Assignment – 29 (Queue) – Write-up

> This is pre-production content, posted in rush due to high demand from R-batch. **Does not have flowcharts.**

Algorithm

Algorithm for the 'queue' class:

Class queue {
 int data[30];
 int front, rear;

 Constructor:
 - Initialize 'front' and 'rear' to -1.

 emptyCheck Method:
 - If 'front' is -1, return 1 (indicating empty).
 - Otherwise, return 0 (indicating not empty).

 fullCheck Method:
 - If 'rear' is greater than or equal to 29 (queue size - 1), return 1 (indicating full).
 - Otherwise, return 0 (indicating not full).

 enqueue Method (int x):
 - If fullCheck() returns 1:
  - Display "Job queue is full."
 - Else:
  - If 'front' is -1, set 'front' to 0.
  - Increment 'rear'.
  - Store 'x' in 'data' at the 'rear' position.

 dequeue Method:
 - If emptyCheck() returns 1:
  - Display "Job queue is empty."
 - Else:
  - Retrieve the job at 'front' from 'data' into 'x'.
  - Increment 'front'.
  - Display "Job [x] has been deleted."

 display Method:
 - Display "Job queue is: [" followed by:
  - Loop from 'front' to 'rear':
   - Display 'data[i]' followed by " | ".
- Display "]" and a new line.

End of class

Algorithm for the 'main' function

1. Initialize integer variables: 'choice', 'job', and 'totalJobs'.
2. Create an instance of the 'queue' class named 'jobQueue'.

3. Prompt the user to enter the total number of initial jobs and store it in 'totalJobs'.

4. Loop from 'i' equals 0 to 'totalJobs - 1':
   a. Prompt the user to enter the 'i+1'-th job number and store it in 'job'.
   b. Enqueue 'job' into 'jobQueue' using the 'enqueue' method.

5. Start an infinite loop (while true):
   a. Display the "JOB QUEUE MENU" with options:
      - 1 -> Add job to queue
      - 2 -> Delete a job from queue
      - 3 -> Display queue
      - 4 -> Exit
   b. Prompt the user to choose an option (1-4) and store it in 'choice'.

   c. Use a switch-case statement based on 'choice':
      - Case 1:
        i. Prompt the user to "Add additional job" and store it in 'job'.
        ii. Enqueue 'job' into 'jobQueue' using the 'enqueue' method.
        iii. Display the queue before and after the operation.

      - Case 2:
        i. Dequeue a job from 'jobQueue' using the 'dequeue' method.
        ii. Display the queue before and after the operation.

      - Case 3:
        i. Display the queue using the 'display' method.

      - Case 4:
        i. Display a termination message.
        ii. Exit the program.

      - Default:
        i. Display "Please choose a valid option (1-4)."

6. End of the loop.

7. Return 0 to indicate a successful execution of the program.

End of the 'main' function.

Pseudocodes

Class queue:

```
  data: array of integers with size 30
  front: integer
  rear: integer


  Constructor queue():
    Set front to -1
    Set rear to -1


  Function emptyCheck():
    If front is equal to -1, return 1
    Else, return 0


  Function fullCheck():
    If rear is greater than or equal to 29 (assuming zero-based indexing), return 1
    Else, return 0


  Function enqueue(x):
    If fullCheck() returns 1:
      Display "Job queue is full."
    Else:
      If front is equal to -1:
        Set front to 0
      Increment rear by 1
      Store x in data at the rear position


  Function dequeue():
    If emptyCheck() returns 1:
      Display "Job queue is empty."
    Else:
      x = data[front]
      Increment front by 1
      Display "Job [x] has been deleted."


  Function display():
    Display "Job queue is: [ "
    For i from front to rear:
      Display data[i], " | "
    Display "]"

End of Class queue
```

Function main():

