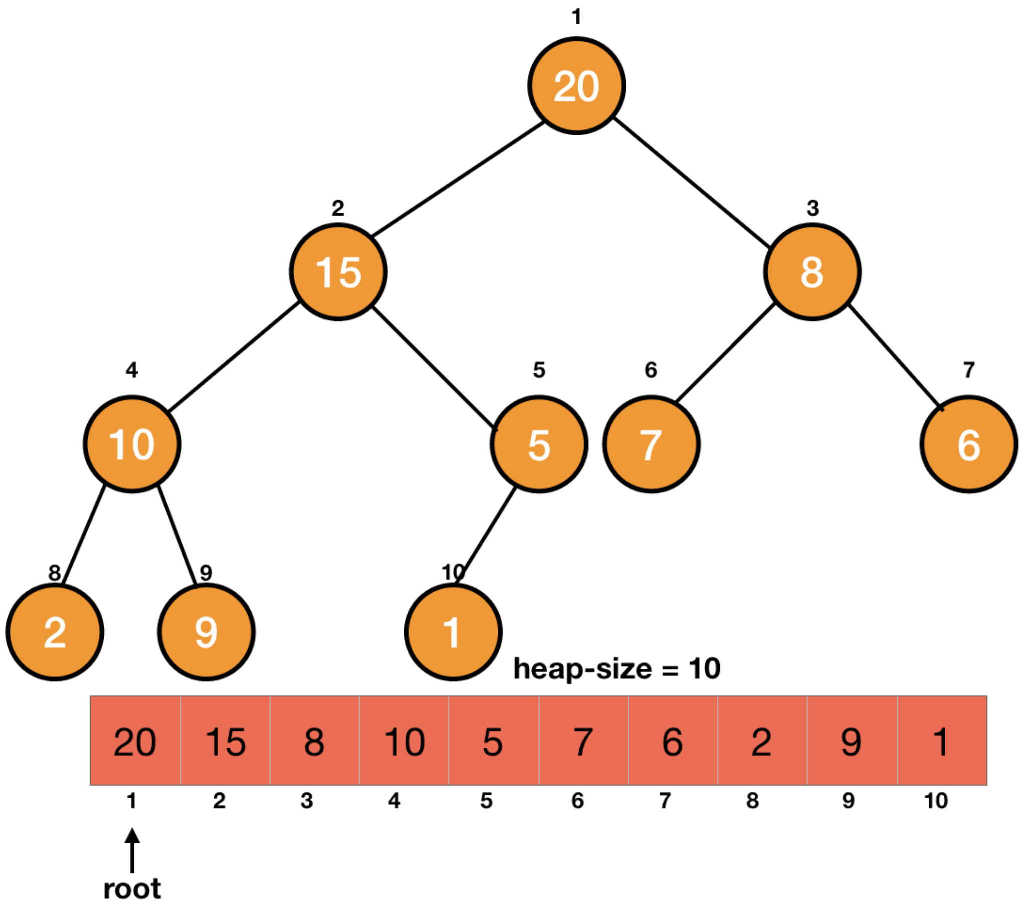
## Max Heap & Min Heap

- Thus in a max-heap, the largest element is at the root and in a min-heap, the smallest element is at the root.

## Heap Tree Representation

- There are 3 ways in which we represent a Heap data-structure
  - Adjacency Matrix or Adjacency List
  - Binary Tree
  - One-dimensional Arrays

## Representation of Heap in One-dimensional Array



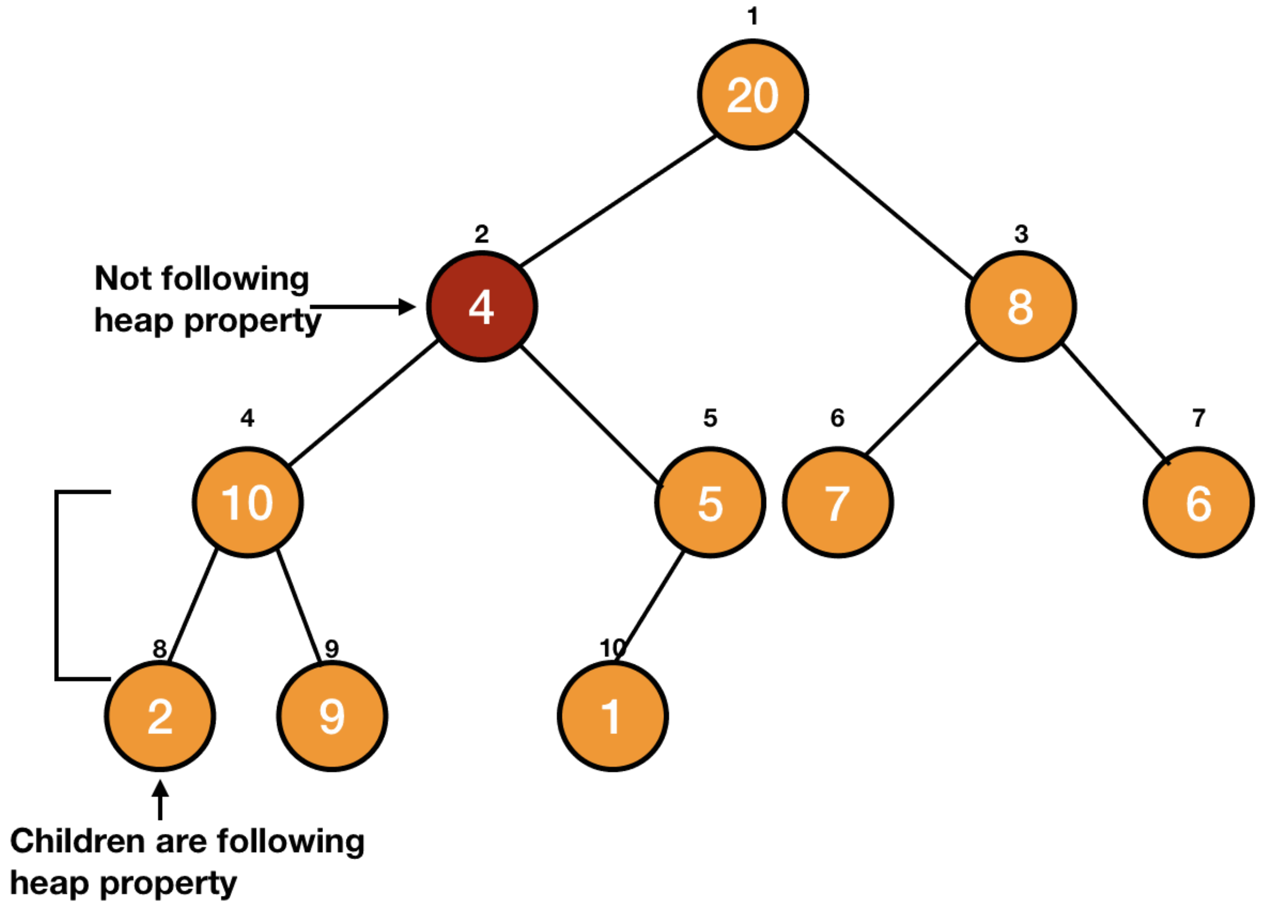
## Heapify

- Heapify is an operation applied on a node of a heap to maintain the heap property. It is applied on a node when its children (left and right) are heap (follow the property of heap) but the node itself may be violating the property.

