**Are there any issues concerning the fair treatment of the tasks?**

This task management system divides tasks into high-priority (stack) and normal-priority (queue) categories. While this separation allows high-priority tasks to be processed first, it may lead to potential fairness issues. High-priority tasks continuously arriving can potentially starve normal-priority tasks, causing them to wait indefinitely. To address fairness, one possible solution is to implement a periodic check or time-based processing for normal-priority tasks. A scheduler or timer could periodically check the queue and process normal-priority tasks, preventing them from being delayed indefinitely.

**Do we need stack and queue both to solve the given problem?**

Using both a stack and a queue is justified in this context. Stacks are ideal for managing high-priority tasks because they ensure the most critical tasks are processed immediately. Conversely, Queues are suitable for normal-priority tasks and ensure a first-come, first-served order. These data structures complement each other to manage tasks with varying priorities efficiently.

**How are such systems usually built in practice? What data structures are used?**

Real-world task management systems typically employ a combination of data structures to effectively handle tasks with different priorities. High-priority tasks are often managed using data structures like priority queues, allowing quick retrieval of the highest-priority task. Normal-priority tasks may be processed using queues or other data structures that ensure tasks are processed in the order they are received.

**Is there any possible best solution still using stacks and queues for fair treatment of task** **processing?**

To better balance between high and normal-priority task processing, a more sophisticated approach might involve using stacks and queues within a priority-aware system. Such a system would continually monitor the state of the stacks and queues and periodically shift processing from high-priority to normal-priority tasks to ensure fairness. For example, for every **k** number of high-priority tasks processed, **n** number of normal-priority tasks are processed where **k** > **n**. Another possible solution is multithreading.

Multithreading offers a viable solution to improve fairness and efficiency in task processing when utilizing stacks and queues. Multiple threads can be employed to concurrently process tasks, which results in a reduction of task waiting times. This approach is particularly advantageous for tasks with varying priorities. Creating separate thread pools for high-priority and normal-priority tasks makes continuous monitoring of their respective data structures, such as a stack or a queue, possible. As tasks become available, the dedicated threads can process them promptly. Proper allocation of the appropriate number of threads to each pool, along with synchronization mechanisms, ensures fairness and prevents any form of task starvation. This approach optimizes task processing and boosts the performance of the task management system.

**Can the application be built using an alternative data structure, such as Deque alone instead of stack and queue? If so, describe the changes you would make referring to the line numbers in your code of the problem number 1 above.**

Yes, the application can be built using a Deque data structure alone to combine the behaviour of both a stack and a queue while considering task priority. It can be achieved by inserting high-priority tasks at the front of the Deque to prioritize them (explained below) and normal-priority tasks at the rear to keep the order of arrival. Here are the changes I would make:

* Instead of using separate Stack and Queue classes, I would use a single Deque to handle both tasks.
* I would define my Deque class with these methods (only methods with atypical behaviour). From the pseudocode, my Deque functions like a merge of the priority stack implemented in my code and a regular FIFO queue.
* However, the push() method now works based on priority and order, such that high-priority tasks are always inserted at the front of the Deque, with the highest priority task being the front. Normal-priority tasks are always inserted at the rear, after the lowest high-priority task, in order of how they are inserted (this process is similar to enqueueing from the front or rear based on some condition).

**Method Push(task):**

if task.getPriority() > 5:

if Deque is empty or task's priority is higher than or equal to the current front task:

Push the task to the front (top) of the Deque.

else:

Pop the current front of the Deque.

Place the new high-priority task at the front of the Deque.

Put the popped front back onto the front of the Deque

else:

Push the task to the rear of the Deque.

* The pop() differs from the conventional pop() by not just taking the front task or the top element. When a quarter of Deque's current length has been popped from the front (high-priority tasks), it actively seeks to balance the priority levels. If there are still high-priority tasks in the Deque, it transitions to the first normal-priority task and pops a quarter of the normal-priority tasks. When this is done, it returns to the front (high-priority tasks), and once again, a quarter of the Deque's current length is popped from the front (high-priority tasks). It then transitions to the first normal-priority task, and so on. This approach maintains a fair distribution of task processing, ensuring that high-priority tasks do not overshadow normal-priority tasks. This method facilitates efficient task management by offering priority-based access and preventing task overshadowing in the Deque structure.

**Method Pop():**

if Deque is empty:

Print "Deque is empty" and return null.

Get the task at the front of the Deque.

Remove the task from the front and increment the front index.

If a quarter of the Deque's length has been popped:

Find the end index of high-priority tasks.

if there are high-priority tasks remaining:

Move to the first normal-priority task.

Pop a quarter of the normal-priority tasks (*this will prevent normal-priority tasks overshadowing*).

Rotate back to the high-priority tasks, and repeat this process.

Ideally, I would not change any lines of my current program as I would be creating an entirely new class (Deque) to handle the storage of the tasks. Based on the user's input, this class would be instantiated in my TaskManager class, to which I would add tasks to and process from. As such, I would change the only lines of my program would be in my TaskManager class. These lines are:

* Lines 9 – 11, 14 – 16, 19 – 24, where I instantiated my Stack and Queue using either default values or user-chosen values
* Lines 52 – 56, where I pushed or enqueued tasks to my stack and queue based on the task's priority.
* In lines 61 – 69, I processed tasks by setting the status to COMPLETED and popped/dequeued them from my stack or queue, respectively. Normal-priority tasks were only processed after all high-priority tasks had been processed.

A sample (almost successful) implementation of my Deque class and pop() and push() methods are provided in my submission folder.