Priority Queues in Python

Introduction to Priority Queues in Python

A priority queue is a data structure that stores a collection of elements, each with an associated priority.In a priority queue, elements are organised based on their priorities in such a way that elements with higher priorities are processed before elements with lower priorities. The exact ordering depends on whether the priority queue is implemented as a min-heap or max-heap.

Difference between PriorityQueue and Queue

PriorityQueue	Queue
Priority Queue is an extension of Queue with priority factor embedded.	Queue is a linear data structure.
Serves the element with higher priority first.	Follows First In First Out (FIFO) algorithm to serve the elements.
Enqueue and dequeue done in O(log n) using binary heaps.	Enqueue and dequeue done in O(1).
Used in algorithms such as Dijkstra's Algorithm, Prim's Algorithms, CPU Scheduling.	Used in algorithms such as Breadth First Search.

Difference between PriorityQueue and Heap

- The priority queue is working on the queue and the heap is working on the tree data structure.
- The priority queue is stored array value in a simple form while the heap is stored array value in a sorted form.
- The heap provides multiple functions and operations than the priority queue. The priority queue provides queue-related functions.
- The heap implements abstract data types such as priority queue but priority queue does not implement heap.
- The priority queue is simpler than the heap data structure. The heap is complicated because of the parent node rule.

Build priority using “heapq”.

```
import heapq
class PriorityQueue:
    def __init__(self):
        self.queue = []
        self.index = 0  # To handle elements with the same priority
    def push(self, priority, item):
        heapq.heappush(self.queue, (priority, self.index, item))
        self.index += 1
    def pop(self):
        return heapq.heappop(self.queue)[-1]
    def is_empty(self):
        return len(self.queue) == 0
    def __len__(self):
        return len(self.queue)
pq = PriorityQueue()
pq.push(3, 'task1')
pq.push(1, 'task2')
pq.push(2, 'task3')
while not pq.is_empty():
    print(pq.pop())
Output-
task2
task3
task1
```

- ## Merge k Sorted Lists

  Explanation - You are given k sorted linked lists. Merge them into a single sorted linked list and return it.

  ### Algorithm -

  To merge k sorted linked lists into a single sorted linked list, we can use a min-heap to efficiently find and merge the smallest element from all the lists. The basic idea is to create a priority queue (min-heap) of size k, where we initially add the head of each linked list. Then, in each iteration, we extract the smallest node from the priority queue, add it to the merged result, and advance its pointer to the next node. We continue this process until all lists are exhausted.

```
import heapq
def mergeKLists(lists):
    # Initialise the priority queue
    min_heap = []
    for i, lst in enumerate(lists):
```

```
        if lst:
            # Add the tuple (node value, list index, node)
to the min-heap
            heapq.heappush(min_heap, (lst.val, i, lst))
            # Move to the next node in the list
            lists[i] = lst.next

    # Initialise the dummy node to simplify the merging
process
    dummy = ListNode(0)
    current = dummy

    while min_heap:
        # Pop the smallest node from the min-heap
        val, idx, node = heapq.heappop(min_heap)
        # Add the node to the merged result
        current.next = node
        current = current.next

        # If the list from which the node was taken still
has nodes, add the next node to the min-heap
        if lists[idx]:
            heapq.heappush(min_heap, (lists[idx].val, idx,
lists[idx]))
            lists[idx] = lists[idx].next

    return dummy.next
```

- ## K Most Frequent Elements

Explanation - Given an array of integers, you are asked to find the K most frequent elements in the array.
Here's a general approach to solving this problem:

1. Create a dictionary or hashmap to store the frequency of each element in the array.
2. Traverse the array and update the frequency count in the dictionary.
3. Create a min-heap (priority queue) that will store the K most frequent elements based on their frequencies. In Python, you can use the heapq module for this purpose.
4. Iterate through the dictionary and add elements to the min-heap. If the size of the min-heap exceeds K, remove the smallest element (i.e., the element with the lowest frequency).
5. After iterating through the dictionary, the min-heap will contain the K most frequent elements. You can retrieve these elements in decreasing order of frequency.

```python
import heapq
from collections import defaultdict
def k_most_frequent(nums, k):
    freq_map = defaultdict(int)
    for num in nums:
        freq_map[num] += 1
    min_heap = []
    for num, freq in freq_map.items():
        heapq.heappush(min_heap, (freq, num))
        if len(min_heap) > k:
            heapq.heappop(min_heap)
    result = [num for freq, num in min_heap]
    result.reverse()
    return result
```

- ## Kth Smallest and Largest Element of Array

Explanation - To find the Kth smallest and Kth largest elements in an array using heaps, you can use a min-heap to find the Kth smallest element and a max-heap to find the Kth largest element. Here's how you can approach this problem:

Kth Smallest Element:

1. Create a max-heap.
2. Iterate through the array and add elements to the max-heap.
3. If the size of the max-heap exceeds K, remove the maximum element from the heap.
4. After iterating through the array, the max-heap will contain the K smallest elements. The top of the max-heap will be the Kth smallest element.

Kth Largest Element:

1. Create a min-heap.
2. Iterate through the array and add elements to the min-heap.
3. If the size of the min-heap exceeds K, remove the minimum element from the heap.
4. After iterating through the array, the min-heap will contain the K largest elements. The top of the min-heap will be the Kth largest element.

```python
import heapq

def kth_smallest_and_largest(nums, k):
    min_heap = nums[:k]
    heapq.heapify(min_heap)

    for num in nums[k:]:
        if num > min_heap[0]:
```

```
            heapq.heappop(min_heap)
            heapq.heappush(min_heap, num)

    kth_smallest = min_heap[0]

    max_heap = [-num for num in nums[:k]]
    heapq.heapify(max_heap)

    for num in nums[k:]:
        if num < -max_heap[0]:
            heapq.heappop(max_heap)
            heapq.heappush(max_heap, -num)

    kth_largest = -max_heap[0]

    return kth_smallest, kth_largest
```

More questions on Priority Queues

1. K Largest Element
2. Merge K Sorted Arrays
3. Matrix Median
4. Median in a stream
5. Hungry Ninja
6. Kth smallest element in an unsorted array
7. Find K-Th Element From Product Array.
8. Sort Array
9. Implement a priority queue
10. Ninja And Stops