## Max-Heapify

- Max-heapify is a process of arranging the nodes in correct order so that they follow max-heap property.

# Max-Heapify – Example

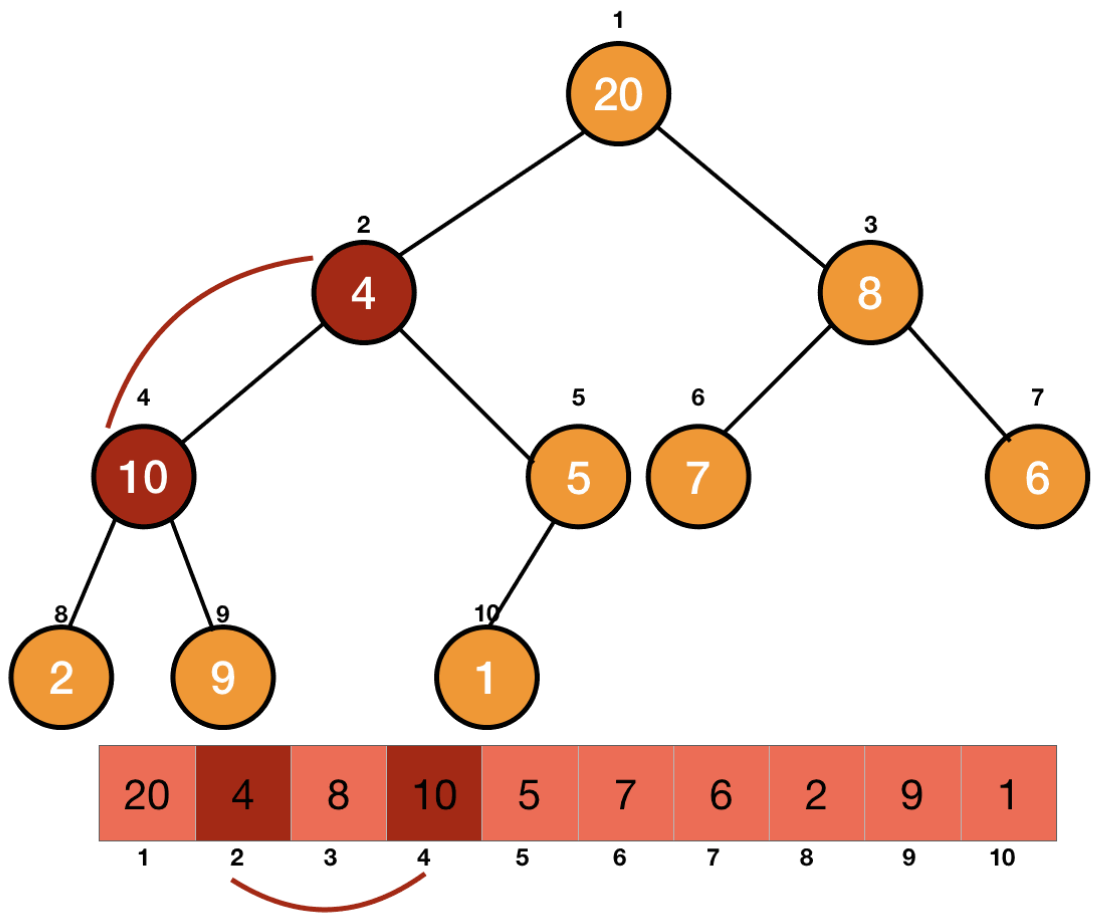


1
20
2
4
3
8
Not following heap property
4
10
5
5
6
7
7
6
8
2
9
9
10
1
Children are following heap property

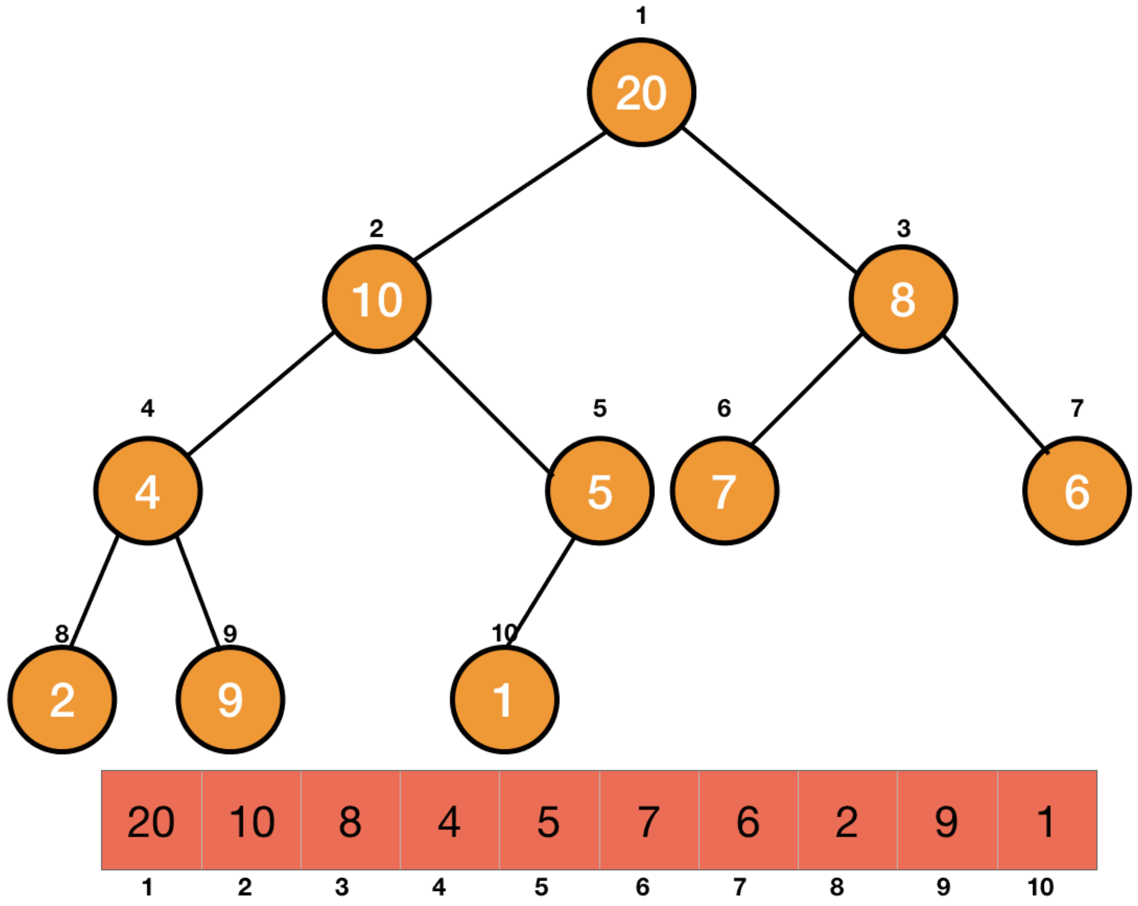
## Max-Heapify – Cont.

