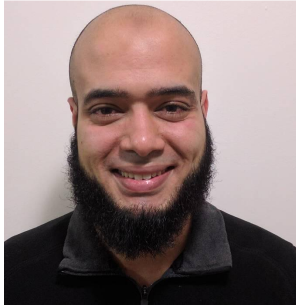
Data Structures

Heap Homework 2

Mostafa S. Ibrahim
Teaching, Training and Coaching since more than a decade!

Artificial Intelligence & Computer Vision Researcher
PhD from Simon Fraser University - Canada
Bachelor / Msc from Cairo University - Egypt
Ex-(Software Engineer / ICPC World Finalist)



Problem #: Kth Largest number (stream)

```python
import heapq


class KthLargestProcessor:
    def __init__(self, k):...

    def next(self, number):...
```

- This class receives a **infinite stream** of numbers, each time returning the kth **largest** number
- In other words, if numbers are sorted decreasingly, it is arr[k-1]
- Use the built-in heap basic functionalities
    - Don’t use nlargest/nsmallest to get the answer

```python
if __name__ == "__main__":
    lst = [2, 17, 22, 10, 8, 37, 14,
           19, 7, 6, 5, 12, 25, 30]

    k = 4
    processor = KthLargestProcessor(k)

    for idx, val in enumerate(lst):
        ans = processor.next(val)
        print(idx, ans)

        right_answer = heapq.nlargest(k, lst[:idx+1])
        assert ans == right_answer[-1]
```

```
0 2
1 2
2 2
3 2
4 8
5 10
6 14
7 17
8 17
9 17
10 17
11 17
12 19
13 22
```

Problem #2: Priority Queue

- Priority queue is a queue in which each element has a "**priority**" associated with it. Elements with **high priority** are **served first** before low priority.
- Assume, in an OS, we have tasks each with priority [a positive value]
  - Assume we enqueued elements as follows:
  - Enqueue (task_id = 1131, priority = 1)
  - Enqueue (task_id = 3111, priority = 3)
  - Enqueue (task_id = 2211, priority = 2)
  - Enqueue (task_id = 3161, priority = 3)
  - Let’s print tasks in order: 3111 3161 2211 1131
- Implement a priority queue based on max-heap code
  - By definition, the top element in any MaxHeap will be the element of maximum value
  - MaxHeap, therefore, possesses the perfect **underlying implementation** for a priority queue **ADT**

Problem #2: Priority Queue

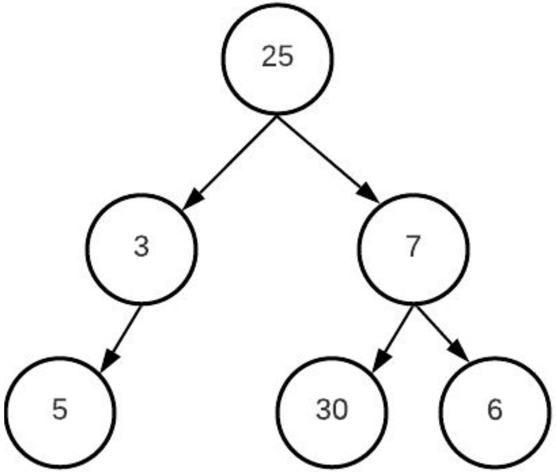
```python
tasks = PriorityQueue()

tasks.enqueue(1131, 1)
tasks.enqueue(3111, 3)
tasks.enqueue(2211, 2)
tasks.enqueue(3161, 3)
tasks.enqueue(7761, 7)

print(tasks.dequeue())
print(tasks.dequeue())

tasks.enqueue(1535, 1)
tasks.enqueue(2815, 2)
tasks.enqueue(3845, 3)
tasks.enqueue(3145, 3)

while not tasks.empty():
    print(tasks.dequeue(), end = ' ')
```

```
7761
3111
3145 3161 3845 2815 2211 1131 1535
```

- Observe
  - Value 3161 is added before 3145
  - Value 3161 is printed after 3145
  - This is valid: the constraint is tasks with higher priority are printed first
  - If the same priority = print in any order
- Study this example to know WHY max-heap can’t **preserve input order**

Problem #3: Binary Tree Special Traversal

- Going back to the **Binary Tree** section: add function
- **def** level_order_traversal_sorted(self):
- It does level order traversal, but at each level values are printed from small to large.
    - We can do this trivially by adding in a list and sort it
- For educational purposes, Use **heapq** to do the task
    - For simplicity, assume all values are **UNIQUE**
- Output for this tree
    - 25
    - 3 7
    - 5 6 30



```python
tree = BinaryTree(1)
tree.add([20, 50, 8], ['L', 'L', 'L'])
tree.add([20, 50, 17], ['L', 'L', 'R'])
tree.add([20, 40, 10], ['L', 'R', 'L'])
tree.add([20, 40, 11], ['L', 'R', 'R'])

tree.add([90, 60, 88], ['R', 'L', 'L'])
tree.add([90, 60, 13], ['R', 'L', 'R'])
tree.add([90, 7, 95], ['R', 'R', 'L'])
tree.add([90, 7, 15], ['R', 'R', 'R'])

#tree.level_order_traversal_normal()
tree.level_order_traversal_sorted()
```

```
Normal:
Level 0: 1
Level 1: 20 90
Level 2: 50 40 60 7
Level 3: 8 17 10 11 88 13 95 15

Sorted per level
Level 0: 1
Level 1: 20 90
Level 2: 7 40 50 60
Level 3: 8 10 11 13 15 17 88 95
```

Optional Mini-Project - No solution/support from me

- Design a data structure that provides **find_min and find_max** in O(1)
  - Other operations: insert(value), delete_min, delete_max
- One approach: let's **reuse** the common data-structures
  - You will use several ones in the same time
  - E.g. min-heap + max-heap + doubly linked list
  - Interesting: min and max heaps store addresses of nodes
- Another approach: let's invent a new data-structure
  - Min-max heap is an interesting DS
  - Even level acts like a min heap (e.g. node smaller than descendants)
  - Odd level acts like a max heap
  - The code merges the 2 heaps together somewhat
  - Interesting read

“Acquire knowledge and impart it to the people.”

“Seek knowledge from the Cradle to the Grave.”