Algorithms & Data Structures I: Heap

Mikhail Anukhin

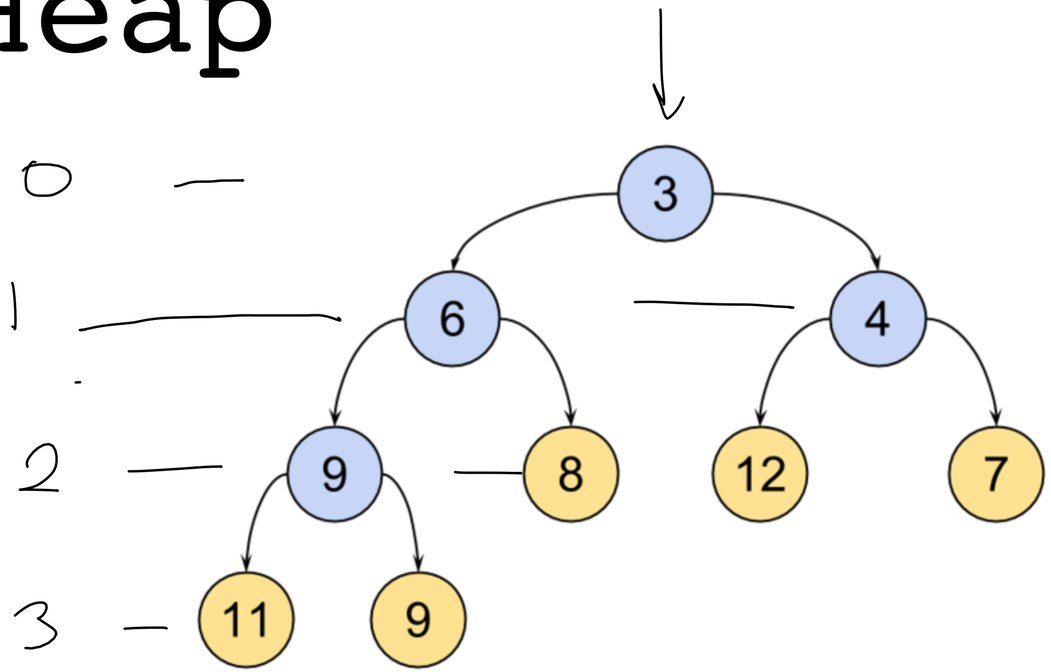
Today's Topics

- Heap. Heap property
- How to build a heap efficiently
- Heap Sort

Binary Heap

Heap is a binary tree that satisfy the following properties:

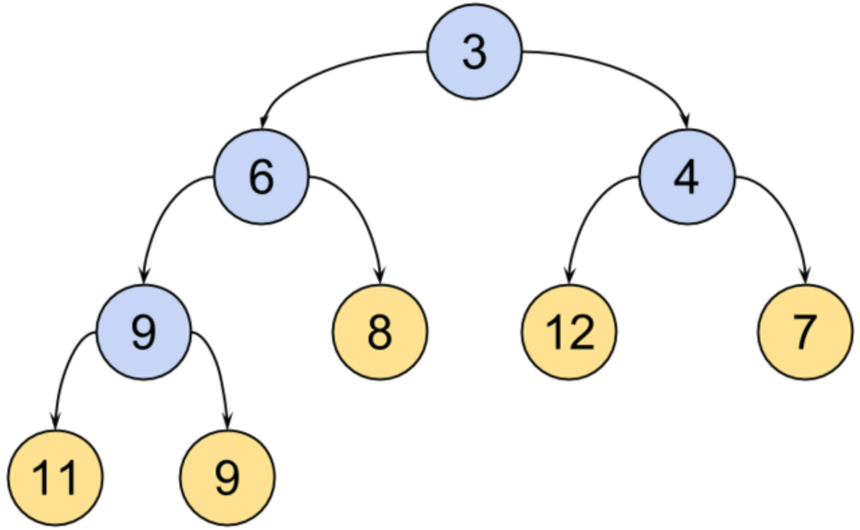
1. **The value of a node is ≤ than the values of its children**
2. i-th tree layer has 2^i nodes, except the last one. Layers enumerated from 0
3. The last layer filled from left to right