- We simply make the node travel down the tree until the property of the heap is satisfied. It is illustrated on a max-heap in the picture given below.

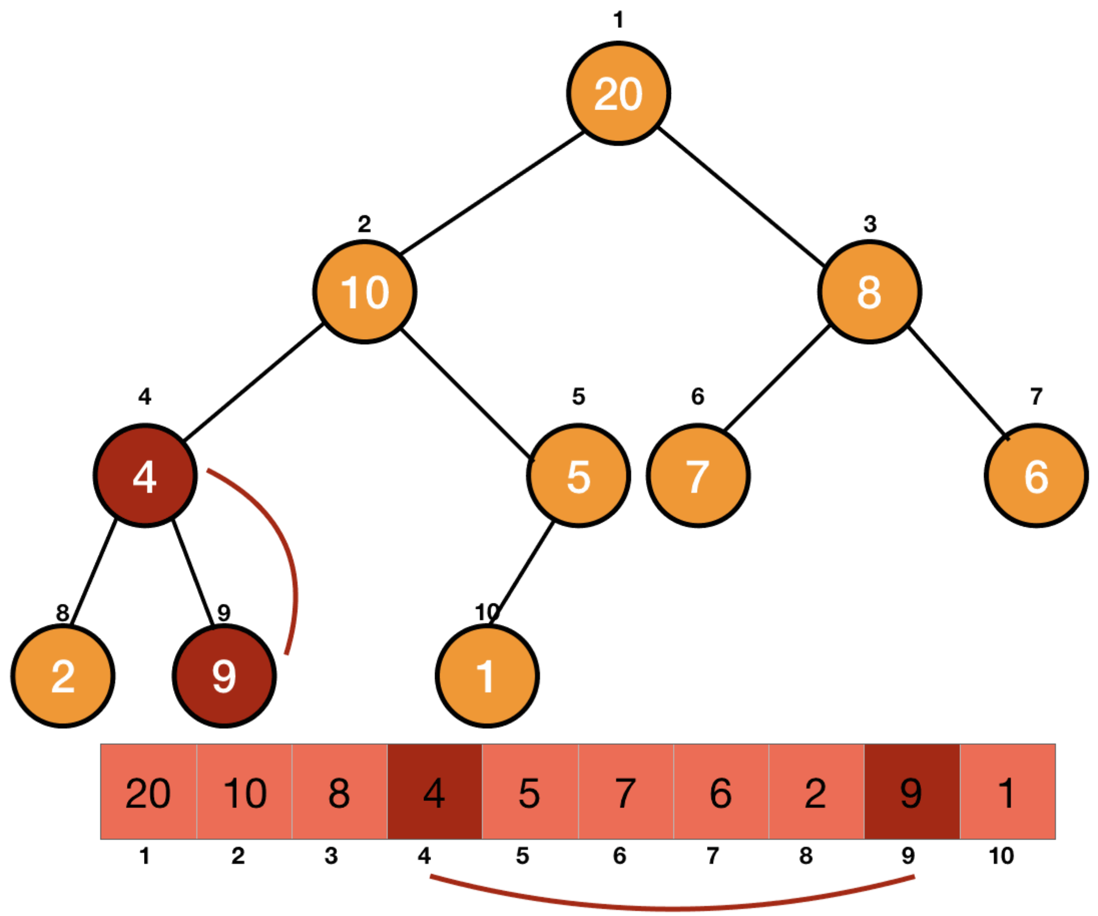
## Max-Heapify – step by step



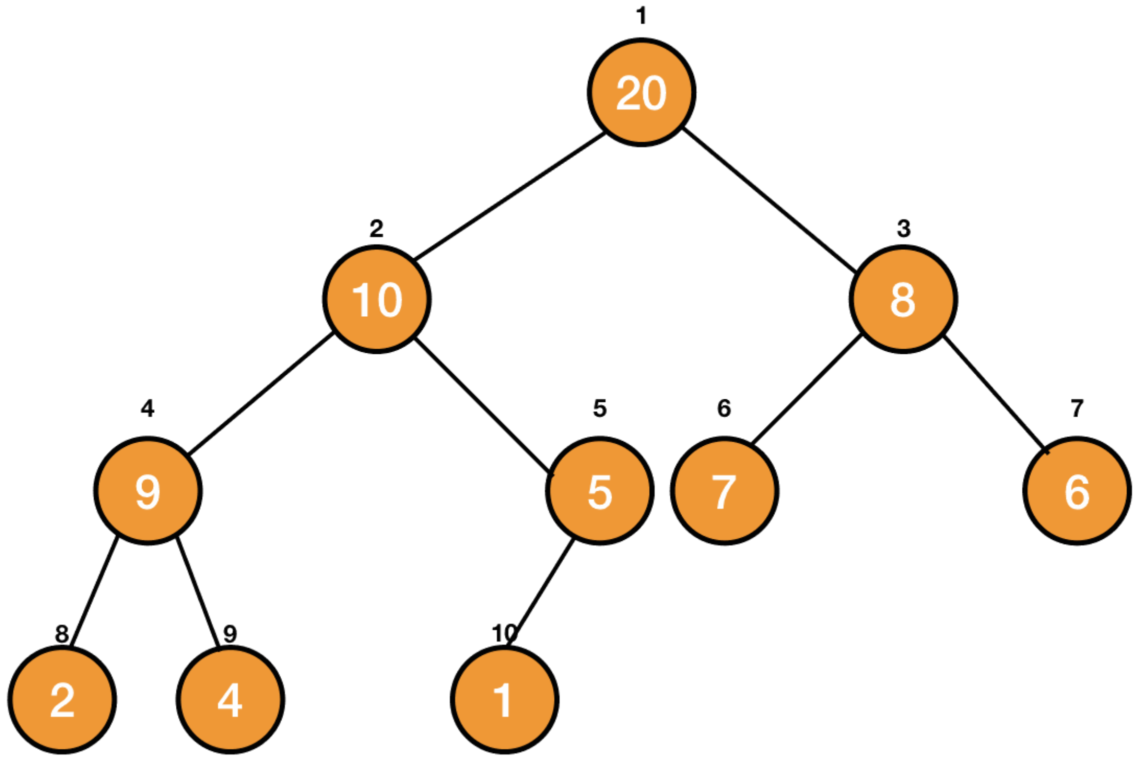
# Max-Heapify – step by step



## Max-Heapify – step by step



## Max-Heapify – step by step



| 20 | 10 | 8 | 9 | 5 | 7 | 6 | 2 | 4 | 1 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

# Max-Heapify Algorithm

```
MAX-HEAPIFY(A, i)
left = 2i
right = 2i + 1

// checking for largest among left, right and node i
largest = i
if left <= heap_size
  if (A[left] > A[largest])
    largest = left

if right <= heap_size
  if(A[right] > A[largest])
    largest = right

if largest != i //node is not the largest, we need to swap
  swap(A[i], A[largest])
  MAX-HEAPIFY(A, largest) // child after swapping might be violating max-heap property
```

## Max-Heapify Complexity

- Since the node on which we are applying Heapify is coming down and in the worst case, it may become a leaf. So, the worst-case running time will be the order of the height of the tree i.e.,$O(\log n)$.

