Priority tail using heaps

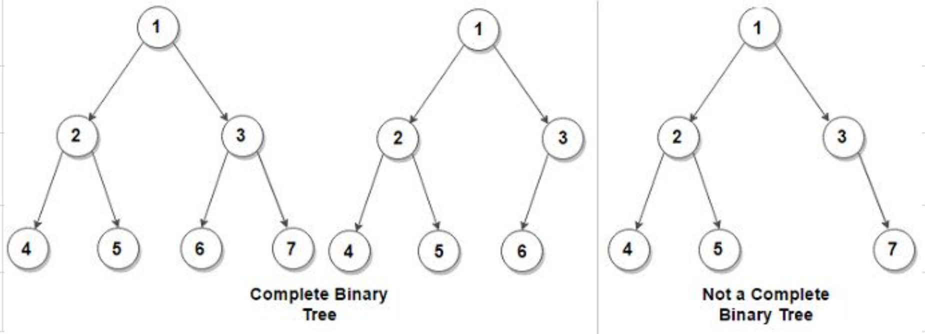
Author: Sanchez Sauñe, Cristhian Wiki

Heaps

Heaps are widely used tree-like data structures in which the parent nodes satisfies any one of the criteria given below.

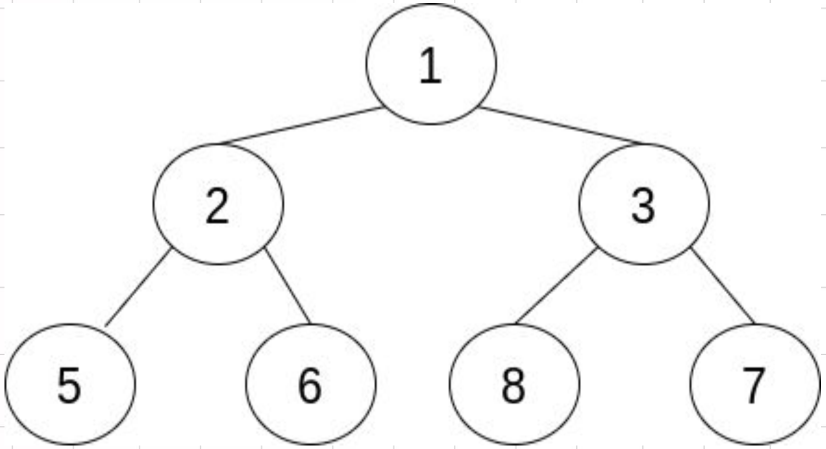
- The value of the parent node in each level is less than or equal to its children's values – **min-heap**.
- The value of the parent node in each level higher than or equal to its children's values – max-heap.

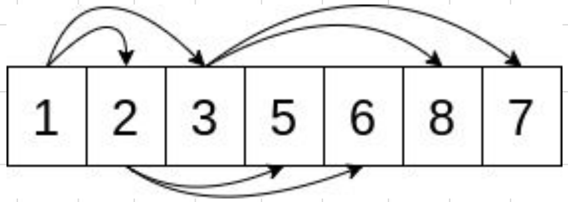
The heaps are complete binary trees and are used in the implementation of the priority queues. A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.



A Min-Heap

A Binary Heap is a Complete Binary Tree. A binary heap is typically represented as an array.

