Priority Queue in Python

Definition:

A Priority Queue is an abstract data type similar to a regular queue or stack data structure, but with each element having a priority assigned. Elements with higher priorities are dequeued before elements with lower priorities, regardless of their order of insertion.

Advantages:

- Efficient retrieval of the highest-priority element.
- Useful for scenarios where elements need to be processed based on priority rather than the order of arrival.

Disadvantages:

- Implementations may vary in efficiency depending on the underlying data structure used (e.g., list, heap).
- Insertion and deletion operations can be slower compared to queues or stacks due to priority management.

Use Cases:

- **Job Scheduling:** Process jobs based on priority.
- **Event Handling:** Handle events in real-time systems.
- **Dijkstra's Algorithm:** Finding the shortest path in graphs.
- **Huffman Coding:** Efficient data compression technique.

Applications:

- **Operating Systems:** Task scheduling.
- **Networking:** Packet scheduling.
- **Data Compression:** Huffman coding.
- **Graph Algorithms:** Shortest path algorithms.

Time and Space Complexity:

- **Time Complexity:**
  - Insertion: O(log n)
  - Deletion of highest priority element: O(log n)
  - Peek (accessing highest priority): O(1)
- **Space Complexity:** O(n)

Methods:

- `put(item, priority)`: Inserts an item with a specified priority.
- `get()`: Removes and returns the item with the highest priority.
- `peek()`: Returns the item with the highest priority without removing it.

Operations:

- **Insertion:** Add an element with its priority.
- **Deletion:** Remove the element with the highest priority.
- **Peeking:** View the element with the highest priority without removing it.

Comparison with Other Data Structures:

Data Structure	Priority Queue
Queue	Elements are processed in FIFO order.
Stack	Elements are processed in LIFO order.
Heap	Efficiently manages highest (or lowest) priority elements.

Interesting Facts:

- Priority Queues can be implemented using different underlying data structures like heaps or lists.

- They are commonly used in algorithms for graph traversal and pathfinding.

Example Use Cases with Problem-Solving Questions:

1. **Job Scheduling:**
    - Problem: You have a list of jobs with priorities (e.g., deadlines). How would you use a priority queue to schedule these jobs to meet deadlines efficiently?
2. **Event Handling:**
    - Problem: Implement an event handler that processes events based on their priority, ensuring high-priority events are handled first.
3. **Dijkstra's Algorithm:**
    - Problem: Use a priority queue to implement Dijkstra's algorithm for finding the shortest path in a graph with weighted edges.

These examples demonstrate the versatility and practical applications of priority queues in various domains requiring efficient priority-based processing.

Priority Queue is an extension of the queue with the following properties.

1. An element with high priority is dequeued before an element with low priority.
2. If two elements have the same priority, they are served according to their order in the queue.

Various applications of the Priority queue in Computer Science are:

Job Scheduling algorithms, CPU and Disk Scheduling, managing resources that are shared among different processes, etc.

Key differences between Priority Queue and Queue:

1. In Queue, the oldest element is dequeued first. While, in Priority Queue, an element based on the highest priority is dequeued.
2. When elements are popped out of a priority queue the result obtained is either sorted in Increasing order or in Decreasing Order. While, when elements are popped from a simple queue, a FIFO order of data is obtained in the result.

priority_queue.py U

Data Structures > Priority Queue > priority_queue.py > ...

```python
class Priority_Queue:
    #declaring empty queue
    def __init__(self):
        self.queue = []

    #inserting elements in the queue
    def enqueue(self, item, priority):
        self.queue.append((priority, item))
        self.queue.sort(reverse = True)  #higher priroity elements come first

    #removing elements from the queue
    def dequeue(self):
        if self.is_empty():
            print("Queue is empty")
            return None
        return self.queue.pop()[1]

    #checking if queue is empty
    def is_empty(self):
        return len(self.queue) == 0

    #displaying the queue
    def display(self):
        if self.is_empty():
            print("Queue is empty")
        else:
            print("Elements in priority Queue are: ", end = " ")
            for item in self.queue:
                print(item[1], end = " ")
            print()


#Usage
pq = Priority_Queue()
```

```python

#inserting elements in the queue
pq.enqueue("A", 3)
pq.enqueue("B", 2)
pq.enqueue("C", 1)

#displaying the queue
pq.display()

#removing elements from the queue
print("Removed element: ", pq.dequeue())
pq.display()
```