**Experiment 4**

(PART B : TO BE COMPLETED BY STUDENTS)

***(Students must submit the soft copy as per following segments within two hours of the practical. The soft copy must be uploaded on the Blackboard or emailed to the concerned lab in charge faculties at the end of the practical in case the there is no Black board access available)***

|  |  |
| --- | --- |
| Roll No. C013 | Name: Ashmit Jain |
| Class : B | Batch : B1 |
| Date of Experiment: 16/08/2024 | Date of Submission: 16/08/2024 |
| Grade : | Time of Submission: |
| Date of Grading: |  |

**B.1 Software Code written by student: (Task 1)**

***(Paste your code completed during the 2 hours of practical in the lab here)***

**Task1:**



**Task2:**

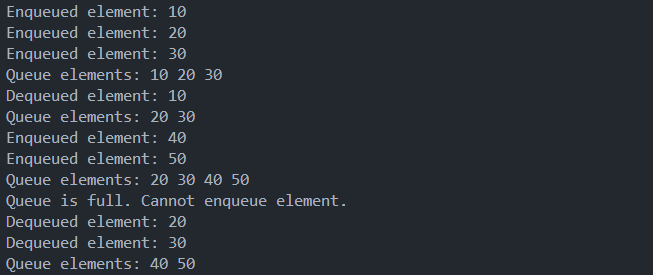
****

****

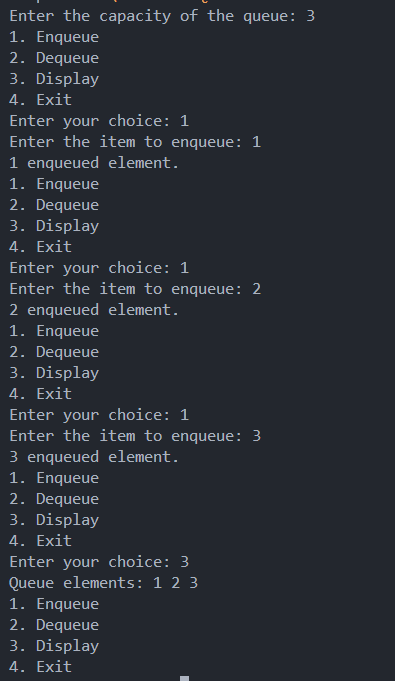
**B.2 Output:**

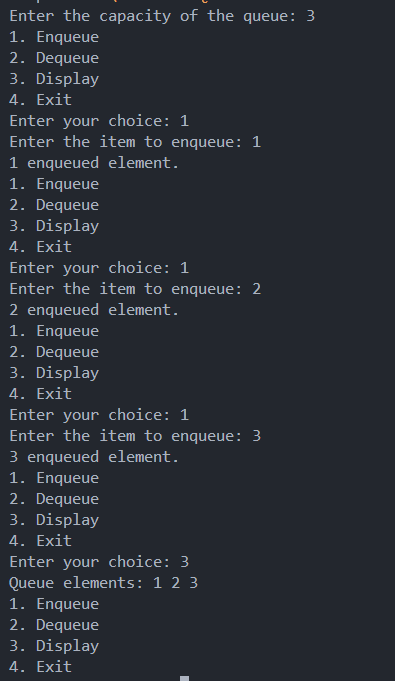
***(Paste your program input and output in following format, If there is error then paste the specific error in the output part. In case of error with due permission of the faculty extension can be given to submit the error free code with output in due course of time. Students will be graded accordingly.)***

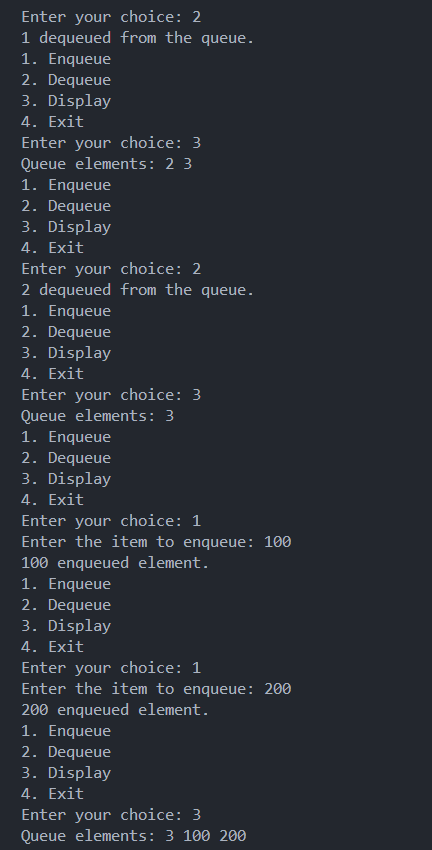
**Task1:**

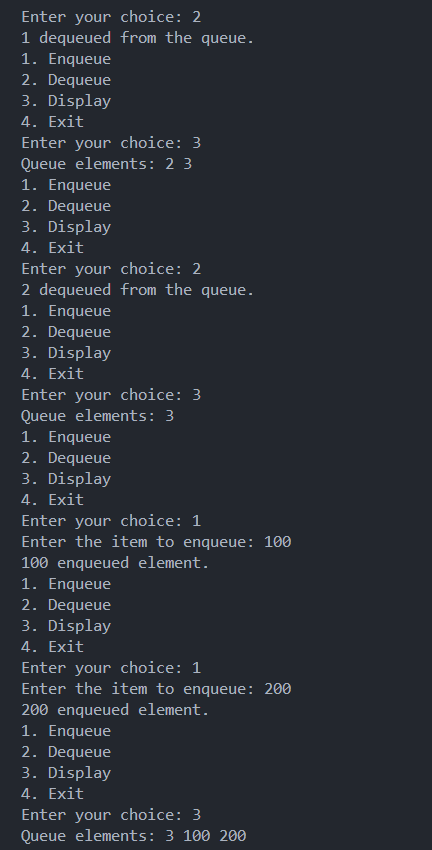
****

**Task2:**

****

****

****

****

**B.3 Observations and learning [w.r.t. all tasks]:**

***(Students are expected to comment on the output obtained with clear observations and learning for each task/ sub part assigned)***