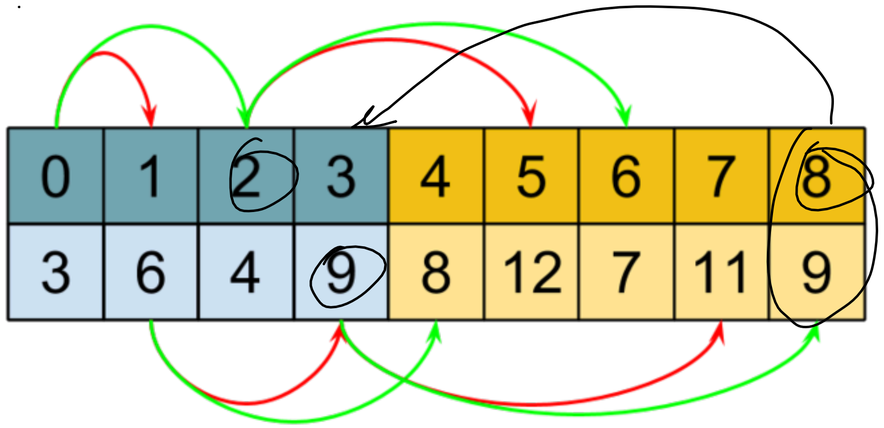


Visualizing an Array as a Tree

1. **Root**: first element in the array, corresponding to index = 0
2. **Parent(i)** = RoundDown(i / 2)
3. **Left(i)** = 2i + 1
4. **Right(i)** = 2i + 1

Parent(8) = (8 - 1) / 2 = 3





Min Heap vs Max Heap

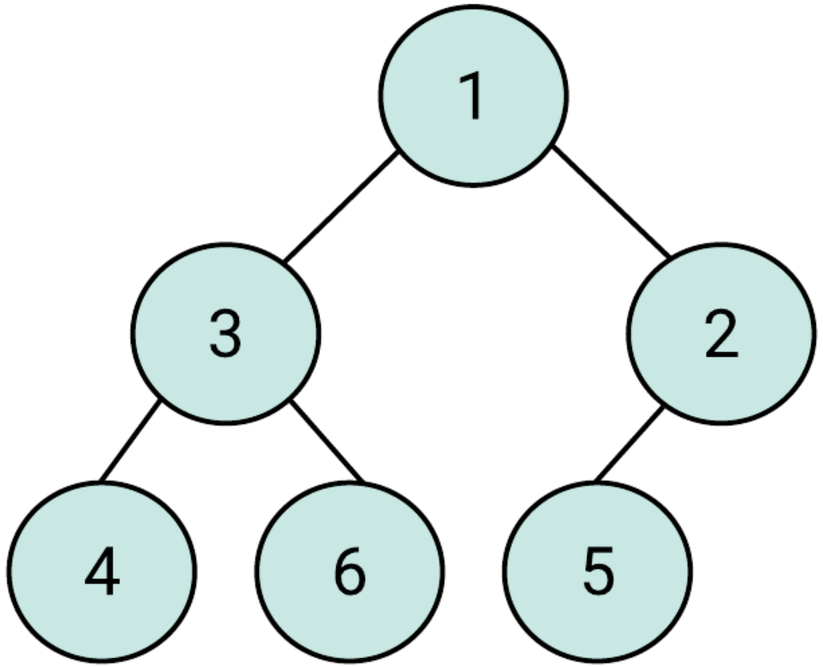
Min Heap is a binary tree that satisfy the following properties:

1. **The value of a node is ≤ than the values of its children**
2. i-th tree layer has 2^i nodes, except the last one. Layers enumerated from 0
3. The last layer filled from left to right

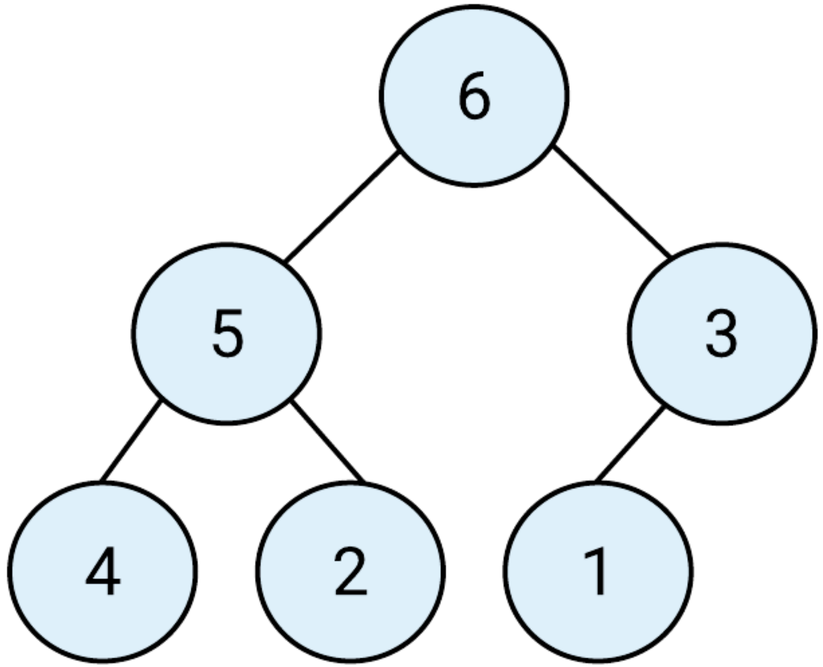
Max Heap is a binary tree that satisfy the following properties:

