Data Structure & Algorithms

Heap

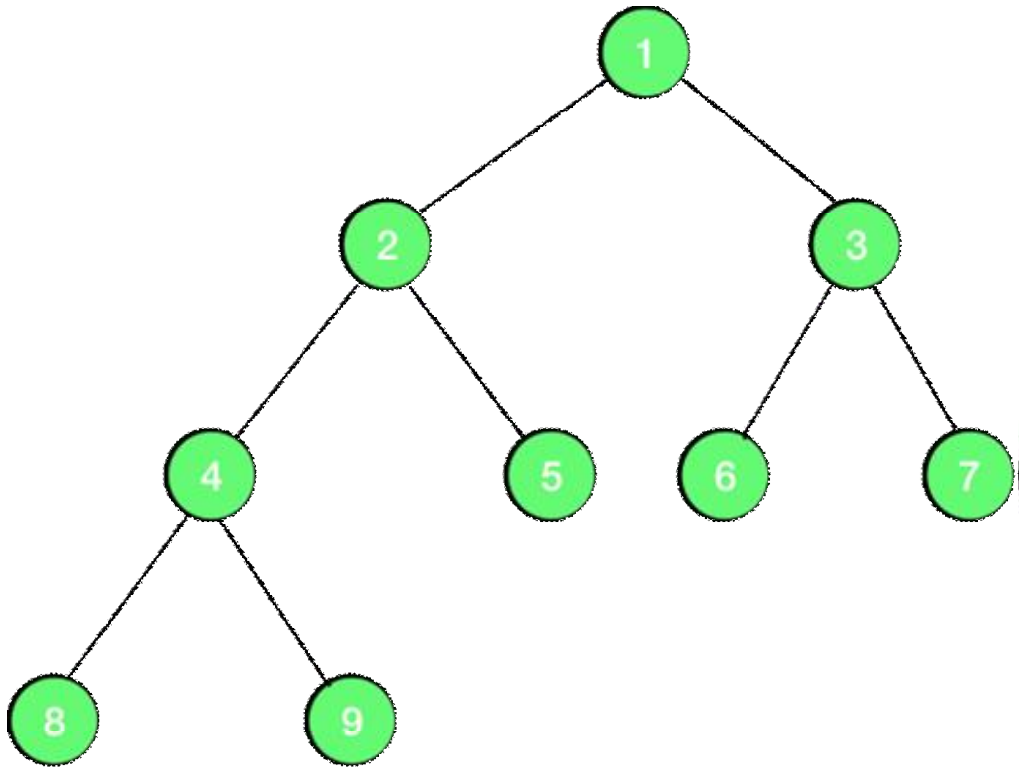
Heap

- 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

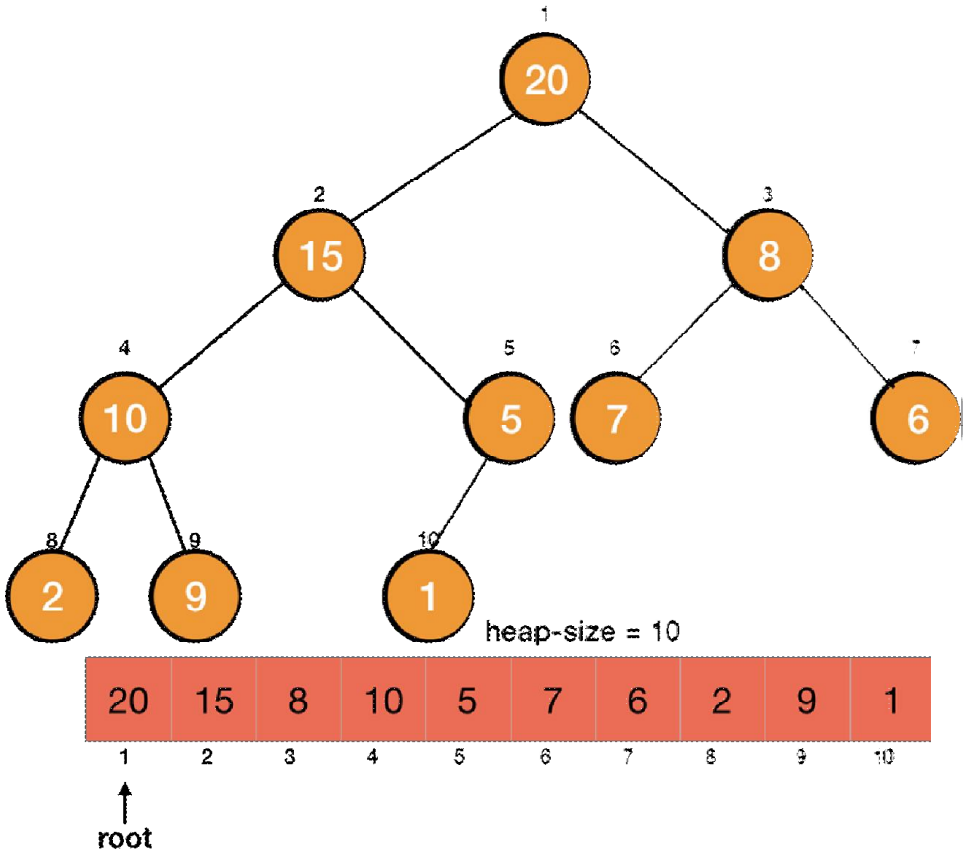
- A binary tree which is completely filled with a possible exception at the bottom level i.e., the last level may not be completely filled and the bottom level is filled from left to right.