**Data Structures Used:**

1. **Array:**
   * Both programs use a dynamically allocated array (int\* arr or int\* queue) as the underlying data structure to implement the queue.
   * The array size is determined by the user or fixed at runtime, and memory is allocated accordingly.
2. **Queue:**
   * The programs implement a queue, which is a linear data structure that follows the **First-In-First-Out (FIFO)** principle.
   * The queue operations implemented include enqueue (to add an element to the rear of the queue), dequeue (to remove an element from the front), and display (to show the current elements in the queue).

**Key Observations and Learning:**

1. **Memory Management:**
   * The use of dynamic memory allocation (new and delete) is essential when implementing data structures that require variable sizes. This allows flexibility but also requires careful memory management to avoid leaks.
   * The destructor (~Queue) ensures that the dynamically allocated memory is properly deallocated, preventing memory leaks.
2. **Circular Queue Implementation:**
   * The first program implements a circular queue. The circular queue solves the problem of unused space that occurs in a linear queue after several dequeue operations by using modulo arithmetic to wrap around the array.
   * This ensures efficient use of the array and allows enqueueing at the beginning of the array once the end is reached, as long as there is free space.
3. **Linear Queue Implementation:**
   * The second program uses a simple linear queue, where the front and rear pointers are incremented linearly.
   * In this implementation, the queue can fill up and become unusable even if there are empty slots at the beginning of the array (due to the front pointer moving forward), leading to inefficient use of space.
4. **Edge Case Handling:**
   * Both implementations handle edge cases such as trying to enqueue when the queue is full and dequeue when it is empty, ensuring the program remains robust.
5. **Queue Resetting:**
   * In both implementations, when all elements are dequeued, the front and rear pointers are reset to -1. This is crucial for reusing the queue without unnecessary memory operations.
6. **Efficiency Considerations:**
   * The circular queue is generally more efficient in terms of space utilization than the linear queue because it makes better use of the array by reusing spaces freed by dequeue operations.
7. **Practical Implementation:**
   * These implementations provide a foundational understanding of how queues work in practice, including the trade-offs between different methods of implementation (circular vs. linear).

**What I Learned:**

* **Understanding of Queue Operations:** The fundamental operations of a queue (enqueue, dequeue, and display) and how they can be implemented using arrays.
* **Importance of Edge Case Handling:** Ensuring that the queue behaves correctly when it is full or empty, and understanding how to reset the queue for further use.
* **Circular vs. Linear Queues:** The circular queue is more efficient in terms of space, while the linear queue is simpler but can lead to wasted space.
* **Memory Management in C++:** The importance of dynamically allocating and deallocating memory to manage resources effectively, especially in data structures.
* **C++ Specific Constructs:** Familiarity with C++ syntax and constructs such as classes, constructors, destructors, and dynamic memory management.

These programs provide a practical introduction to implementing basic data structures in C++ and highlight the importance of considering efficiency and proper memory management.

**B.4 Conclusion:**

*(****Students must write the conclusion as per the attainment of individual outcome listed above and learning/observation noted in section B.3)***

In conclusion, these programs provide a practical understanding of implementing queues using arrays in C++. The comparison between circular and linear queues highlights the importance of efficient space utilization and proper memory management. The circular queue is more efficient in handling space, while the linear queue is simpler but may lead to wasted space. Additionally, the importance of handling edge cases and resetting pointers is emphasized to ensure robust and reusable code. Overall, these implementations offer valuable insights into basic data structure operations and memory management in C++.

**B.5 Question of Curiosity**

***(To be answered by student based on the practical performed and learning/observations)***

**Answer Following Question**

1. A linear queue, if implemented using an array of size MAX\_SIZE, gets full when?  
   a) Rear == MAX\_SIZE – 1  
   b) Front == (rear + 1)mod MAX\_SIZE  
   c) Front == rear + 1  
   d) Rear == front

**Answer: A**

1. A linear queue, if implemented using an array of size MAX\_SIZE, gets empty when?  
   a) Front== MAX\_SIZE – 1  
   b) Rear == -1 Or FRONT > REAR  
   c) Front == rear + 1  
   d) Rear == front

**Answer: B**

1. A circular queue, if implemented using an array of size MAX\_SIZE, gets full when?  
   a) Rear == MAX\_SIZE – 1  
   b) Front == (rear + 1)mod MAX\_SIZE  
   c) FRONT = 0 and Rear = MAX – 1 ) || (Front==Rear +1 )  
   d) Rear == front

**Answer: C**

1. A circular queue, if implemented using an array of size MAX\_SIZE, gets empty when?  
   a) Front== MAX\_SIZE – 1  
   b) Rear == -1   
   c) Front == rear + 1  
   d) Rear == front

**Answer: B**

Write any two applications of queue data structure.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*