1. **The value of a node is ≥ than the values of its children**
2. i-th tree layer has 2^i nodes, except the last one. Layers enumerated from 0
3. The last layer filled from left to right

Min Heap vs Max Heap



Min heap



Max Heap

Heap Operations

SiftDown: correct a single violation of the heap property occurring at the root of a subtree in O(log n)

SiftUp: correct a single violation of the heap property occurring at a leaf of a subtree in O(log n)

MakeHeap: produce a min-heap from an unordered array in O(n)

Insert: insert an element in a heap in O(log n)

ExtractMin: Find and delete the min element from a heap in O(log n)

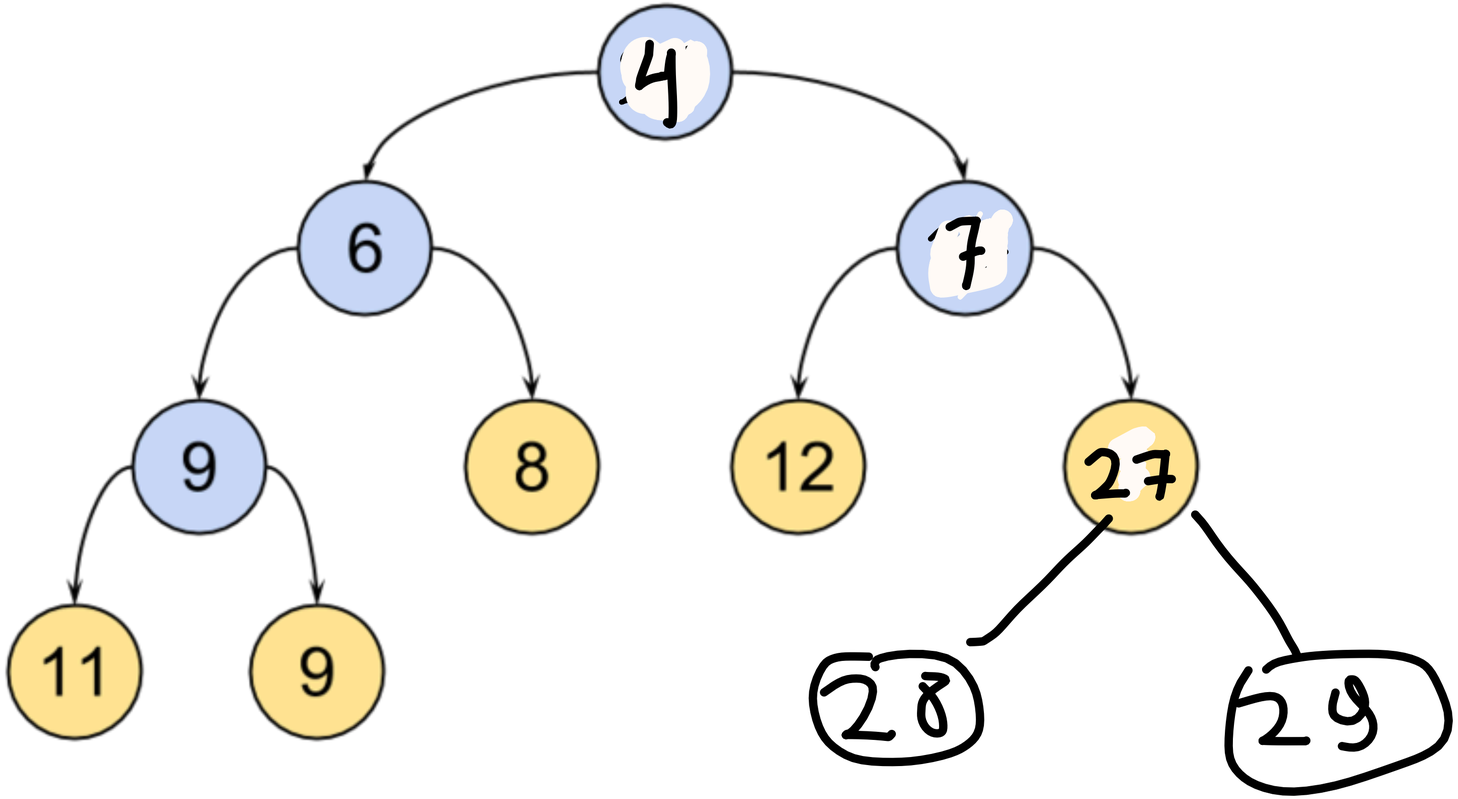
Sift Down

SiftDown: correct a single violation of the heap property occurring at the root of a subtree in O(log n)

Assume that the trees rooted at left(i) and right(i) are min-heaps, but element A[i] violates the min-heap property;

The goal is to correct the violation. Do this by trickling element A[i] down the tree, making the subtree rooted at index i a min-heap.