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.,

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

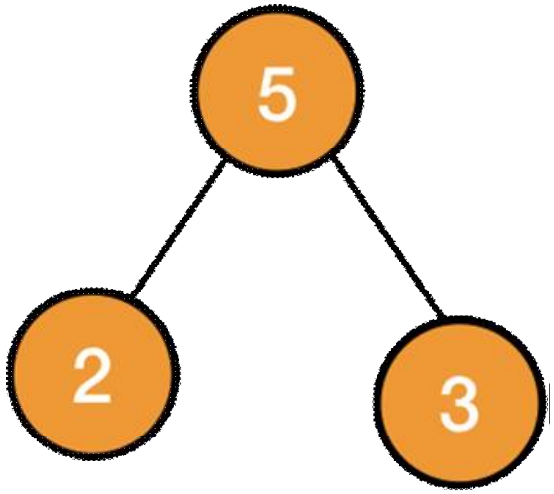
Properties of a Heap

- 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.)

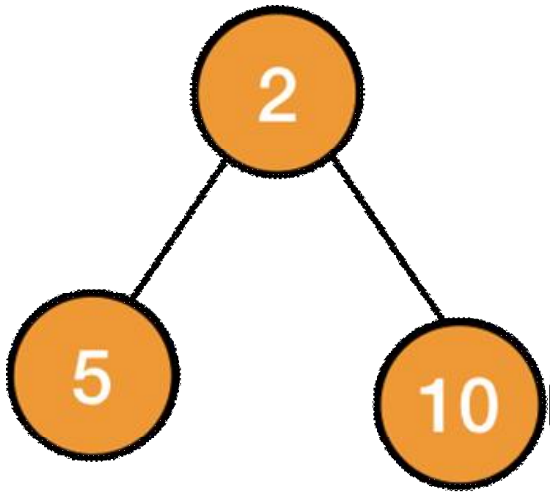


Max Heap

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

