# **HEAP**

* **Heapify** is a process of creating a heap
* **Heapify** is the process of converting a binary tree into a Heap data structure.

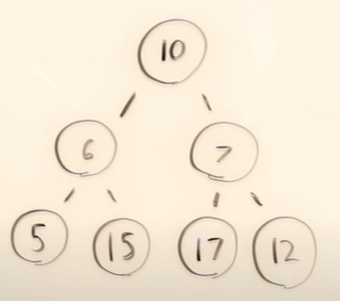
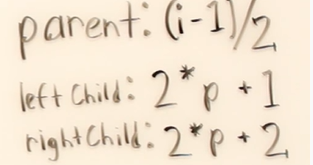
Heap is not considered an abstract data type. Heap is a specialized tree-based data structure

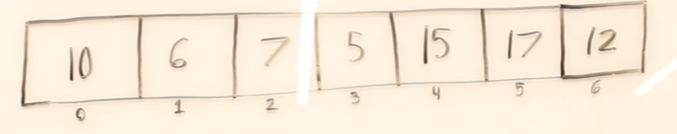
**Heap** is a binary tree that satisfies two conditions:

* Complete binary tree
* Every node has grater elements than all child (for max heap)

# **REPRESENTATION OF A BINARY TREE IN ARRAY**

**1-st variant. Start array from index [0]:**

** **

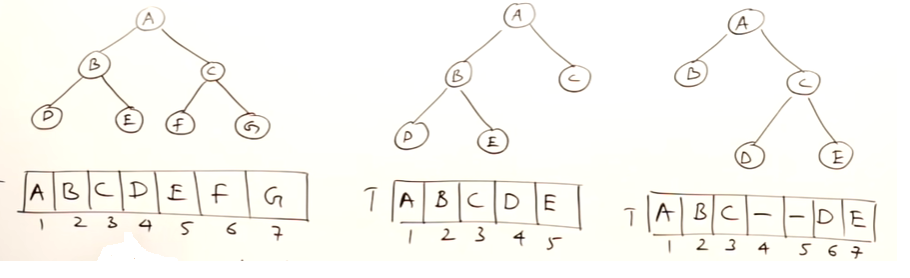


**Formula 1-st option when starting from index=0:**

If [i] is node, where [i] = 0,1,2,3,4 …

* parent is [(i-1)/2]
* Left child is [2\*i+1]
* right child is [2\*i+2]

**2-nd variant. Start array from index [1]:**



**Formula 2-nd option when starting from index=1:**

If [i] is node, where [i] = 1,2,3,4 …

* parent is [i/2]
* Left child is [2\*i]
* right child is [2\*i+1]

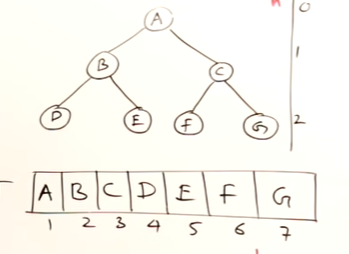
# **BINARY TREE – COMPLETE AND FULL**

**DEF1. Complete binary tree** is a binary tree where all levels filled out except possibly last and in the last level there is no gaps between nodes

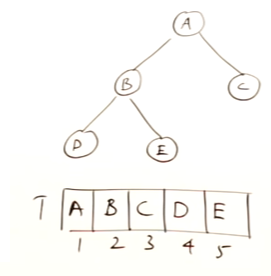
**DEF1. Complete binary tree** is a binary tree and if make an array representation all elements will be filled out

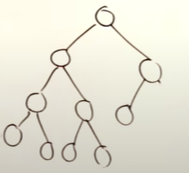
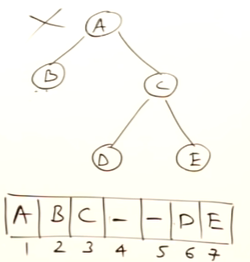
**Full binary tree** is a binary tree where all elements are filled out

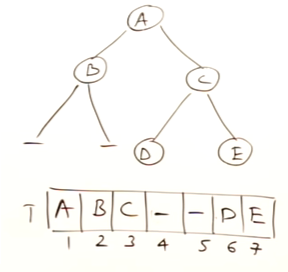
*FULL BINARY TREE*



*COMPLETE BINARY TREE*





# **HEAP**

**Heap** is a binary tree that satisfies two conditions

1. Complete binary tree
2. Every node has grater elements than all child (for max heap)

Facts:

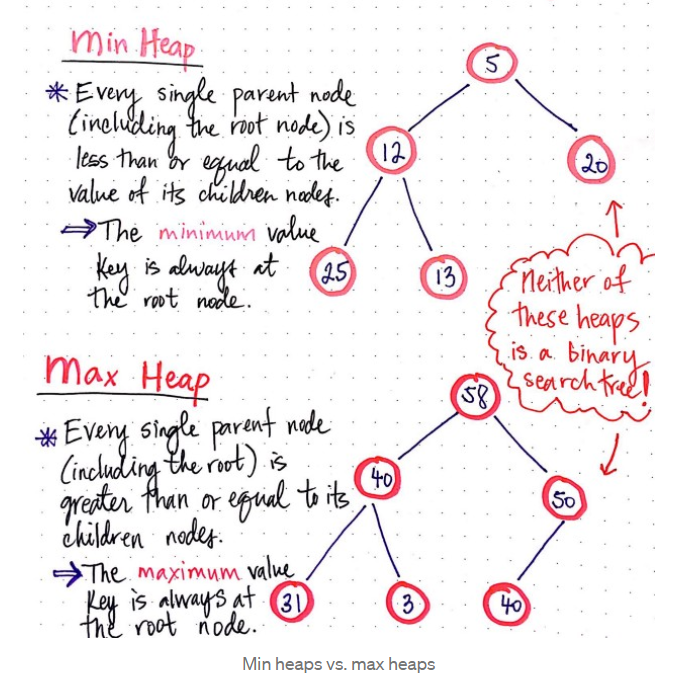
* Heap is not considered an abstract data type. Heap is a specialized tree-based data structure
* The most efficient implementation of heap is PriorityQueue
* Height of heap - O(log(n))
* Insertion into a heap - Ω(1), O(log(n))

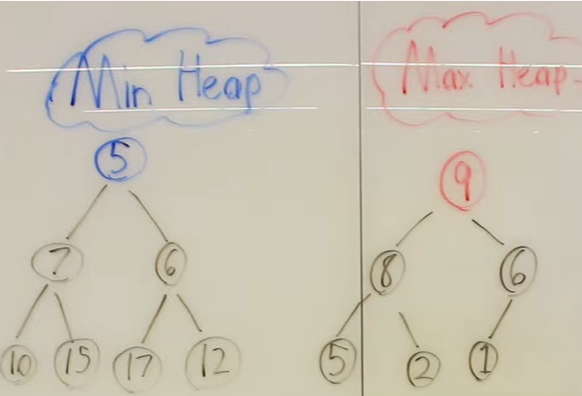
|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
| Insertion, deletion |  |  | O(log(n)) |
| Finding Min, Max (for Min and Max heaps) |  |  | O(1) |
| Height of heap |  |  | O(log(n)) |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

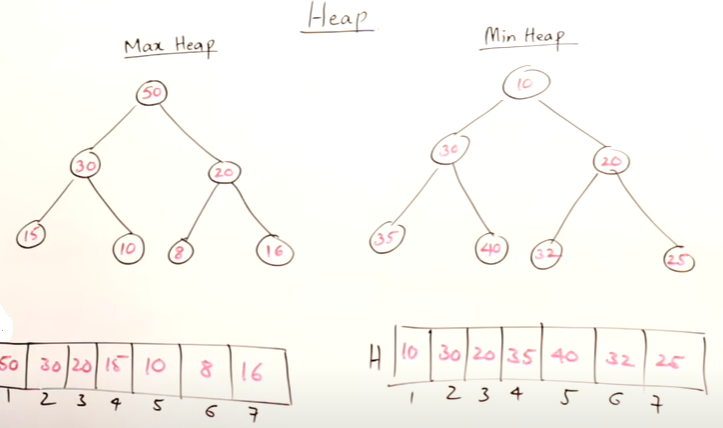
Type of heaps:

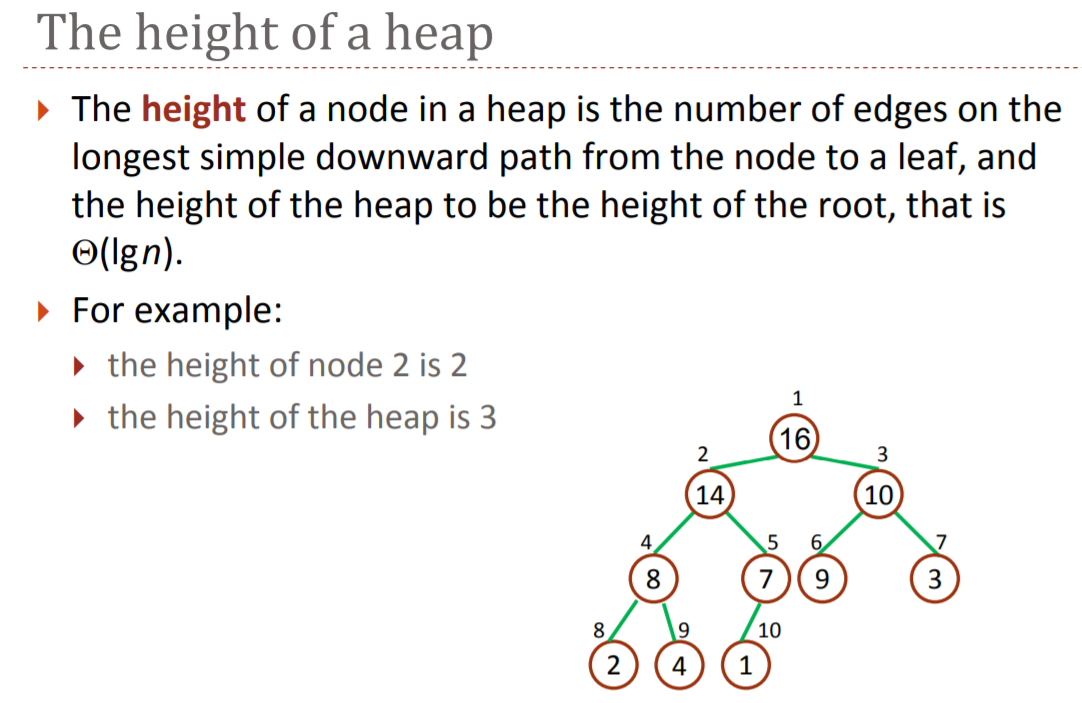
Min Heap – root has a minimum element among a tree

Max Heap – root has a maximum element among a tree









# **INSERTION IN A HEAP**

* Insertion into a heap - Ω(1), O(log(n))

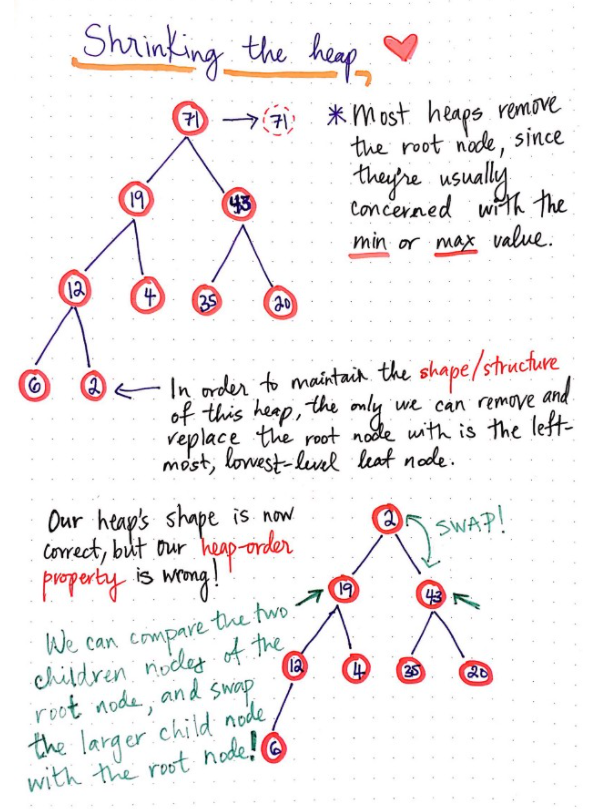
# **DELETION IN A HEAP**

You can delete only root node in a heap

* Deletion into a heap - Ω(1), O(log(n))
* Extract min and max - O(1)
* Insertion into a heap - O(log(n))

**Deletion**:

* Remove root – it goes to the end of array and pointer moves to one position left from right end. So, removed root is already in an ordered position and not considered in an array structure anymore
* Last node goes to the root place
* Heapify down process – checking heap structure



# **HEAP SORT**

Heap sort is based on a heap data structure.