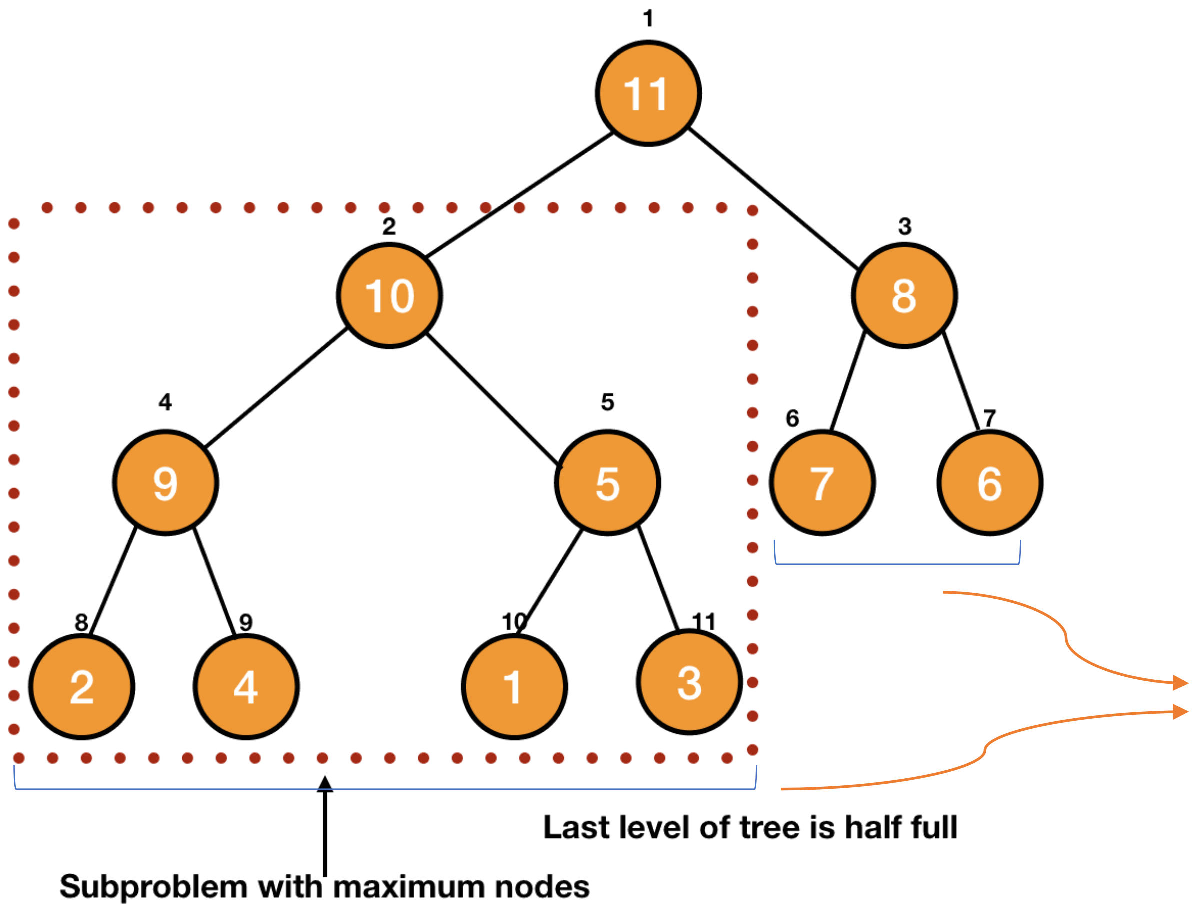
## Analysis of Heapify

- Although we have predicted the running time to be $O(\log n)$, let's see it mathematically.
- The calculations of the left, right and maximum elements are going to take $\Theta(1)$ time.
- Now, we are left with the calculation of the time that will be taken by $MAX - HEAPIFY(A, largest)$ and it will depend on the size of the input.

## Analysis of Heapify

- The tree is divided into two subtrees. $MAX - HEAPIFY$ is dependent on the size of the **tree** (or **subtree** in recursive calls).
- In the worst case, this size will be maximum. This will happen when the last level of the **tree** is half full.

# Analysis of Heapify

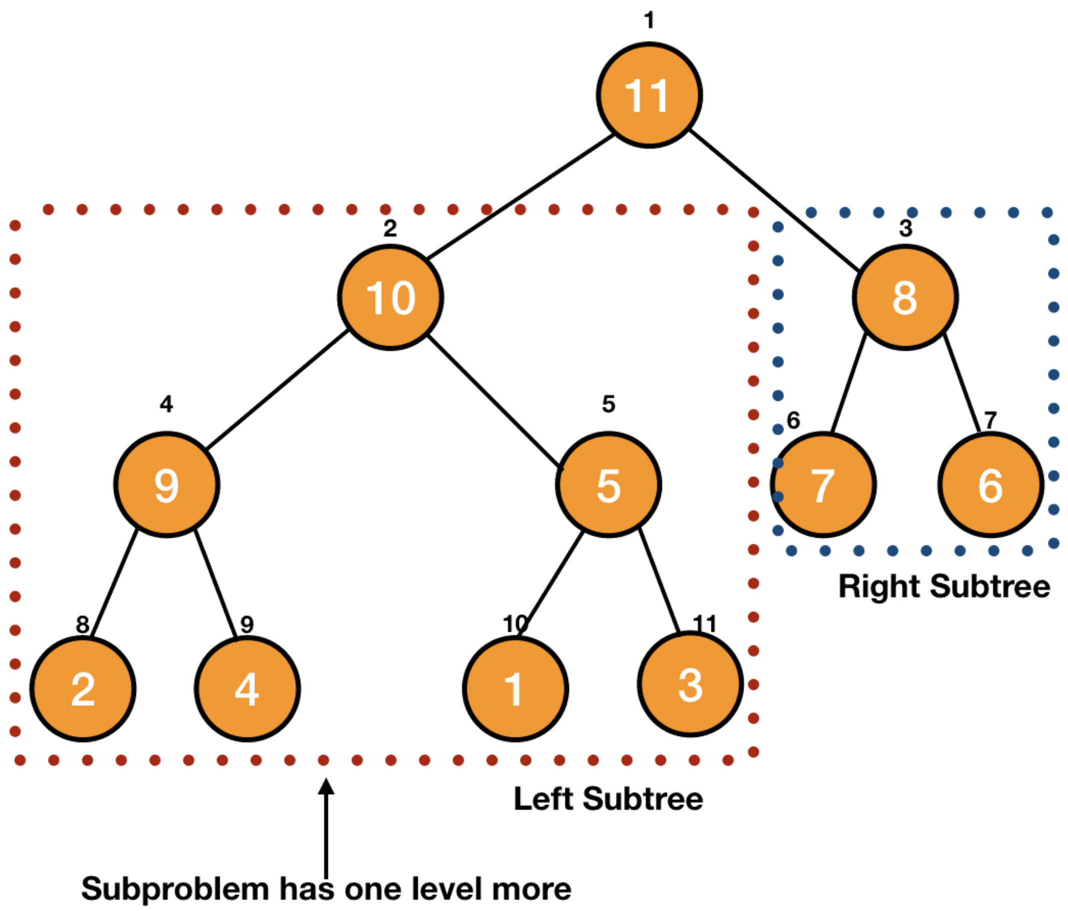

1
11
2
10
3
8
4
9
5
5
6
7
7
6
8
2
9
4
10
1
11
3
Last level of tree is half full
Subproblem with maximum nodes

The other half of the tree has 0 nodes. Thus the left subtree has 1 more level than the right subtree.