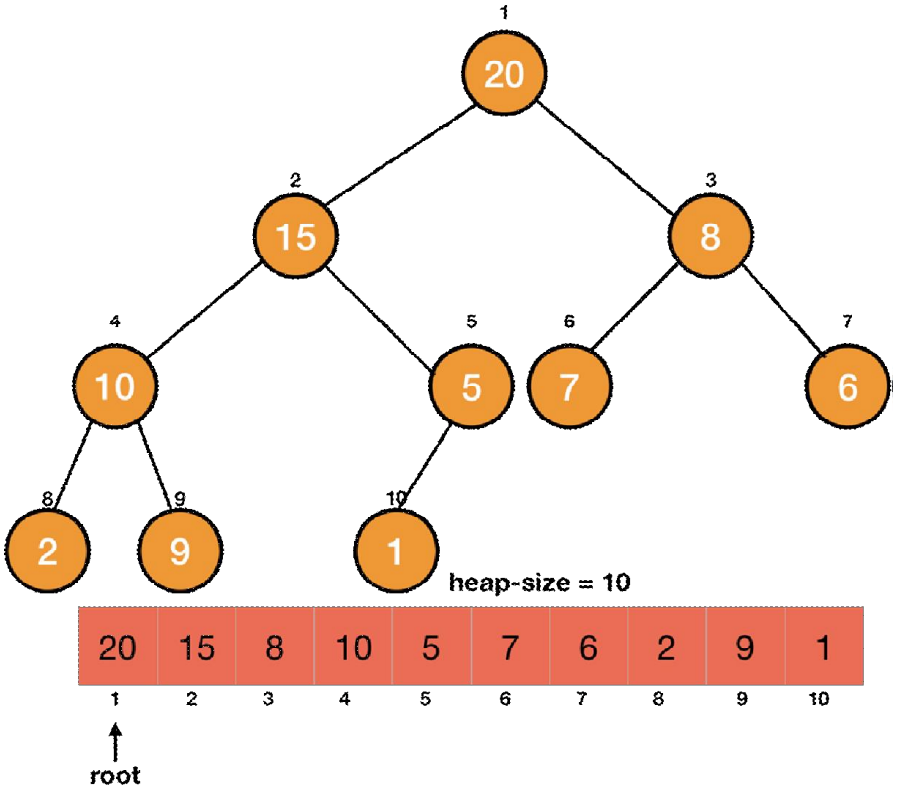
Max Heap & Min Heap

- 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

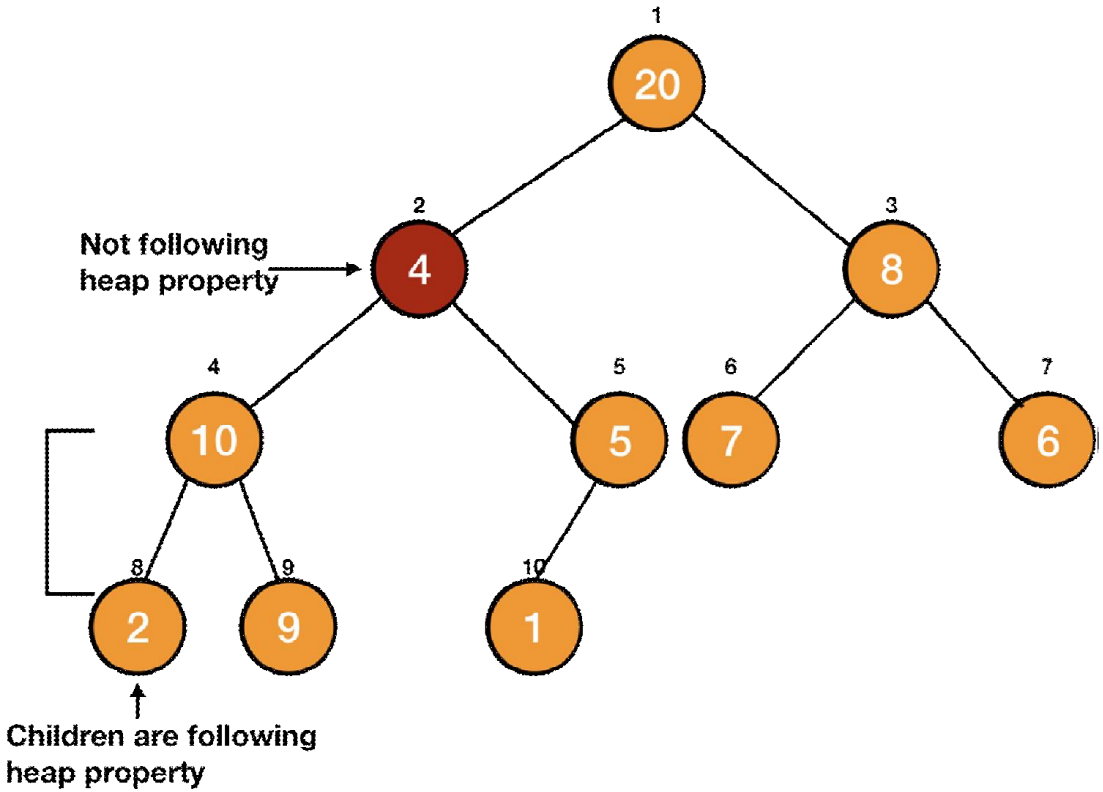


Heapify

- 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 heap property), but the node itself may be violating the property.