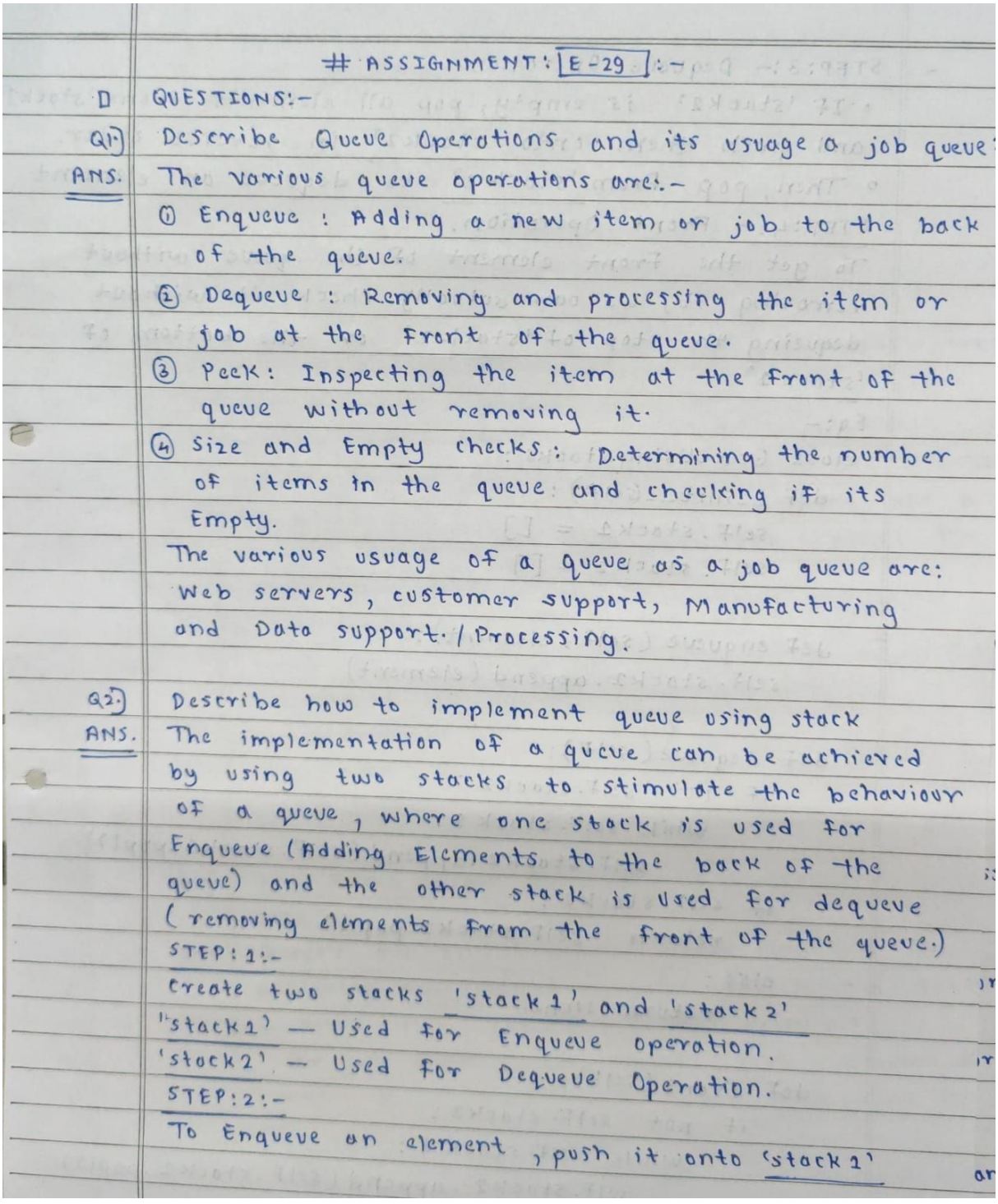
```
choice: integer
job: integer
totalJobs: integer
jobQueue: queue

Display "Enter number of jobs:"
Read totalJobs from user input

For i from 0 to totalJobs - 1:
  Display "Enter job number i+1:"
  Read job from user input
  Call jobQueue.enqueue(job)

While true:
  Display "----- JOB QUEUE MENU -----"
  Display "1 -> Add job to queue"
  Display "2 -> Delete a job from queue"
  Display "3 -> Display queue"
  Display "4 -> Exit"
  Display "Choose an option (1-4):"
  Read choice from user input

  Switch choice:
    Case 1:
      Display "Add additional job:"
      Read job from user input
      Call jobQueue.enqueue(job)
      Display "=============="
      Call jobQueue.display()
      Display "=============="
      Break

    Case 2:
      Call jobQueue.dequeue()
      Display "=============="
      Call jobQueue.display()
      Display "=============="
      Break

    Case 3:
      Display "=============="
      Call jobQueue.display()
      Display "=============="
      Break

    Case 4:
      Display "## END OF CODE"
```

Exit the program

Default:
Display "Please choose a valid option (1-4)."