## Analysis of Heapify

- In this case, one of the subtrees will have one level more than the other one. This will maximize the number of nodes in the subtree for a fixed number of nodes $n$ in the complete binary tree.

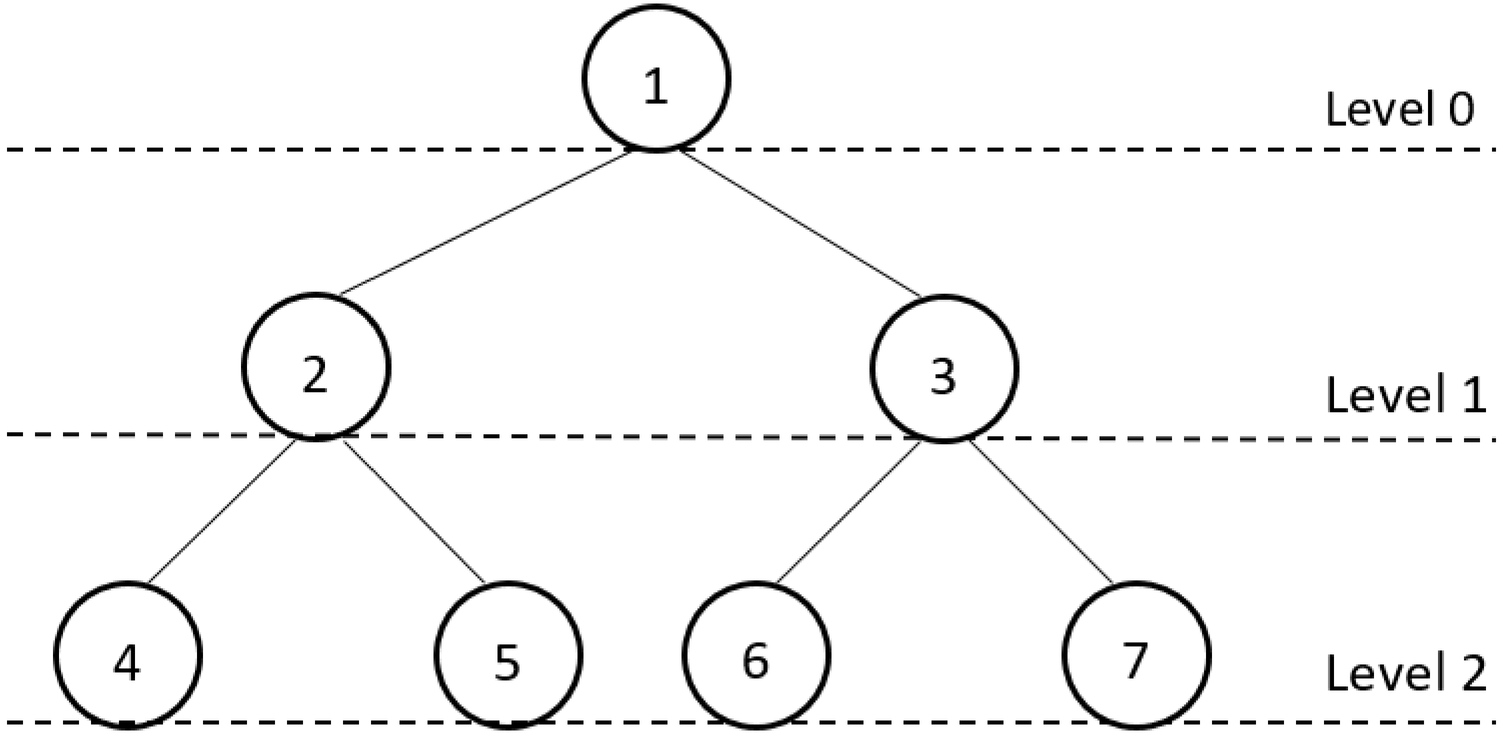
# Analysis of Heapify



## Maximum Number of Nodes

- To construct a binary tree of level n with the maximum number of nodes, we need to make sure all the internal nodes have two children. In addition, all the leaf nodes must be at the level n.
- For example, at level 0, we only have the root node. At level 1, we have 2 nodes that are the children of the root. Similarly, we have 4 nodes at level 2 who are the children of the nodes in level 1:

## Maximum Number of Nodes



## Maximum Number of Nodes

- Based on this observation, we can see that each level doubles the number of nodes from its previous level. This is because every internal node has two children. Therefore, the maximum number of nodes of a level $n$ binary tree is $1 + 2 + 4 + \ldots + 2^n = 2^{n+1} - 1$.
- We also call this type of binary tree a full binary tree.

## Maximum Number of Nodes

- We can conclude the maximum number of nodes with the following theorem:

$$Theorem: Let\ T\ be\ a\ binary\ tree\ with\ level\ n\ (n \geq 0). Then\ T\ contains\ at\ most\ 2^{n+1} - 1\ nodes.$$

- Note: each step from top to bottom is called as level of a tree. The level count starts with 0 and increments by 1 at each level or step.

## Analysis of Heapify

We know that a tree with $i$ levels has a total number of $2^{i+1} - 1$ nodes. Thus, if the right subtree has $i$ levels, it will have $2^{i+1} - 1$ nodes and the left subtree will have $i + 1$ levels and thus a total number of $2^{i+2} - 1$ nodes.

The total number of nodes in the tree:

- $2^{i+1} - 1 + 2^{i+2} - 1 + 1(root) = n$
- $2^{i+1} - 1 + 2^{i+2} = n$
- $2 * 2^{i} + 4 * 2^{i} = n + 1$
- $6 * 2^{i} = n + 1$
- $i = \log \frac{n+1}{6}$

## Analysis of Heapify

Now, the total number of nodes in the left subtree:

- $2^{i+2} - 1 = 4 * 2^{i} - 1 = \frac{4(n+1)}{6} - 1 = \frac{2(n+1)}{3} - 1 = \frac{2n}{3} - \frac{1}{3}$
- $\frac{2n}{3} - \frac{1}{3} \leq \frac{2n}{3}$