The idea is:

* Create and populate a heap
* Eliminate element from a root: you swap elements from a root from last element and then heapify new elrment in a root. It repeats for all remaining elements in array(according to pointer)

**Fact**:

* It turns out that heap sort is a lot like selection sort in its logic: both algorithms find either the smallest or largest element, “select” it out, and put that item in its correct location in the sorted list.
* However, as similar as they are, heap sort is much better than selection sort in one massive way: its performance! Heap sort is basically a super-improved version of selection sort. Yes, it does find the largest element in an unsorted collection and orders it at the back of the list — however, it does all of this work so much faster than selection sort would!

Sorting:

* Max heap – is used for sorting array in ascending order (after removal elements)
* Min heap – is used for sorting array in descending order (after removal elements)

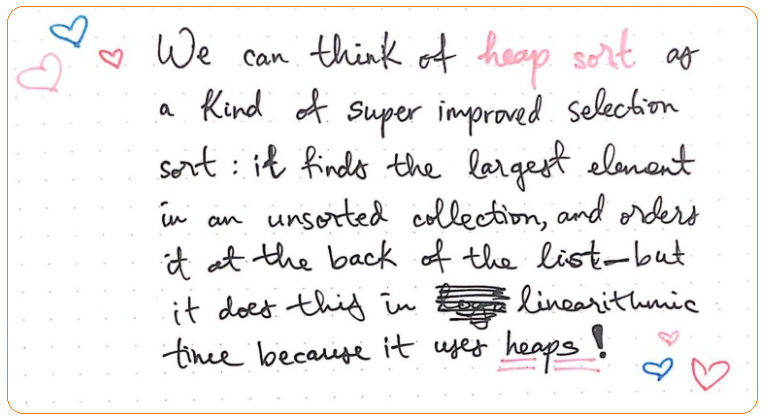
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **NAME** | **BEST** | **AVERAGE** | **WORST** | **SPACE COMPLEXITY** | **IN-PLACE** | **STABLE** | **COMPARISON** |
| **BUCKET SORT** | Ω(n+k) | Θ(n+k) | O(n2) | O(n) | No | N/A | No |

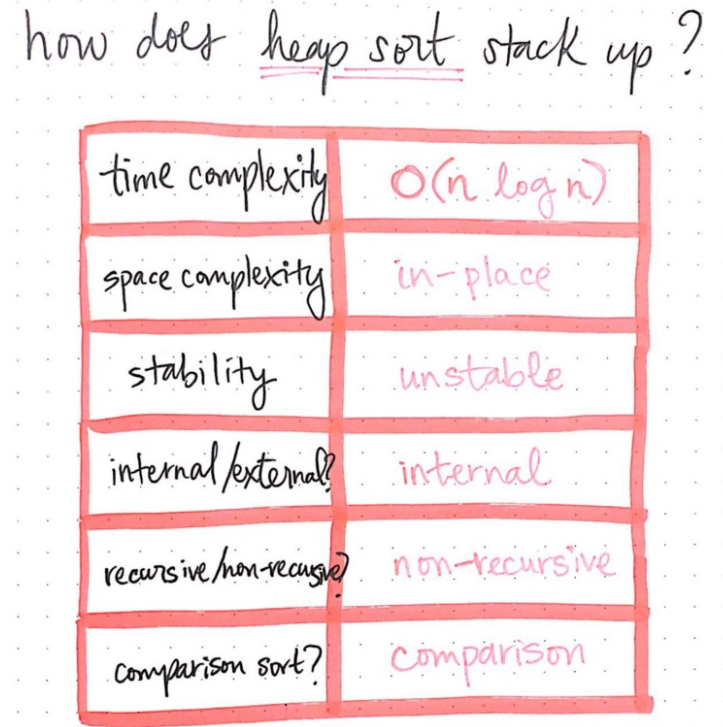
**Pros**:

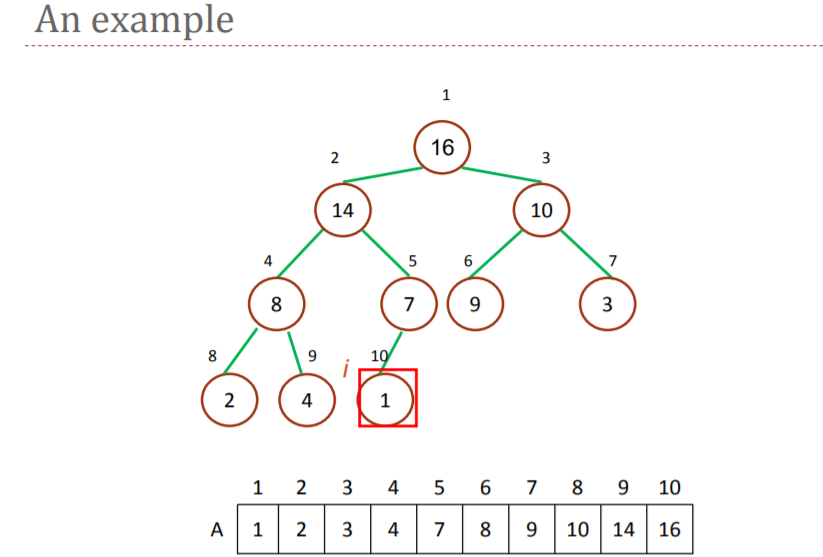
* Very useful if need to find top k elements
* An efficient way to build PriorityQueue (there are two options of PriorityQueue – max-priority queue and min-priority queue)
* Heap sort is particularly suitable for sorting a huge list of items.
* **Job Scheduling** - In Linux OS, heapsort is widely used for job scheduling of processes due to it's O(nlogn) time complexity and O(1) space complexity.
* **Graph Algorithms** - It can used in the implementation of priority queue for Djikstra's algorithm, Prim's algorithm and Huffmann encoding as well.

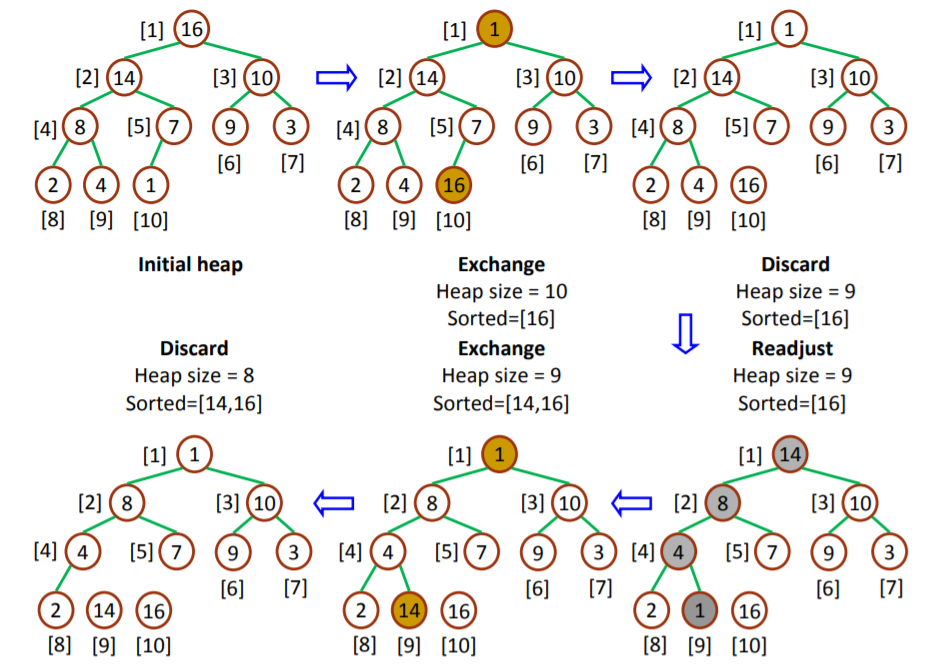
**Cons**:

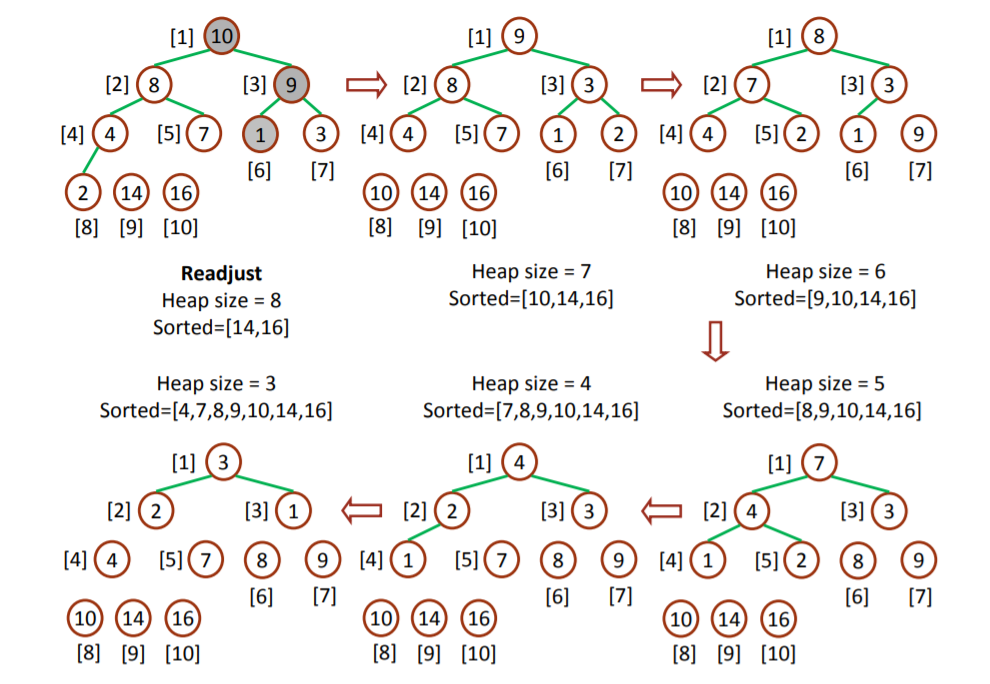
* Is not stable
* Although Heap Sort has O(n log n) time complexity even for the worst case, it doesn't have more applications ( compared to other sorting algorithms like Quick Sort, Merge Sort )











# **HEAPIFY**

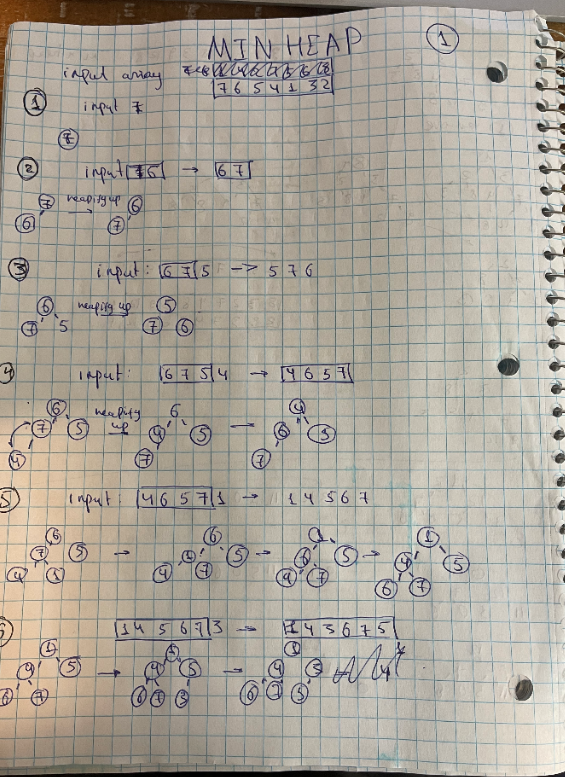
**Heapify** is a process of creating a heap

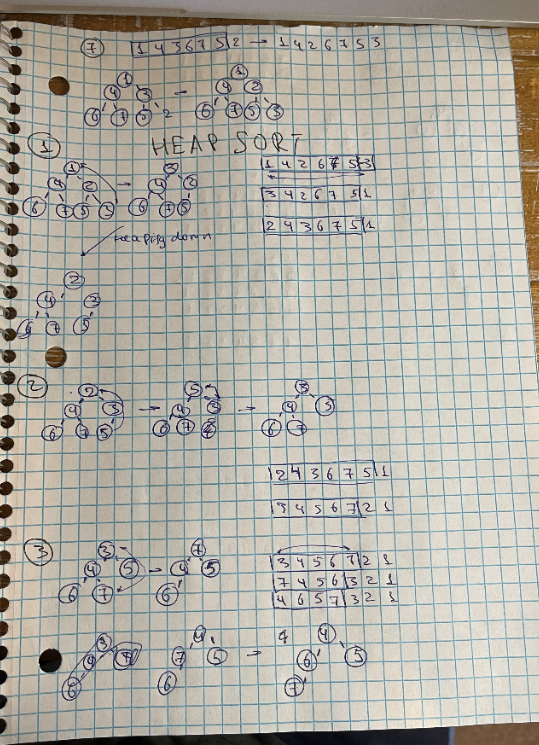
* Heapify - O(n)