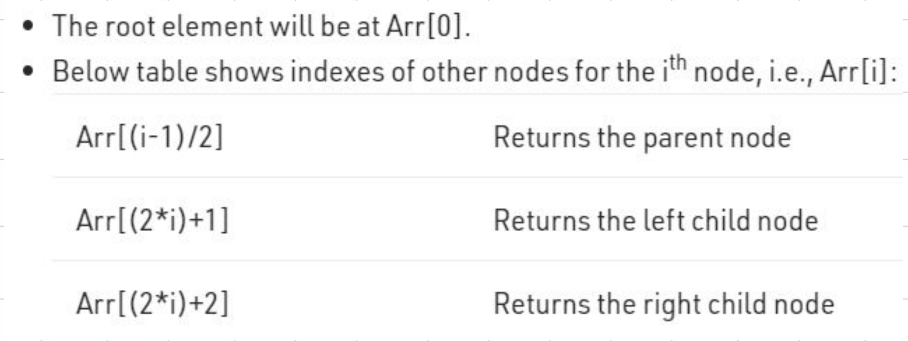




Binary Heap

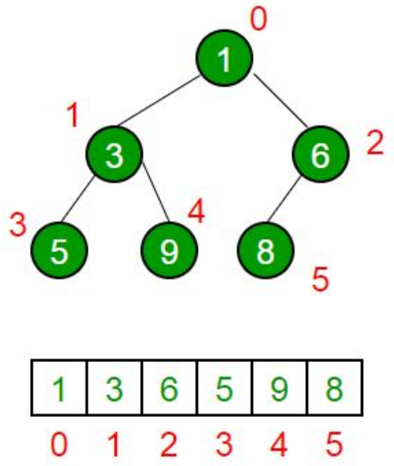
How is Binary Heap represented?

A Binary Heap is a Complete Binary Tree. A binary heap is typically represented as an array.



- The root element will be at Arr[0].
- Below table shows indexes of other nodes for the ith node, i.e., Arr[i]:

Arr[(i-1)/2]	Returns the parent node
Arr[(2*i)+1]	Returns the left child node
Arr[(2*i)+2]	Returns the right child node



The traversal method use to achieve Array representation is Level Order

Binary heap

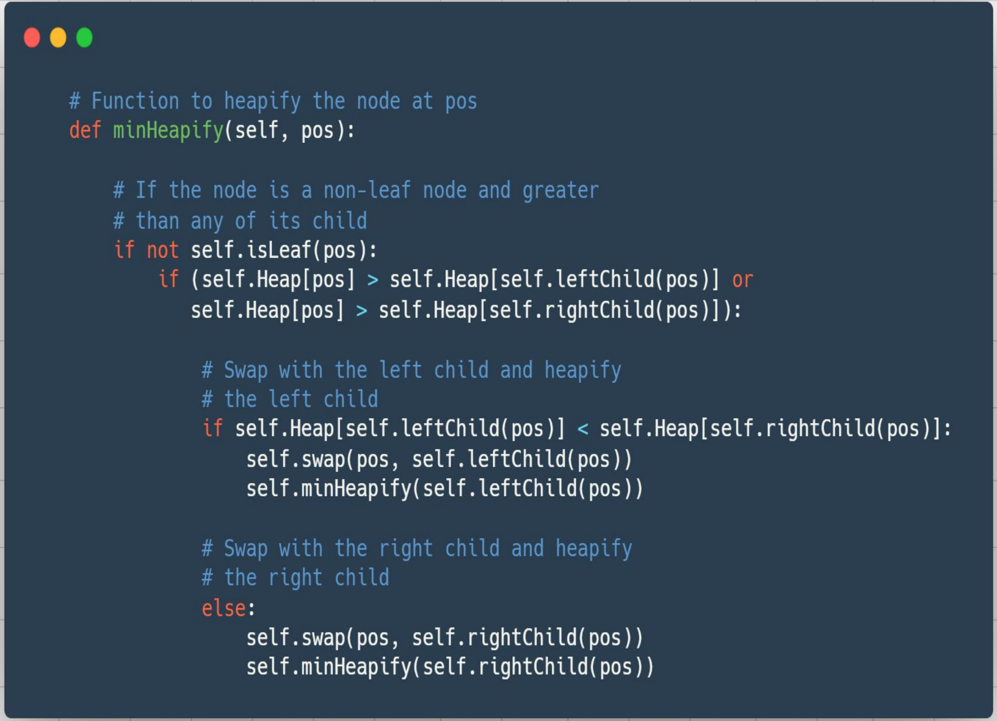
Heapify

- The parent node in index **'i'** is less (or max) than or equal to its children.
- The left child of a node in index **'i'** is in index '(2*i) + 1'.
- The right child of a node in index **'i'** is in index '(2*i) + 2'.