## Analysis of Heapify

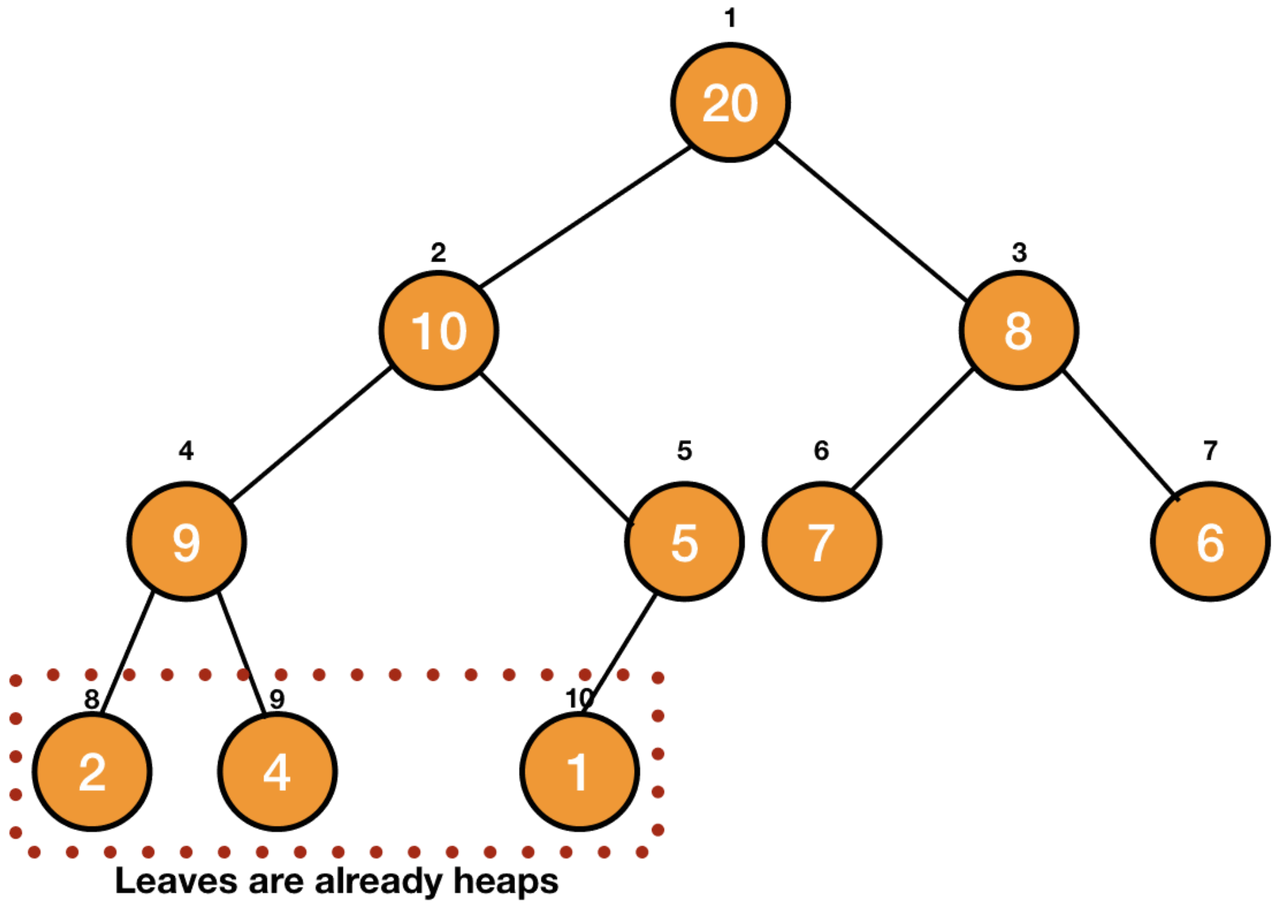
We can now use $\frac{2n}{3}$ as its upper bound and write the recurrence equation as $T(n) \leq T(2n3) + \Theta(1)$

By using Master's theorem, we can easily find out the running time of the algorithm to be $O(\log n)$.

## Build a Heap

- We are left with one final task, to make a heap by the array provided to us. We know that Heapify when applied to a node whose children are heaps, makes the node also a heap. The leaves of a tree don't have any child, so they follow the property of a heap and are already heap.

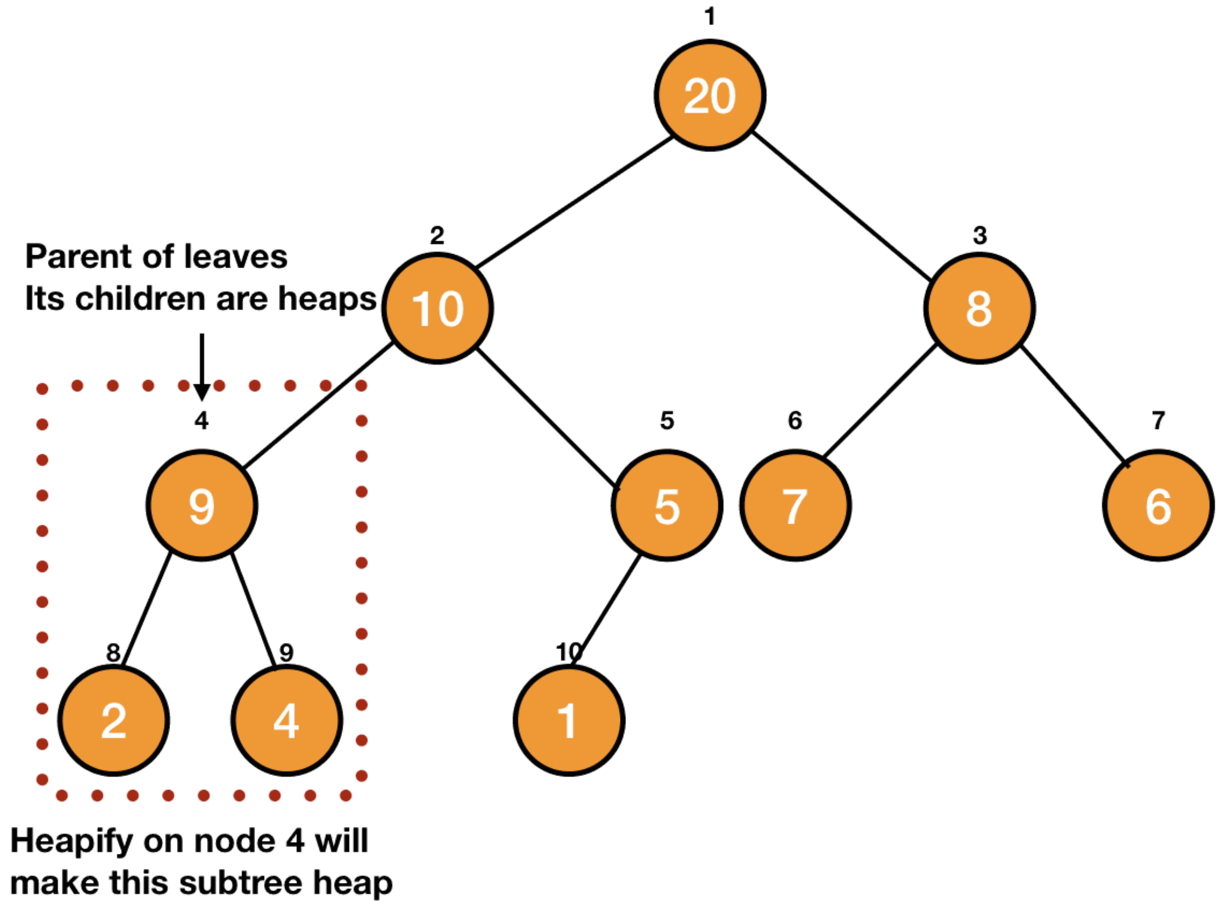
# Build a Heap



## Build a Heap

- We can implement the Heapify operation on the parent of these leaves to make them heaps.