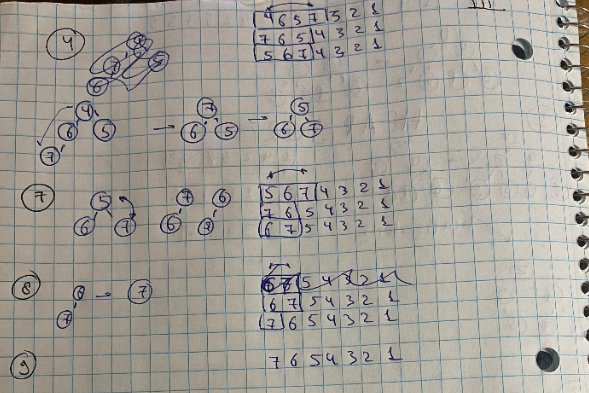
**Heapify** Rearrange a heap to maintain the heap property, that is, the key of the root node is more extreme (greater or less) than or equal to the keys of its children. If the root node's key is not more extreme, swap it with the most extreme child key, then recursively heapify that child's subtree. The child subtrees must be heaps to start.

bubble-up operation(it is also called as up-heap, percolate-up, sift-up, trickle-up, heapify-up, or cascade-up

.







# **HEAP IMPLEMENTATION – PRIORITY QUEUE**

Heaps are often implemented as arrays because they are super efficient ways of representing priority queues.

**Priority queue** is a special type of queue where every element has a priority and an item with a high priority is dequeued before an item with a lower priority

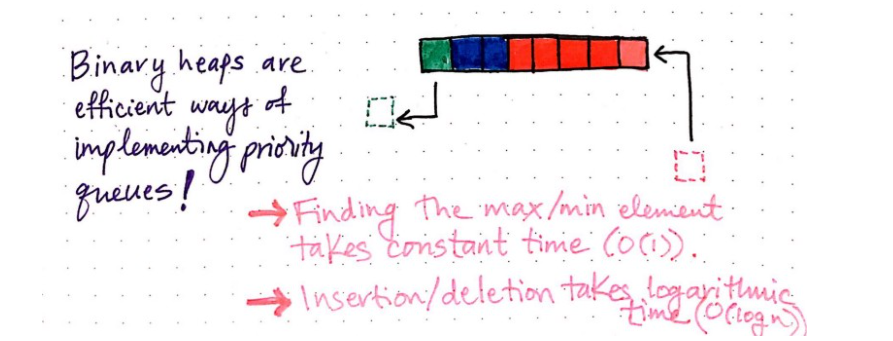
* A priority queue can be implemented using many of the data structures - an array, a linked list, or a binary search tree. However, those data structures do not provide the most efficient operations. To make all of the operations very efficient, we'll use a new data structure called a heap.
* Heap is generally preferred for priority queue implementation because heaps provide better performance compared arrays or linked list. In a Binary Heap, getHighestPriority() can be implemented in O(1) time, insert() can be implemented in O(Logn) time and deleteHighestPriority() can also be implemented in O(Logn) time.

Time complexity:

Insertion, deletion - O(log(n))

Peek(), poll()

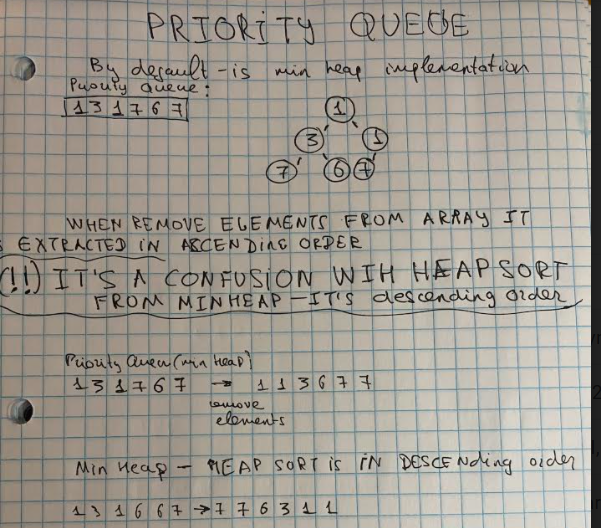
**Applications of Priority Queue**:

* CPU Scheduling
* Graph algorithms like Dijkstra’s shortest path algorithm, Prim’s Minimum Spanning Tree, etc
* All queue applications where priority is involve
* 

**Example of queue**:

priority queue might be used, for example, to handle the jobs sent to the Computer Science Department's printer: Jobs sent by the department chair should be printed first, then jobs sent by professors, then those sent by graduate students, and finally those sent by undergraduates. The values put into the priority queue would be the priority of the sender.