The time complexity of heapify depends on where it is within the heap. It takes **O(1)** time when the node is a leaf node (which makes up at least half of the nodes) and **O(logn)** time when it's at the root.

Heapify

Binary heap


```
# Function to heapify the node at pos
def minHeapify(self, pos):

    # If the node is a non-leaf node and greater
    # than any of its child
    if not self.isLeaf(pos):
        if (self.Heap[pos] > self.Heap[self.leftChild(pos)] or
           self.Heap[pos] > self.Heap[self.rightChild(pos)]):

            # Swap with the left child and heapify
            # the left child
            if self.Heap[self.leftChild(pos)] < self.Heap[self.rightChild(pos)]:
                self.swap(pos, self.leftChild(pos))
                self.minHeapify(self.leftChild(pos))

            # Swap with the right child and heapify
            # the right child
            else:
                self.swap(pos, self.rightChild(pos))
                self.minHeapify(self.rightChild(pos))
```

Implementations

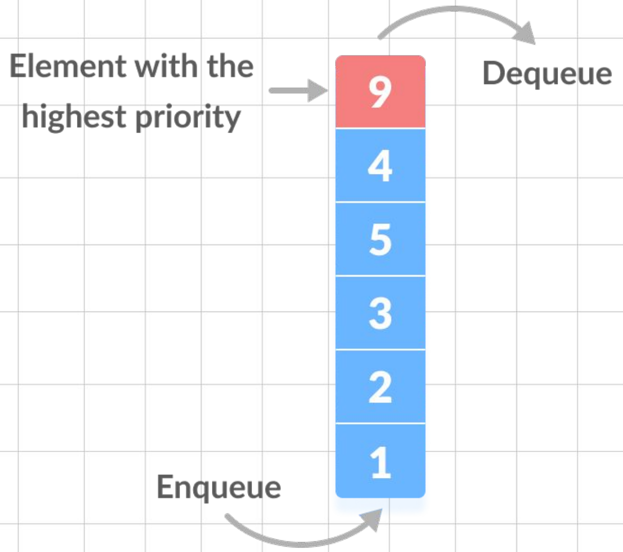
The heaps are complete binary trees and are used in the implementation of the priority queues. The min-heaps play a vital role in scheduling jobs, scheduling emails or in assigning the resources to tasks based on the priority (such as a self-driving vehicle) .

LEFT REARWARD VEHICLE CAMERA
MEDIUM RANGE VEHICLE CAMERA
RIGHT REARWARD VEHICLE CAMERA
MOTION FLOW
LANE LINES
LANE LINES
ROAD FLOW
IN-PATH OBJECTS
ROAD LIGHTS
OBJECTS
ROAD SIGNS

Priority queues

These are abstract data types and are a special form of queues. The elements in the queue have priorities assigned to them. Based on the priorities, the first element in the priority queue will be the one with the highest priority. The basic operations associated with these priority queues are listed below:

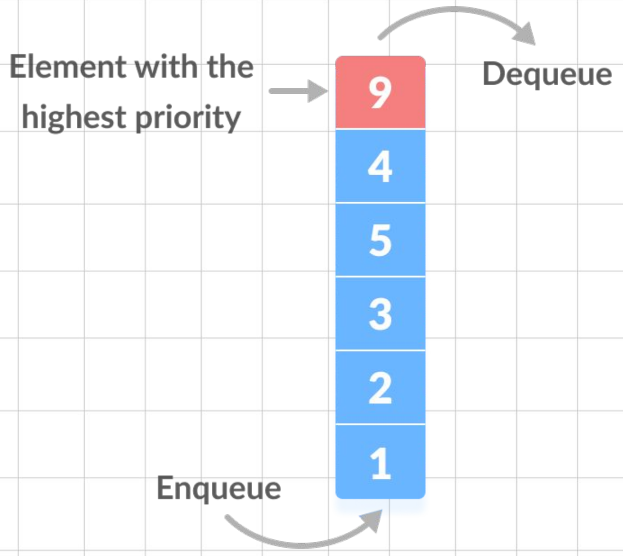
- ***insert :*** To insert an element along with its priority. The element will be placed in the order of its priority only.
- ***Pop (remove):*** To pop the element with the highest priority. The first element will be the element with the highest priority.