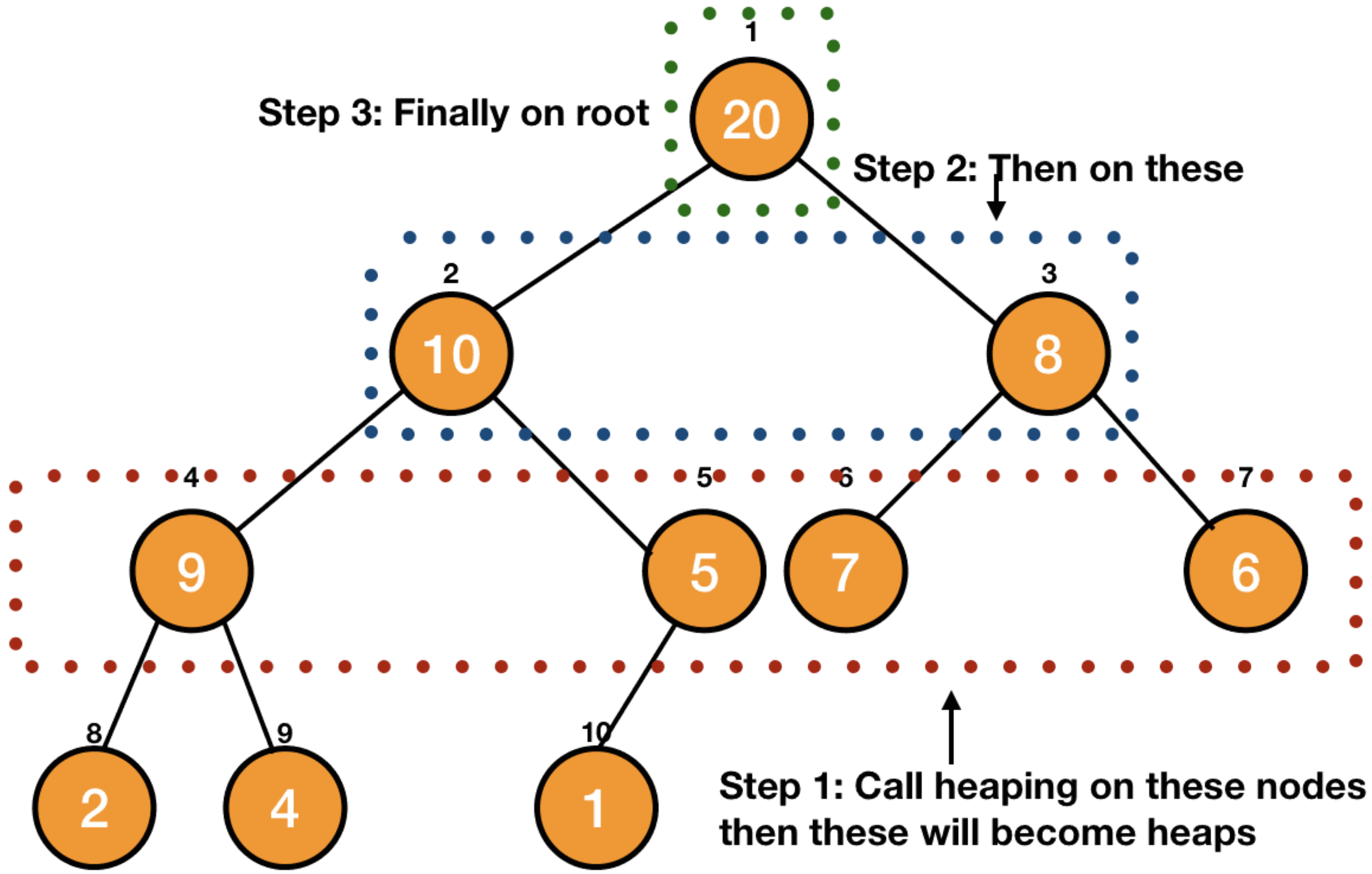
## Build a Heap


1
20
2
10
3
8
4
9
5
5
6
7
7
6
8
2
9
4
10
1
Parent of leaves
Its children are heaps
Heapify on node 4 will
make this subtree heap

## Build a Heap

- We can simply iterate up to root and use the Heapify operation to make the entire tree a heap.

# Build a Heap


Step 3: Finally on root
Step 2: Then on these
Step 1: Call heaping on these nodes then these will become heaps
1
20
2
10
3
8
4
9
5
5
6
7
7
6
8
2
9
4
10
1

# Build a Heap – Algorithm

- We simply have to iterate from the parent of the leaves to the root of the tree to call Heapify on each node. For this, we need to find the leaves of the tree. The nodes from $\lceil\frac{n}{2}\rceil + 1$ to $n$ are leaves.
- We can easily check this because $2 * i = 2 * (\lceil\frac{n}{2}\rceil + 1) = n + 2$ which is outside the heap and thus, this node doesn't have any child, so it is a leaf. Thus, we can make our iteration from $\lceil\frac{n}{2}\rceil$ to root and call the Heapify operation.