Sift Down Example



Sift Down

```
function SiftDown(i : int):
    while 2 * i + 1 < a.heapSize // heapSize -
number of elements in heap
        left = 2 * i + 1 // left son
        right = 2 * i + 2 // right son
        j = left
        if right < a.heapSize and a[right] < a[left]
            j = right
        if a[i] <= a[j]
            break
        swap(a[i], a[j])
        i = j
```

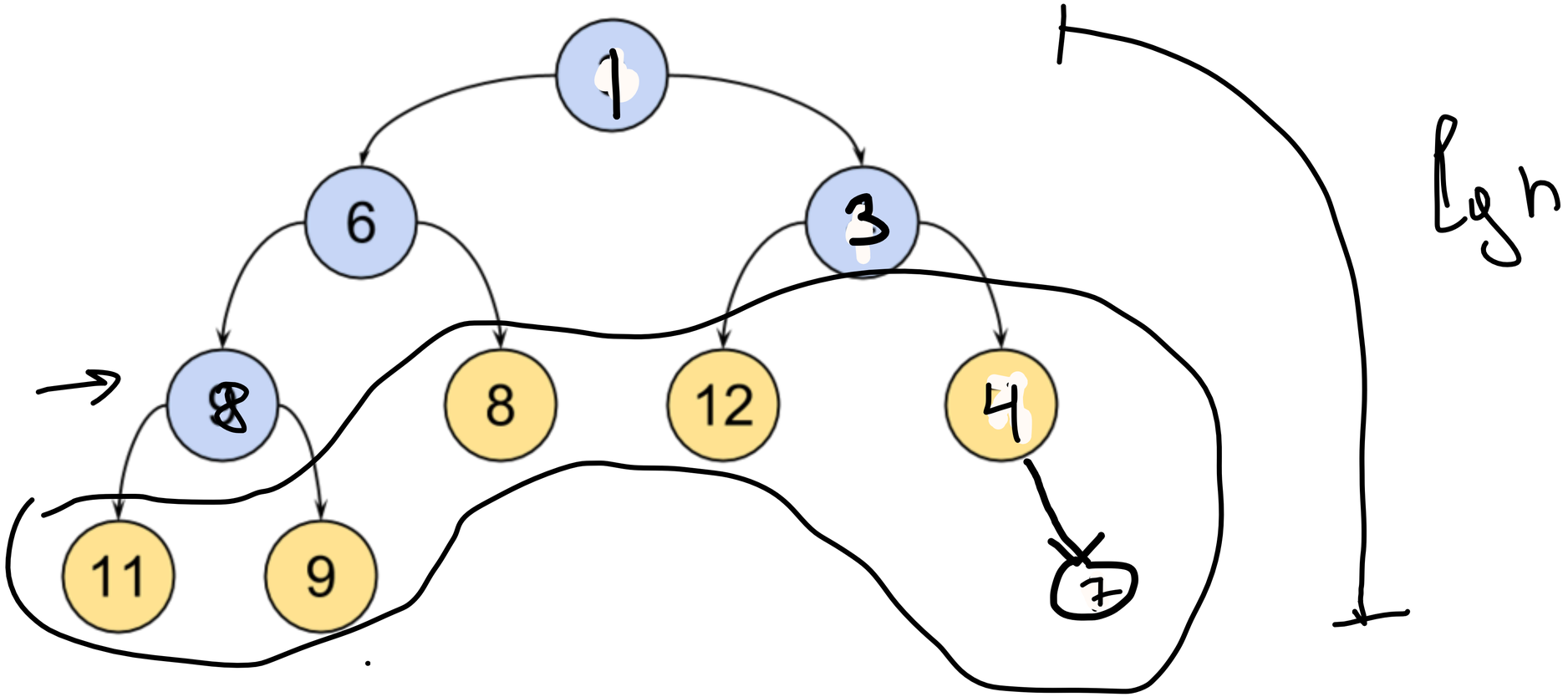
Sift Up

SiftUp: correct a single violation of the heap property occurring at a leaf of a subtree in O(log n)

Assume that a leaf violates the min-heap property;

The goal is to correct the violation. Do this by trickling element A[i] up the tree.