Return 0
End of Function main

Answers



ASSIGNMENT: E-29 :-

D QUESTIONS:-

Q1) Describe Queue Operations and its usuage a job queue?

ANS. The various queue operations are:-

① Enqueue : Adding a new item or job to the back of the queue.

② Dequeue : Removing and processing the item or job at the front of the queue.

③ Peek: Inspecting the item at the front of the queue without removing it.

④ Size and Empty checks: Determining the number of items in the queue and checking if its Empty.

The various usuage of a queue as a job queue are: Web servers, customer support, Manufacturing and Data support. / Processing.

Q2) Describe how to implement queue using stack

ANS. The implementation of a queue can be achieved by using two stacks to stimulate the behaviour of a queue, where one stack is used for Enqueue (Adding Elements to the back of the queue) and the other stack is used for dequeue (removing elements from the front of the queue.)

STEP: 1:-

Create two stacks 'stack 1' and 'stack 2'

'stack 1' — Used for Enqueue operation.

'stack 2' — Used for Dequeue Operation.

STEP: 2:-

To Enqueue an element, push it onto 'stack 1'

STEP:3:- Dequeue Operation.

- If 'stack2' is empty, pop all elements from 'stack1' and push them onto 'stack2' in reverse order.
- Then, pop from 'stack2' to dequeue an element

STEP:4:- Front Operation.

To get the front element of the queue without removing it, we can actually check it without dequeing the top of 'stack2' or the bottom of 'stack1'

Eg:-

```
class QueueUsingStacks :
  def __init__(self) :
      self.stack1 = []
      self.stack2 = []

  def enqueue (self, element):
     self.stack1.append (element)

  def dequeue (self):
     if not self.stack2 :
        while self.stack1 :
           self.stack2.append (self.stack1.pop())
     if self.stack2 :
        return self.stack2.pop ()
     else :
        return None.

  def front (self):
     if not self.stack2 :
        while self.stack1 :
           self.stack2.append (self.stack1.pop())
```

```
        if self.stack2
            return self.stack2[-1]
        else
            return None.
```

Q3.) What do you mean by Linear Data structures? Give Examples of it.

ANS.
- Linear Data structures are a type of Data structure in Computer science that organize and store data elements in a linear or sequential manner.
- In other words, the Data elements are arranged in a specific order, and each element has a unique predecessor and successor. (except for the first and last elements.)
- Linear data structures are often used for the tasks that involve processing data sequentially.
- Examples of Linear Data structure.

→ i.) Array
ii.) Linked List.
iii.) Stack
iv.) Queue.
v.) Vectors / Lists.

Q4.) Why Queue is an Efficient Data structure to assign Jobs?

ANS. Queue's are an Efficient Data structure for assigning and managing jobs for several reasons:

1. First-in - First-Out (FIFO) Principle. The job waiting the longest gets processed first which or is a fair and predictable way to manage Job assignments.
2. Orderly Processing: Jobs are processed and organized in a systematic order they were recieved. This can

help maintain a structured and organized workflow.

3. Scalability: Queues are scalable, and additional jobs can be added to the queue as they arrive.
4. Reliability: Queues can help ensure that no job is overlooked or forgotten. All jobs are added to the queue, and they remain in a queue until they are successfully processed, reducing the risk of job loss or omission.
5. Error handling: In case of failures or errors during job processing, queues can provide mechanisms for handling these situations, such as requeuing or moving jobs to a dead letter queue for further analysis.
6. Synchorization.
7. Effective Prioritization
8. Concurrency.
9. Load Balancing.
10. Monitoring and Analytics.

These Factors make queue an Efficient Data structure.