## Build a Heap – Algorithm

```
BUILD-HEAP(A):
for i in floor(A.length/2) downto 1
    MAX-HEAPIFY(A, i)
```

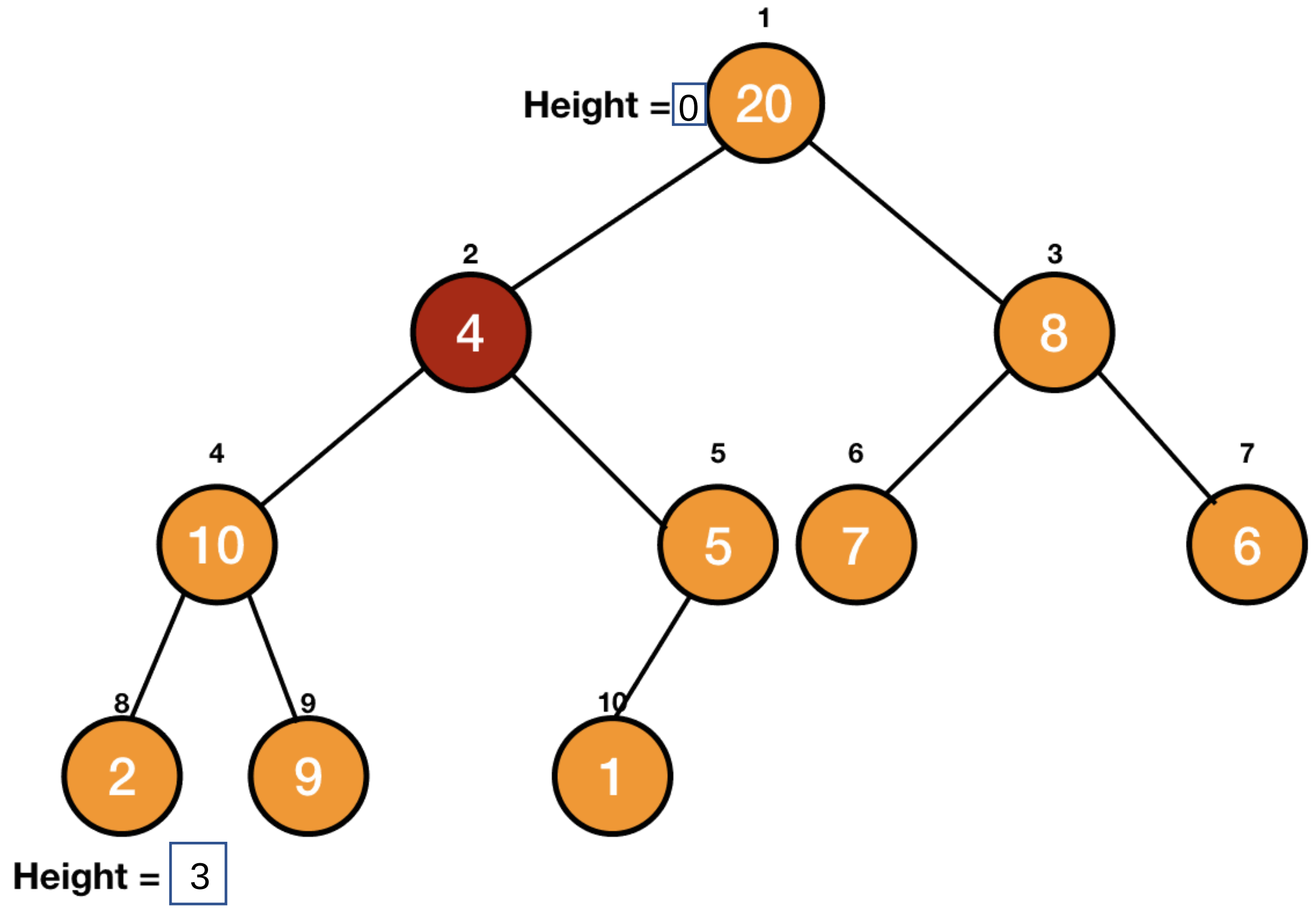
# Analysis of Build-Heap

- We know that Heapify takes $O(\log n)$ time and there are $O(n)$ such calls. Thus a total of $O(n \log n)$ time.
- This gives us an upper bound for our operation but we can reduce this upper bound and get a more precise running time of $O(n)$.

# A More Precise Analysis

- We know that the Heapify makes a node travel down the tree, so it will take $O(h)$ time, where h is the height of the node.
- Definition: The maximum distance of any node from the **root**. If a tree has only **one** node (the root), the height is **zero**.

## A More Precise Analysis



# A More Precise Analysis

We also know that the height of a node is $O(\log n)$, where $n$ is the number of nodes in the subtree.

Also, the maximum number of nodes with height h is $\lceil\frac{n}{2^{h+1}}\rceil$(You can prove it by induction).

So, the total time taken by the Heapify function for all the nodes at height $h\ =\ O(h) * \lceil\frac{n}{2^{h+1}}\rceil$(height of the nodes*number of nodes).

Now, this height will change from $0$ to $\lfloor\log n\rfloor$.

## A More Precise Analysis

Thus, the total time taken for all the nodes =

- $\sum_{h=0}^{\lfloor \log n \rfloor} (O(h) * \left\lceil \frac{n}{2^{h+1}} \right\rceil) =$
- $O\left(n * \sum_{h=0}^{\lfloor \log n \rfloor} \left( \left\lceil \frac{h}{2 * 2^h} \right\rceil \right) \right) =$
- $O\left(n * \sum_{h=0}^{\lfloor \log n \rfloor} \left( \left\lceil \frac{h}{2^h} \right\rceil \right) \right)$

## A More Precise Analysis

Taking the term $\sum_{h=0}^{\lfloor \log n \rfloor} \left( \left\lceil \frac{h}{2^h} \right\rceil \right)$.

- $\sum_{h=0}^{\lfloor \log n \rfloor} \left( \left\lceil \frac{h}{2^h} \right\rceil \right) < \sum_{h=0}^{\infty} \left( \left\lceil \frac{h}{2^h} \right\rceil \right)$
- $Let\ S = \sum_{h=0}^{\infty} \left( \left\lceil \frac{h}{2^h} \right\rceil \right)$
- $or, S = 1 + \frac{1}{2} + \frac{2}{2^2} + \frac{3}{2^3} + \cdots$
- $2S = 2 + 1 + \frac{2}{2} + \frac{3}{2^2} + \frac{4}{2^3} + \cdots$
- $2S - S,$

## A More Precise Analysis

- $2S = 2 + 1 + \frac{2}{2} + \frac{3}{2^2} + \frac{4}{2^3} + \cdots$
- $S = 0 + 1 + \frac{1}{2} + \frac{2}{2^2} + \frac{3}{2^3} + \cdots$
- $2S - S = S = 2 + 0 + \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots$

## A More Precise Analysis

- The above equation is an infinite G.P. as $\frac{1}{2}$ as the first term as well as the common ratio.

$$S = 2 + \frac{\frac{1}{2}}{1 - \frac{1}{2}} = 2$$

## A More Precise Analysis

- So, $S = \sum_{h=0}^{\infty}\left(\left\lceil\frac{h}{2^h}\right\rceil\right) = 2.$
- Putting this value in $O\left(n * \sum_{h=0}^{\lfloor \log n \rfloor}\left(\left\lceil\frac{h}{2^h}\right\rceil\right)\right).$
- Running Time = $O(2 \ * n) = O(n).$
- So, we can make a heap from an array in a linear time.

## Codes in C – Build Max Heap

```c
void build_max_heap(int A[]) {
  int i;
  for(i=heap_size/2; i>=1; i--) {
    max_heapify(A, i);
  }
}
```

## Codes in C – Max Heapify

```c
void max_heapify(int A[], int index) {
  int left_child_index = get_left_child(A, index);
  int right_child_index = get_right_child(A, index);

  // finding largest among index, left child and right child
  int largest = index;

  if ((left_child_index <= heap_size) && (left_child_index>0)) {
    if (A[left_child_index] > A[largest]) {
      largest = left_child_index;
    }
  }

  if ((right_child_index <= heap_size && (right_child_index>0))) {
    if (A[right_child_index] > A[largest]) {
      largest = right_child_index;
    }
  }
```

## Codes in C – Max Heapify

```
  // largest is not the node, node is not a heap
  if (largest != index) {
    swap(&A[index], &A[largest]);
    max_heapify(A, largest);
  }
}
```

## Codes in C – get right/left child

```c
//function to get right child of a node of a tree
int get_right_child(int A[], int index) {
  if((((2*index)+1) < tree_array_size) && (index >= 1))
    return (2*index)+1;
  return -1;
}

//function to get left child of a node of a tree
int get_left_child(int A[], int index) {
    if(((2*index) < tree_array_size) && (index >= 1))
        return 2*index;
    return -1;
}
```

## Codes in C – swap/get parent

```c
void swap( int *a, int *b ) {
  int t;
  t = *a;
  *a = *b;
  *b = t;
}

//function to get the parent of a node of a tree
int get_parent(int A[], int index) {
  if ((index > 1) && (index < tree_array_size)) {
    return index/2;
  }
  return -1;
}
```