Sift Up



Sift Up

```
function SiftUp(i : int):
    while a[i] < a[(i - 1) / 2]
        swap(a[i], a[(i - 1) / 2])
        i = (i - 1) / 2
```

Make Heap

MakeHeap: produce a min-heap from an unordered array in O(n)

Convert an array A[1..n] into a min heap

Observation: Elements A[roundDown(n / 2) + 1, … , n] are leaves cause $2i > n$, for all $i \geq$ roundDown(n / 2) + 1

That means heap property may only be violated at nodes 1 … roundDown$(h-1/2)$ of the tree. So we need to fix violation at half of the elements.

Make Heap

```
function MakeHeap():
    for i = (a.heapSize - 1) / 2 downto 0
        SiftDown(i)
```

Make Heap Running time

Observe however that MakeHeap takes O(1) for time for nodes that are one level above the leaves, and in general, O(l) for the nodes that are l levels above the leaves.

We have n/4 nodes with level 1, n/8 with level 2, and so on till we have one root node that is lg n levels above the leaves.

Total amount of work in the for loop can be summed as:

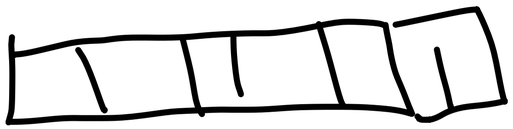
$T(n) = n/4\ (1\ c) + n/8\ (2\ c) + n/16\ (3\ c) + \ldots + 1\ (\lg n\ c)$

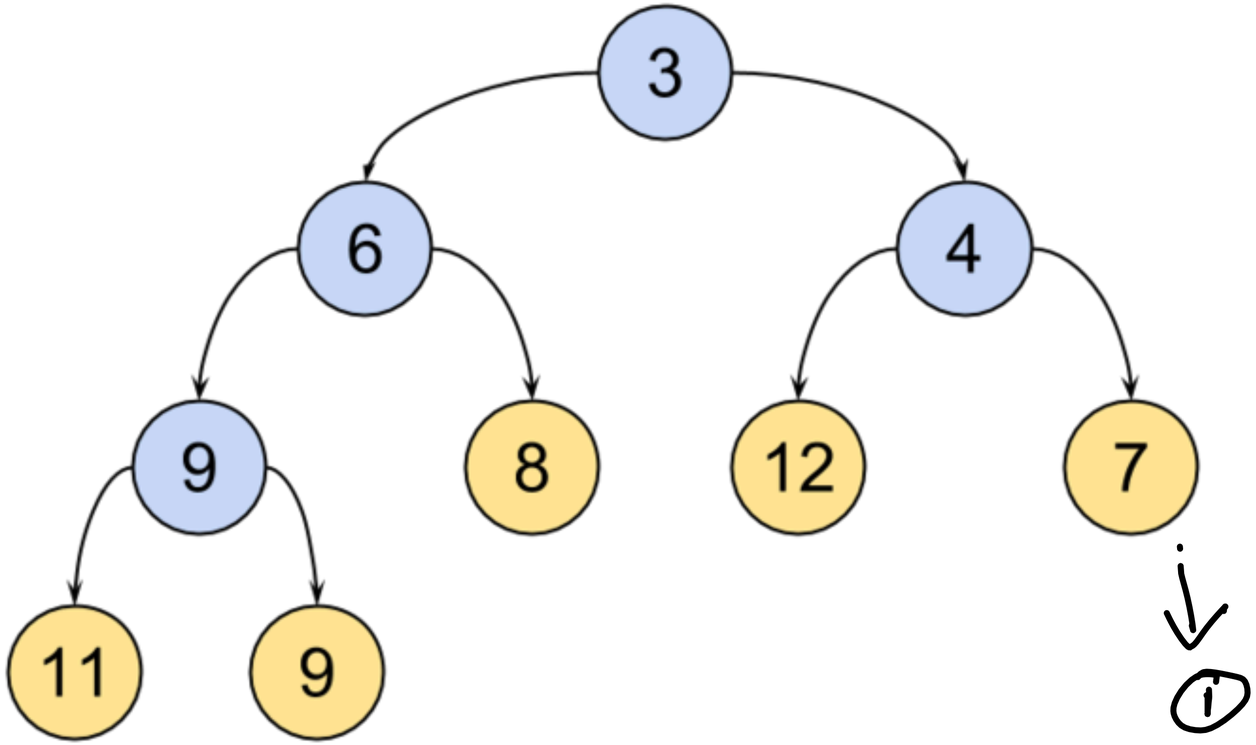
Setting $n/4 = 2^k$ and simplifying we get:

$T(n) = c\ 2^k\ (\ 1/2^0 + 2/2^1 + 3/2^2 + \ldots (k+1)/2^k\ )$ = **O(n)**

Insert

Insert: insert an element in a heap in O(log n)





1

SiftUp(n-1)