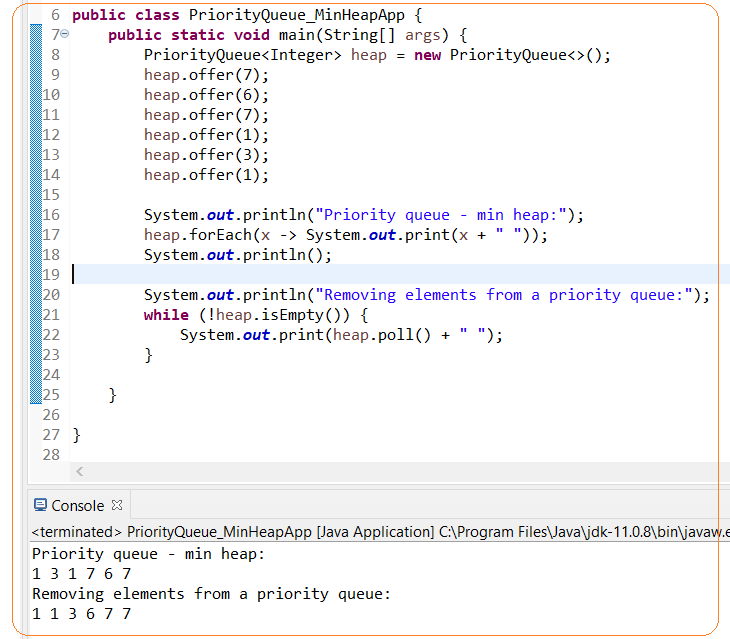
**(!) It might be confusion - removing elements from min heap is in a ascending order, but when we make heapsort based on min heap it goes in descending order**.



# **#1.PRIORITY QUEUE – MIN HEAP**

**Priority Queue is a min heap by default**. But when you remove elements it comes in an ascending order

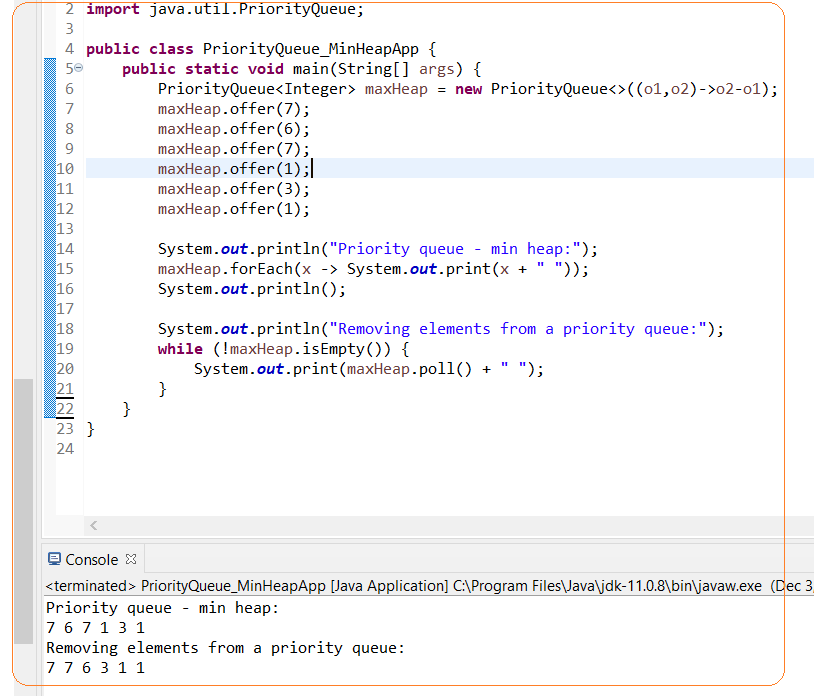
Note: PAY ATTENTION! when you make a heap sort from min heap you get a descending order.



# **#2.PRIORITY QUEUE – MAX HEAP (IMPLICIT COMPARATOR)**

If you want to build a priority queue from a max heap you need implement a comparator. When you remove elements from a max heap it comes in a descending order

Note: PAY ATTENTION! when you make a heap sort from max heap you get an ascending order.



# **#3.PRIORITY QUEUE – MAX HEAP (EXPLICIT COMPARATOR)**