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

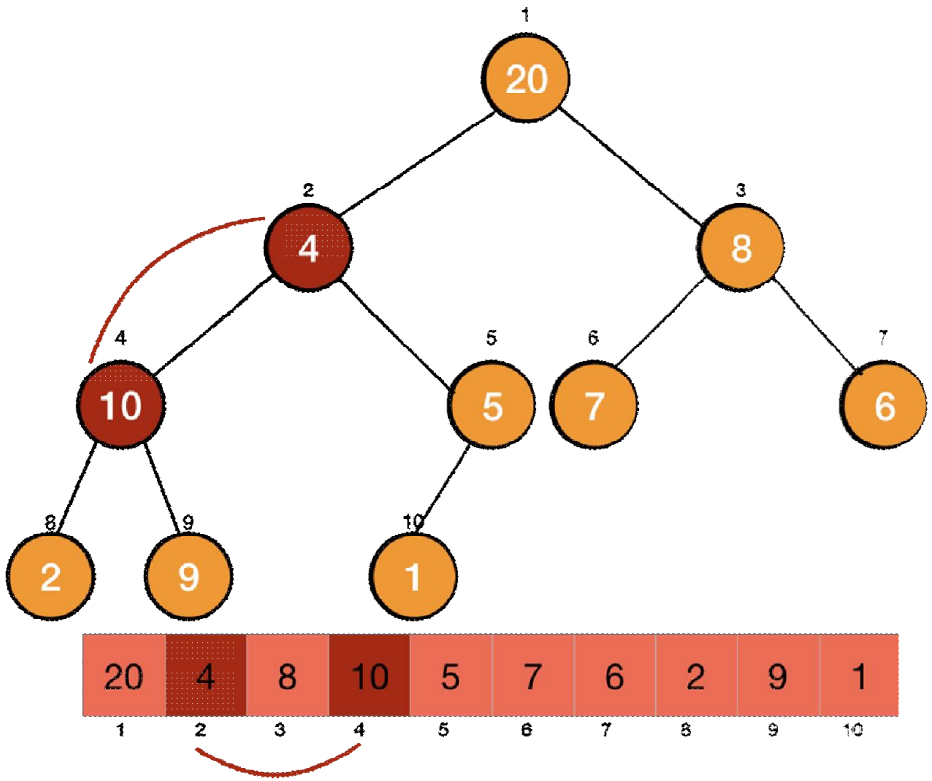
Max-Heapify – Example



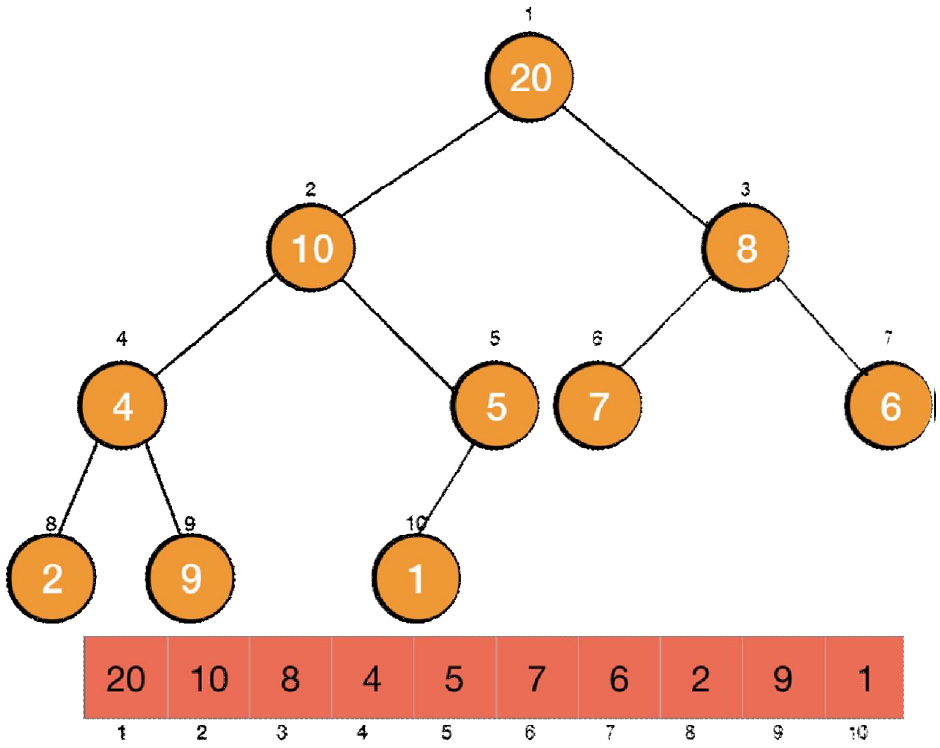
Max-Heapify (cont.)