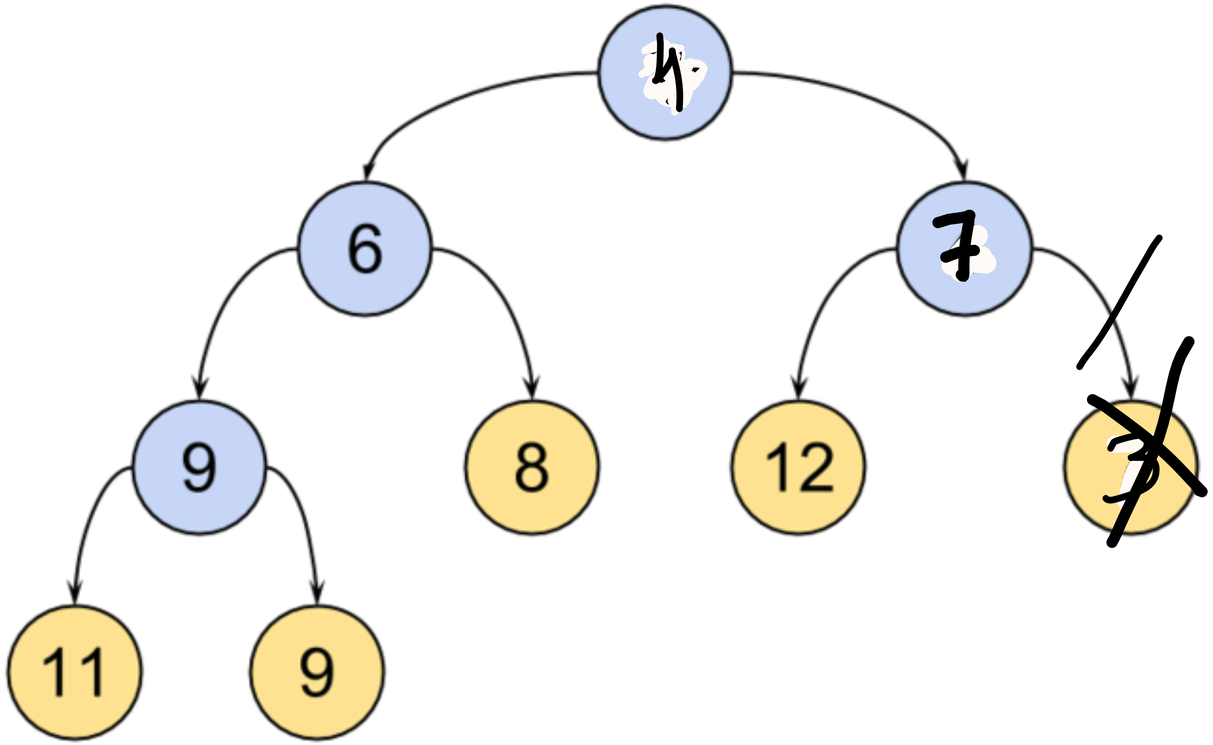
1

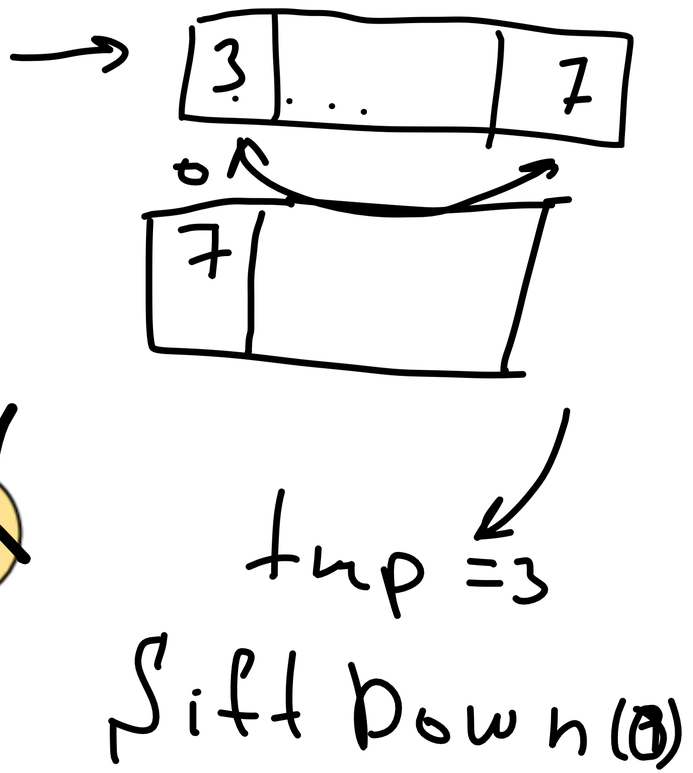
Insert

```
function Insert(value : int):
    a.pushBack(value)
    siftUp(a.heapSize - 1)
```