Priority queues

The priority queues can be used for all scheduling kind of processes. The programmer can decide whether the largest number is considered as the highest priority or the lowest number will be considered as the highest priority.

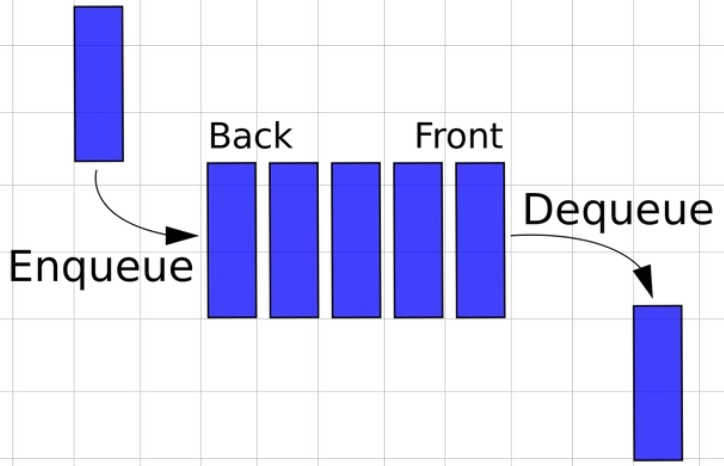
If two elements have the same priority, then they appear in the order in which they appear in the queue.



Removing Highest Priority Element

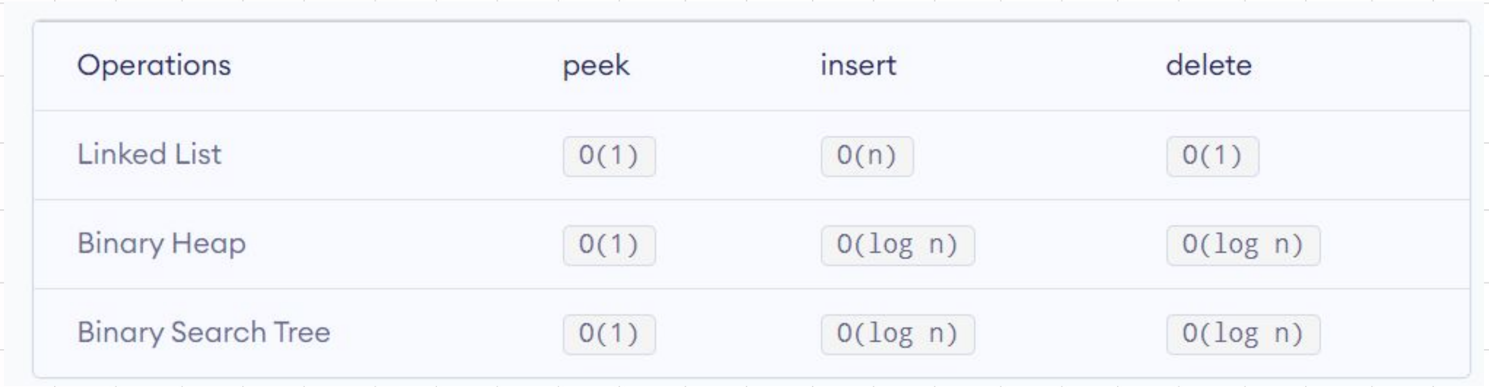
Difference between Priority Queue and Normal Queue

In a queue, the first-in-first-out rule is implemented whereas, in a priority queue, the values are removed on the basis of priority. The element with the highest priority is removed first.



Implementation of Priority Queue

Priority queue can be implemented using an array, a linked list, a heap data structure, or a binary search tree. Among these data structures, heap data structure provides an efficient implementation of priority queues. A comparative analysis of different implementations of priority queue is given below.