- We simply make the node travel down the tree until the property of the heap is satisfied.
- The steps are illustrated on a max-heap as follows:

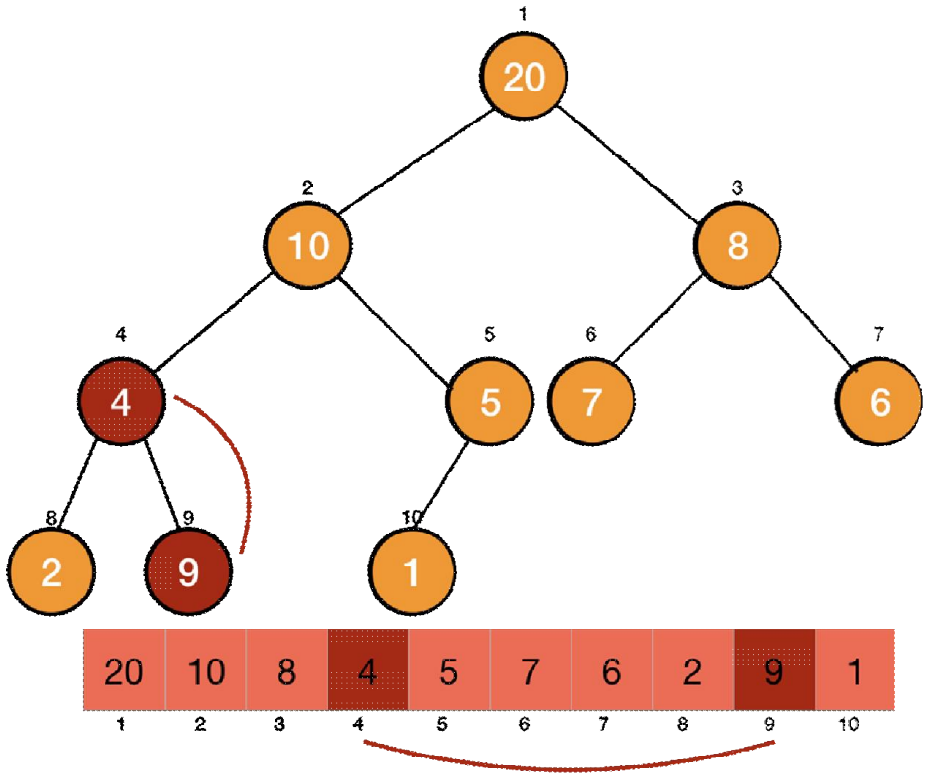
Max-Heapify – step by step (1)



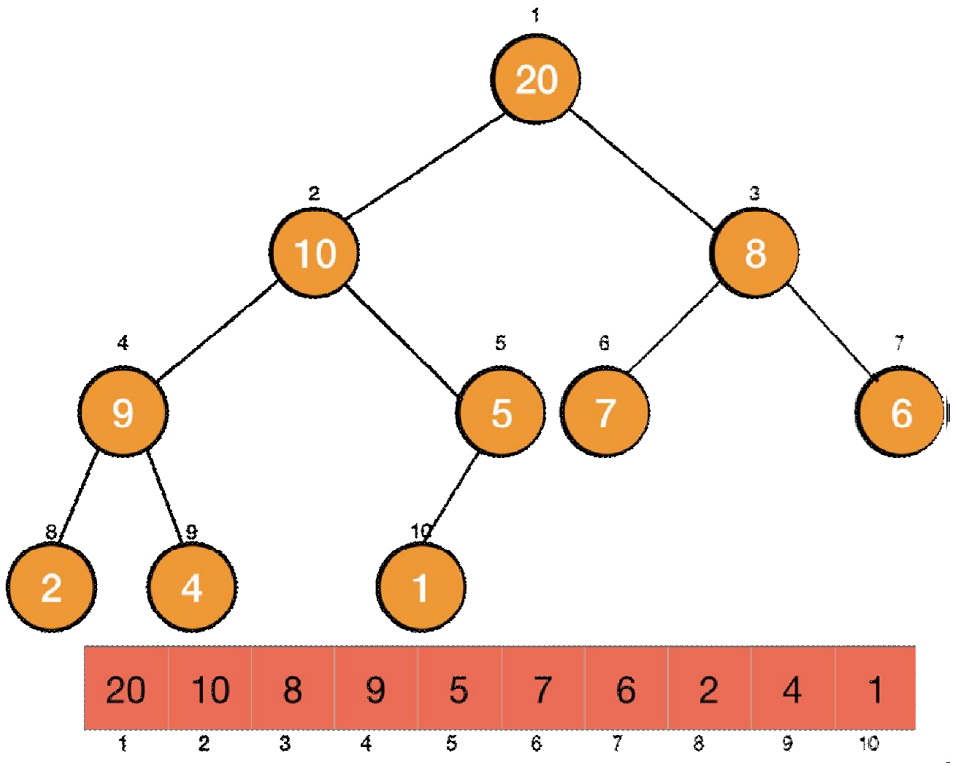
Max-Heapify – step by step (2)



