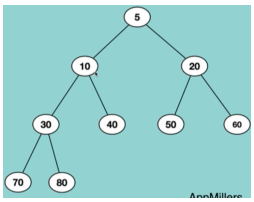
Binary heap:

A Binary Heap is a Binary Tree with following properties:

- A Binary Heap is either Min Heap or Max Heap. In a Min Binary Heap, the key at root must be minimum among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary Tree.
- It's a complete tree (All levels are completely filled except possibly the last level and the last level has all keys as left as possible). This property of Binary Heap makes them suitable to be stored in an array.

This is how a minimum binary heap looks:



Why we need Binary Heap?

Let's say we want to solve this problem:

Find the minimum or maximum number among a set of numbers in logN time. And also we want to make sure that inserting additional numbers does not take more than O(logN) time

Possible solutions are:

- Store the numbers in sorted array => inserting an element at the beginning takes O(n) time complexity
- Linked List: inserting an element in the proper location may take in the worst-case O(n) time complexity
- Binary Heap: it takes O (log n) for insertion and finding the Min or Max takes O (1) time complexity

the Real problems of Binary Heap: (the practical usage of Binary heap)

- Prim's Algorithm
- Heap sort
- Priority Queue

Insert a node to Binary Heap:
We insert a node in to the first available node in the list: so the index is : heapSize +1 in the following example:

```
class Heap:
    def __init__(self,size):
        self.customList = (size+1) * [None]
        self.heapSize = 0
        self.maxSize = size+1
```

whenever we insert a node in the list we increment the heapSize by 1;
and each time we need to run a function to check if it is right by the rules of Heap algorithm (depend on what kind of heap we are using (max or Min))
this is how we adjust the list to keep it a Binary Heap:

```
def heapifyTreeInsert(rootNode,index,heapType): # O(log n ) time and space complexity
    parentIndex = int(index/2)
```

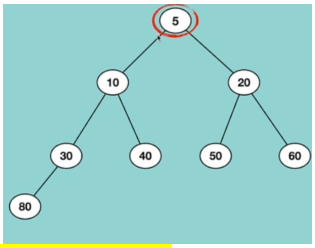
```
    if index <=1:
        return
    if heapType == "Min":
        if rootNode.customList[index] < rootNode.customList[parentIndex]:
            temp = rootNode.customList[index]
            rootNode.customList[index] = rootNode.customList[parentIndex]
            rootNode.customList[parentIndex] = temp
            heapifyTreeInsert(rootNode,parentIndex,heapType) # -----> O(log n) time
complexity
    elif heapType == "Max":
        if rootNode.customList[index] > rootNode.customList[parentIndex]:
            temp = rootNode.customList[index]
            rootNode.customList[index] = rootNode.customList[parentIndex]
            rootNode.customList[parentIndex] = temp
            heapifyTreeInsert(rootNode,parentIndex,heapType) # -----> O(log n) time
complexity
```

and the actual insert function:

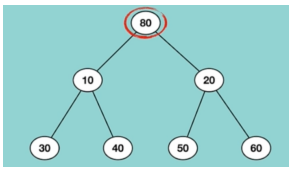
```
def insertNode(rootNode, nodevalue,heapType): # O(log n) time complexity and space
complexity
    if rootNode.heapSize +1 == rootNode.maxSize:
        return "the Binary Heap is full"
    rootNode.customList[rootNode.heapSize + 1] = nodevalue
    rootNode.heapSize +=1
    heapifyTreeInsert(rootNode,rootNode.heapSize,heapType)
```

Extract an element from Binary Heap:

The only node that can be extracted from Binary heap is always rootNode
Let's say we have a Minimum heap like this:

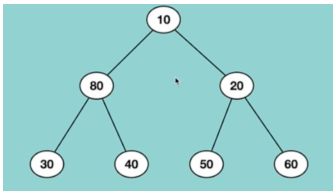


And the rootNode is 5 (it's a Minimum Heap) in this case we will find the last element in the Binary Heap (the last element is the element which is located in the last level of Binary heap and the most right location of them (so in this case we have only one element in the last level and that is 80)) and we replace the last element with the root node:



Now our Binary heap needs to be Heapified because it's a Minimum Binary Heap and the root node is greater that both left and right Children

We swap the root Node with the smallest child:



We do this until the tree becomes Binary Heap.