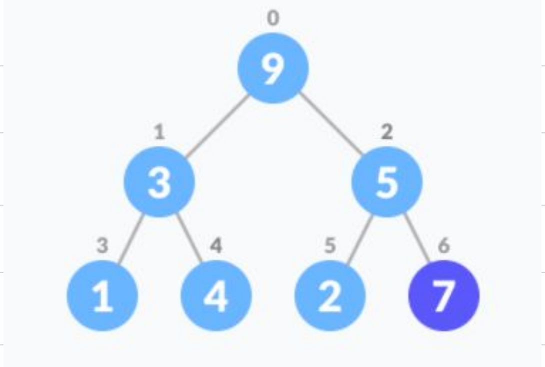
Operations	peek	insert	delete
Linked List	`O(1)`	`O(n)`	`O(1)`
Binary Heap	`O(1)`	`O(log n)`	`O(log n)`
Binary Search Tree	`O(1)`	`O(log n)`	`O(log n)`

Priority Queue Operations

1. Inserting an Element into the Priority Queue

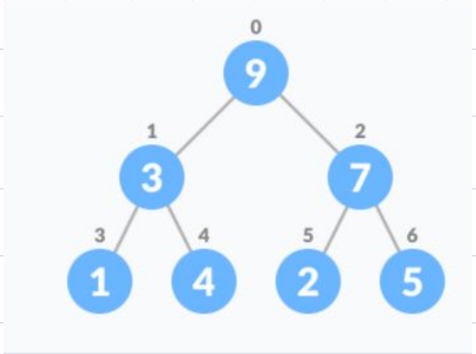
Inserting an element into a priority queue (max-heap) is done by the following steps

- Insert the new element at the end of the tree.



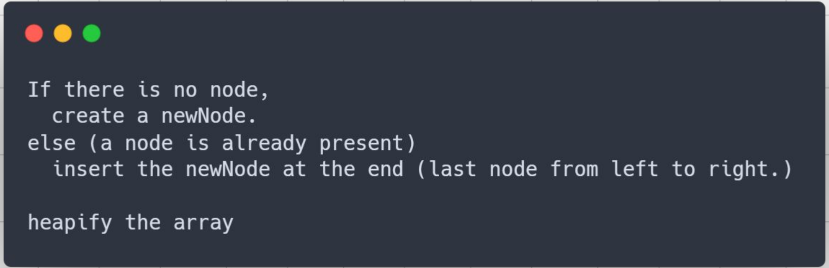
Insert an element at the end of the queue

- Heapify the tree



Heapify after insertion

Algorithm for insertion of an element into priority queue (max-heap).



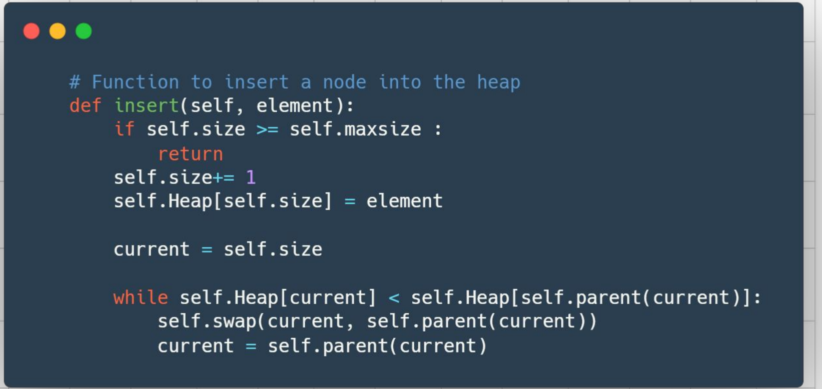
```
If there is no node,
  create a newNode.
else (a node is already present)
  insert the newNode at the end (last node from left to right.)

heapify the array
```

Complexity

If a node is to be inserted at a level of height H:

I. Complexity of adding a node is: **O(1)**
II. Complexity of swapping the nodes(up heapify): **O(H)** (swapping will be done H times in the worst case scenario)
III. Total complexity: **O(1) + O(H) = O(H)**
IV. For a Complete Binary tree, its height **H = O(log N)**, where **N** represents total n° of nodes.
V. Therefore, Overall Complexity of insert operation is **O(log N)**.

