Name : Wei Nee Long
Class : MET CS521 – Spring2
Date : 4/4/2023
Title : MET CS526 – Project Documentation

In MET CS 526: Data Structures and Algorithms, I successfully completed the term project, which involved creating a small simulation program reflecting the basic operations of a process scheduler.

To implement the program, I created D as a HeapAdaptablePriorityQueue filled with `Entry <int arrivalTime, ProcessObject>`. I selected this data structure because it offers a ".min" method that facilitates the retrieval of the minimum entry based on the key, making it easier to compare the arrivalTime to the current time using the following code.

```
if (D_pq.min().getKey() == currtime):
        // move ProcessObj to Q as Entry < priority, ProcessObj>
```

I initially created D as an ArrayList. However, I learned that creating D as an ArrayList would require iterating through the list to identify the ProcessObj with the minimum arrivalTime. Doing so would introduce O(n^2) runtime, which would compromise the code's readability and efficiency.

For processes that had equal priority, it may have been better to execute the process with earlier arrival time instead of choosing arbitrarily. At a high level, this can be done by using radix sorting. First, sort the process objects Q using their arrivalTime and then sort them again using priority.

For example, consider the process objects shown below, where the top object is executed first, and the bottom object is the last to be sorted.

Original Q	id = 1	priority = 2	duration = 25	arrival time = 17
	id = 2	priority = 1	duration = 15	arrival time = 10
	id = 3	priority = 1	duration = 17	arrival time = 26
	id = 4	priority = 2	duration = 17	arrival time = 30
After first sort using arrivalTime	id = 2	priority = 2	duration = 25	arrival time = 10
	id = 1	priority = 1	duration = 15	arrival time = 17
	id = 3	priority = 1	duration = 17	arrival time = 26
	id = 4	priority = 2	duration = 17	arrival time = 30
After the second sort using priority	id = 1	priority = 1	duration = 15	arrival time = 17
	id = 3	priority = 1	duration = 17	arrival time = 26
	id = 2	priority = 2	duration = 25	arrival time = 10
	id = 4	priority = 2	duration = 17	arrival time = 30

Alternatively, to solve this problem, we could also check the arrivalTime each time we encounter duplicate keys. This can be accomplished using the following pseudocode:

```
// When priority is reduced . . .
      // priorityKey must be unique
      // So, before updating Q with new priority for currObj, check. . .

      for (e : Q){
          if (e.priority == currObj.priority){
              if (currObj.arrTime < e.arrTime){
                  currObj.setPriority(priority-1);
              }
              else if (currObj.arrTime > e.arrTime){
                  currObj.setPriority(priority+1);
              }
              else{ // if arrival time is same, currObj has higher priorityfd
                  currObj.setPriority(priority-1);
              }
          }
      }
```

Some other changes I think would improve efficiency, readability or reusability of my code are:

- Creating individual methods for the subprocesses like moving processObj from D to Q, removing processObj from Q when runtime reaches 0, etc.
- Include clear descriptions for each of these methods.

I think there are several changes that can be made to improve the efficiency, readability, or reusability of my code, including creating separate methods for subprocesses like moving processObj from D to Q and removing processObj from Q when the runtime reaches 0. It is essential to provide clear descriptions for each of these methods.