Max-Heapify – step by step (4)



Max-Heapify – Done



Max-Heapify (pseudocode)

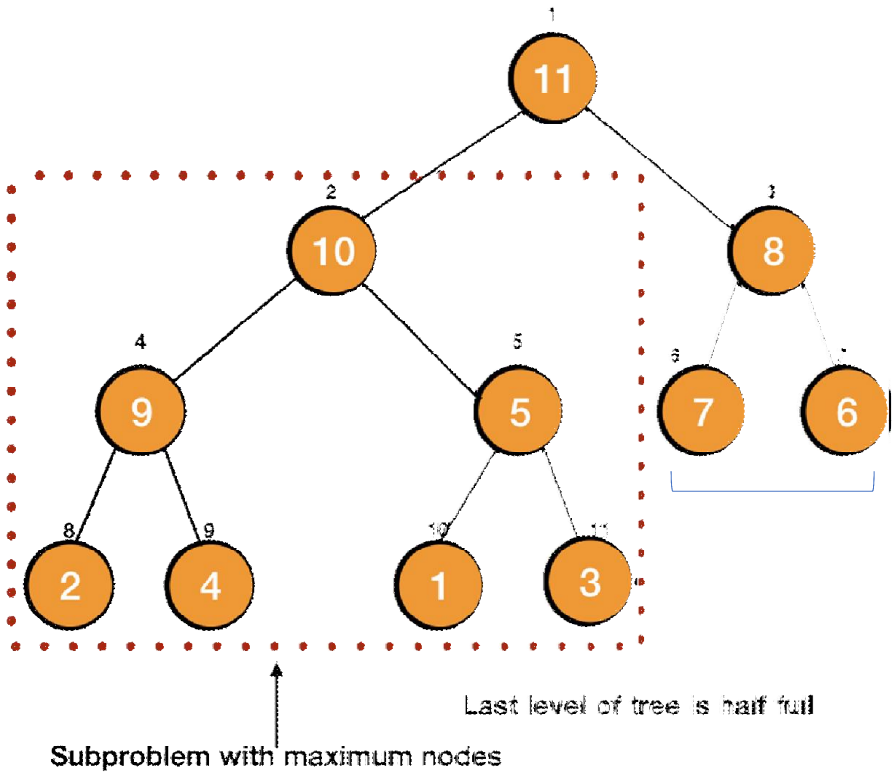
```
MAX-HEAPIFY(A, i)
left = 2i
right = 2i + 1

// 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

- The running time of MAX-HEAPIFY on a subtree of size n rooted at a given node i is the $\Theta(1)$ time to fix up the relationships among the elements $A[i]$, $A[left]$ and $A[right]$, plus the time to run MAX-HEAPIFY on a subtree rooted at one of the children of node i.
- The children's subtrees each have size at most $2n/3$—the worst case occurs when the bottom level of the tree is