Code

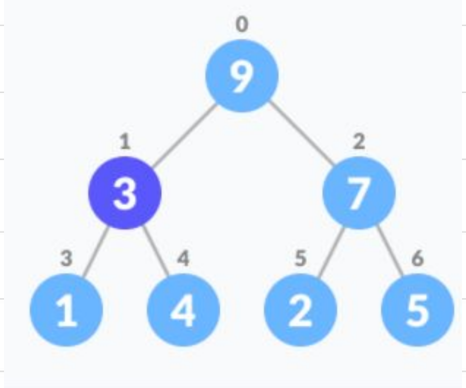


```python
# Function to insert a node into the heap
def insert(self, element):
    if self.size >= self.maxsize :
        return
    self.size+= 1
    self.Heap[self.size] = element

    current = self.size

    while self.Heap[current] < self.Heap[self.parent(current)]:
        self.swap(current, self.parent(current))
        current = self.parent(current)
```

2. Deleting an Element from the Priority Queue

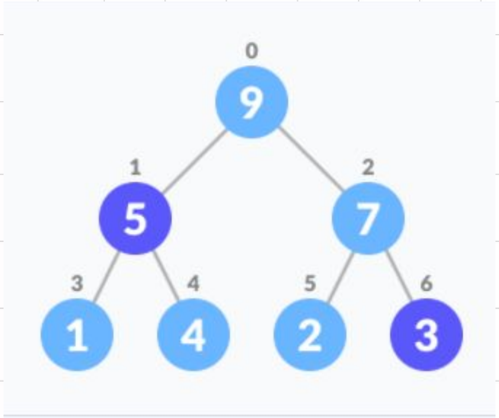
Deleting an element from a priority queue (max-heap) is done as follows:

- Select the element to be deleted.



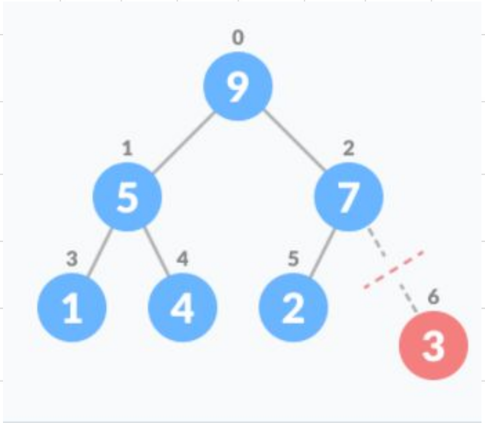
Select the element to be deleted

- Swap it with the last element.



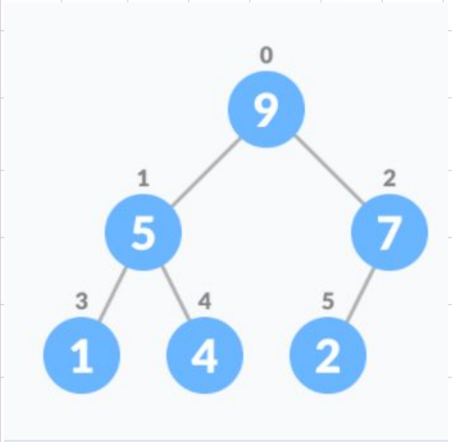
Swap with the last leaf node element

- Remove the last element.