Extract Min

ExtractMin: Find and delete the min element from a heap in O(log n)





Extract Min

```
int extractMin():
    int min = a[0]
    a[0] = a[a.heapSize - 1]
    a.pop_back()
    siftDown(0)
    return min
```

O(1)

O(height)

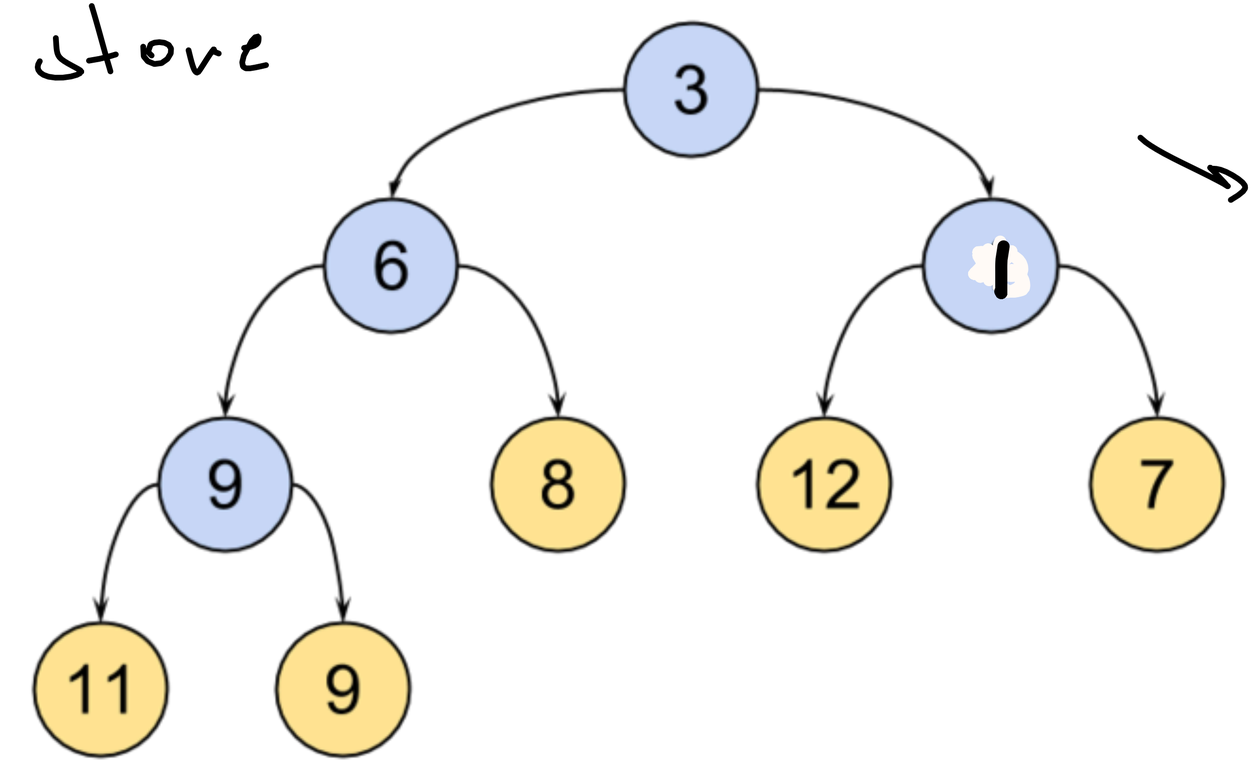
O(log n)

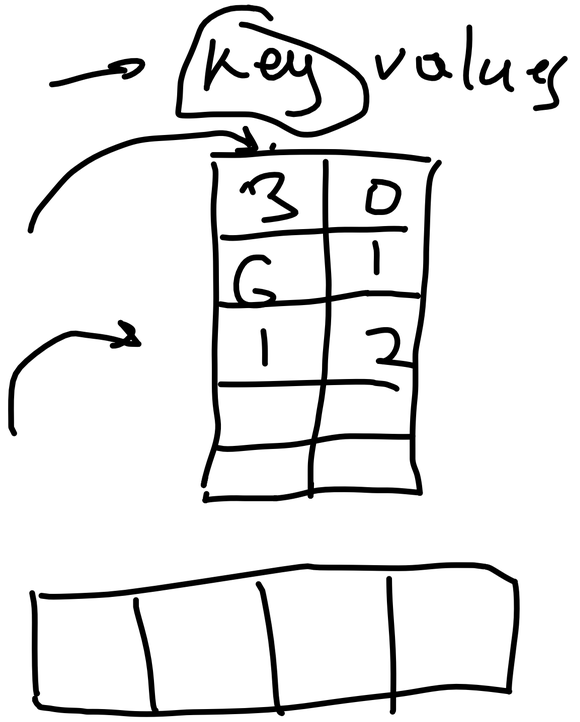
Time Complexity O(log n)

Decrease Key

Decrease Key: Decrease a value of element in O(log n)

Use map to store indices





Decrease Key

```
void DecreaseKey(element, new_element):
    elementPosition = GetIndexOfElement(element)
    a[elementPosition] = new_element
     siftUp(elementPosition)
```

In case all elements are different!