exactly half full—and therefore we can describe the running time of MAX-HEAPIFY by the recurrence:
- $T(n) \leq T\left(\frac{2n}{3}\right) + \Theta(1)$  $T(n) = O(\log n)$

Max Heapify Time Complexity - Proof

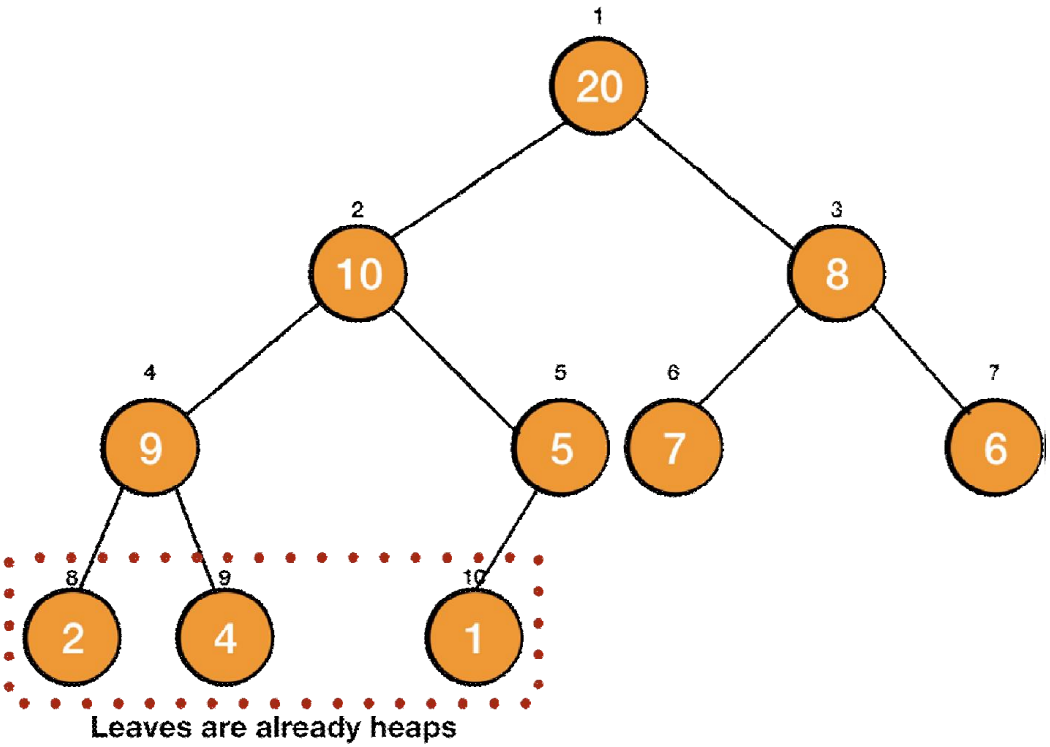
Analysis of Heapify



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.

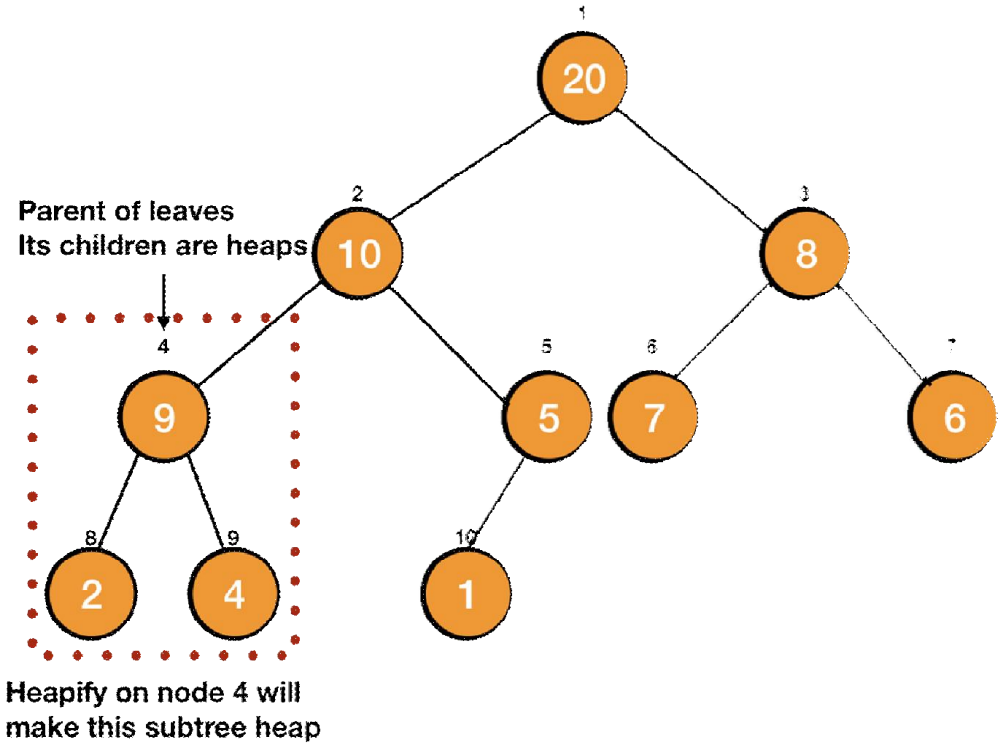
Build a Heap (cont.)



Build a Heap (cont.)

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

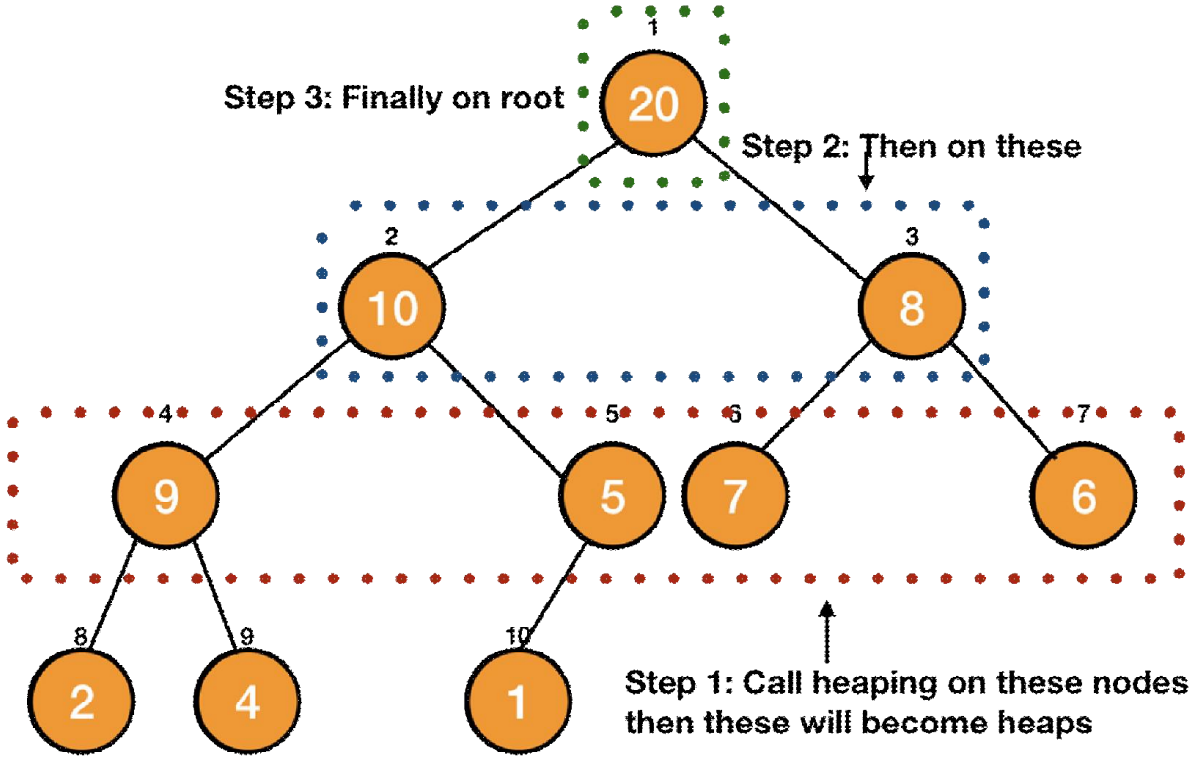
Build a Heap (cont.)



Build a Heap (cont.)

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

Build a Heap (cont.)



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 $\lfloor \frac{n}{2} \rfloor$ down to 1 (the root) and call the Heapify operation.

Build a Heap – Algorithm (cont.)

```
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 $O(n)$.

How?

✓Our analysis relies on the properties that an n-element heap has height $\lfloor \log n \rfloor$ and at most $\lceil n/2^{h+1} \rceil$ nodes of height h.

Build Heap Time Complexity - Proof

Codes in C – Build Max Heap

```
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 (cont.)

```
  // 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;
}
```