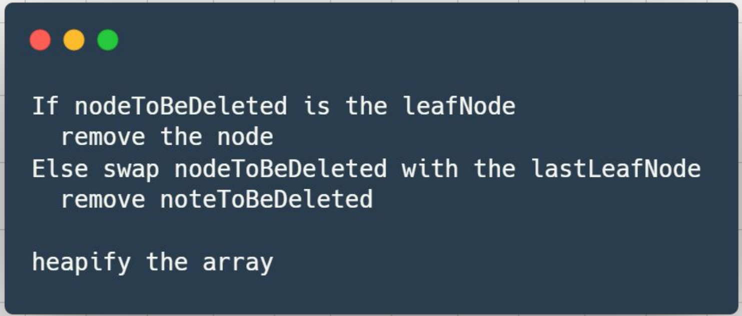
Remove the last element leaf

- Heapify the tree



Heapify the priority queue

Algorithm for deletion of an element in the priority queue (max-heap).



```
If nodeToBeDeleted is the leafNode
  remove the node
Else swap nodeToBeDeleted with the lastLeafNode
  remove noteToBeDeleted

heapify the array
```

Complexity

If a node is to be deleted from a heap with height H:

I. Complexity of swapping parent node and leaf node is: **O(1)**
II. Complexity of swapping the nodes(down heapify): **O(H)** (swapping will be done H times in the worst case scenario)
III. Total complexity: **O(1) + O(H) = O(H)**
IV. For a Complete Binary tree, its height **H = O(log N)**, where **N** represents total n˚ of nodes.
V. Therefore, Overall Complexity of delete operation is **O(log N).**

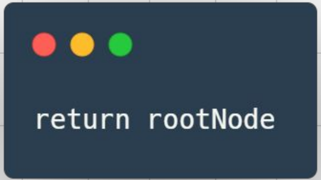
3. Peeking from the Priority Queue (Find max/min)

Peek operation returns the maximum element from Max Heap or minimum element from Min Heap without deleting the node.

Complexity

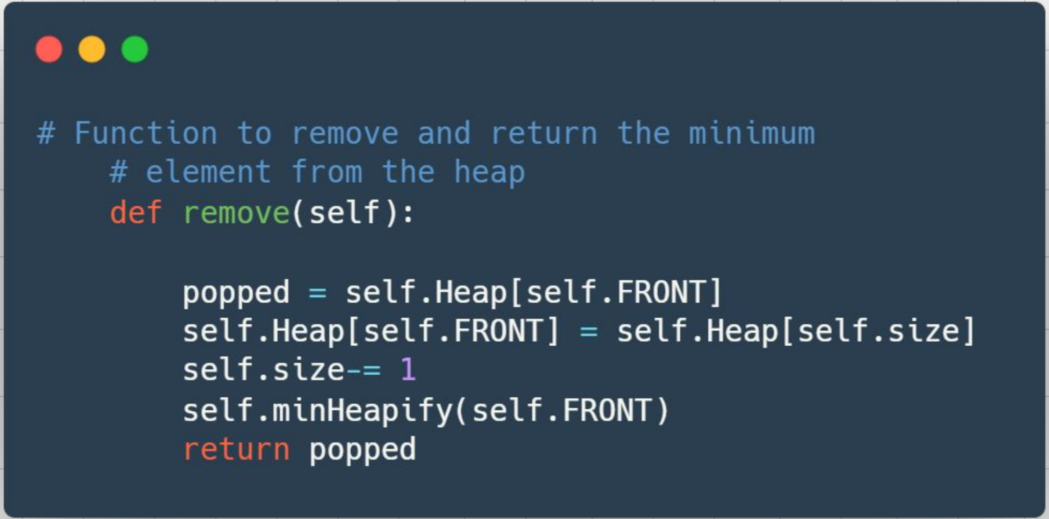
In order to obtain the maximum value just return the value of the root node (which is the biggest element in Max Heap), So simply return the element at index 0 of the array.

Hence, Complexity of getting maximum (or min) value is: O(1)



```
return rootNode
```

3. Peeking from the Priority Queue (Find max/min)



```
# Function to remove and return the minimum
    # element from the heap
    def remove(self):

        popped = self.Heap[self.FRONT]
        self.Heap[self.FRONT] = self.Heap[self.size]
        self.size-= 1
        self.minHeapify(self.FRONT)
        return popped
```