Recall Sort Naive Variant

Sorting Idea: Find largest element of array, place it in last position; then find the largest among the remaining elements, and place it next to the largest, etc

Trivial Finding Maximum: O(n) for single finding -> $O(n^2)$

We have a data structure for fast maximum finding!

Heap Sort

Sorting Strategy:

1. Build Max Heap from unordered array;

2. Find maximum element A[1];

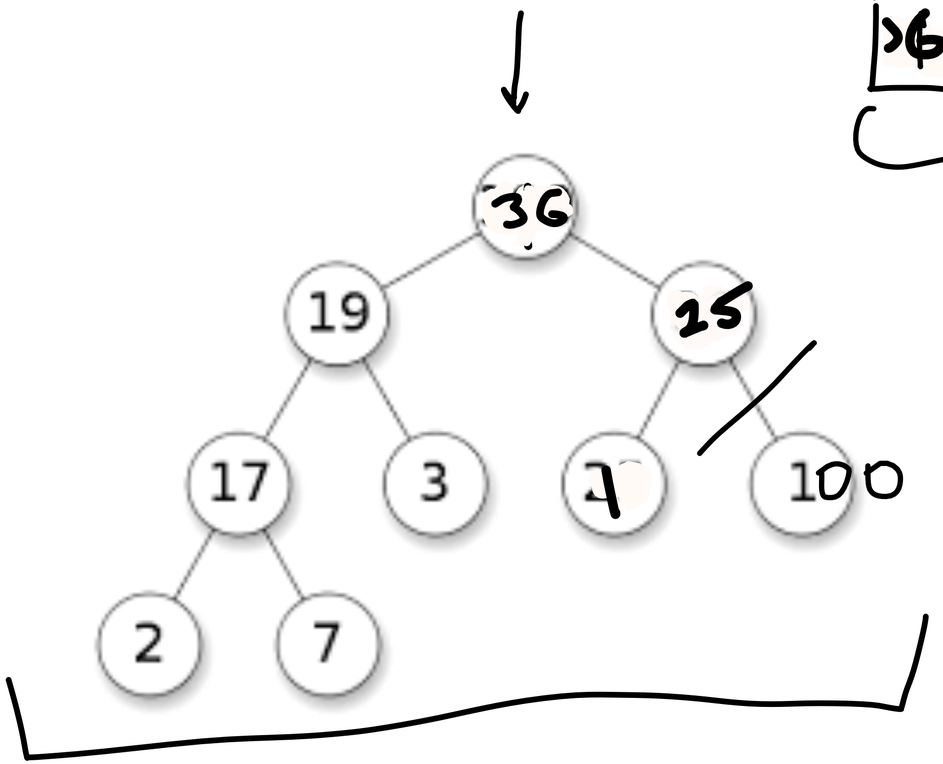
3. Swap elements A[n] and A[1]: now max element is at the end of the array!

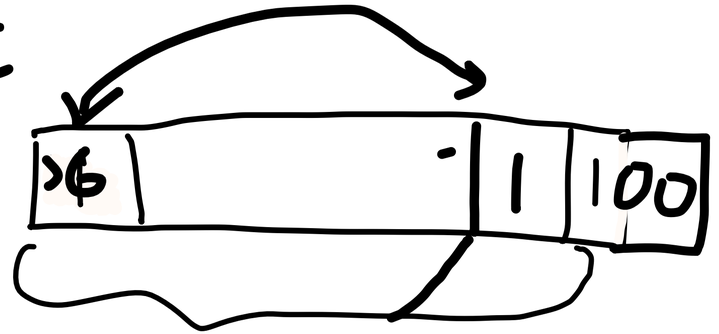
4. Discard node n from heap (by decrementing heap-size variable)

5. New root may violate max heap property, but its children are max heaps. Run sift down to fix this.

6. Go to step 2.

Heap Sort





h-2

Heap Sort

Sorting Strategy:

1. Build Max Heap from unordered array; $O(n)$

2. Find maximum element A[1]; $O(1)$

3. Swap elements A[n] and A[1]: now max element is at the end of the array! $O(1)$

4. Discard node n from heap (by decrementing heap-size variable) $O(1)$

5. New root may violate max heap property, but its children are max heaps. Run sift down to fix this. $O(\log n)$

6. Go to step 2.

Your questions!