**EXPERIMENT NO. 5**

| **Objective(s):**  To implement a queue is to understand the basic operations of a queue data structure, |
| --- |
| **Outcome:**  Gain practical understanding and experience in working with a queue data structure, including its basic operations of enqueue, dequeue, traverse, and search. |
| **Problem Statement:**  Implement queue and its operations like enqueue, dequeue, traverse,search. |
| **Background Study:**  1. **Definition**:   * **Queue** is a linear data structure that follows the **FIFO** (First In, First Out) principle. * Elements are inserted at the **rear** (enqueue) and removed from the **front** (dequeue).   2. **Operations**:   * **Enqueue**: Adds an element to the rear of the queue. * **Dequeue**: Removes and returns the element from the front of the queue. * **Traverse**: Iterates through all elements in the queue. * **Search**: Checks if a specific element is present in the queue.   3. **Key Concepts**:   * **Front and Rear**: Pointers indicating the position of the first and last elements in the queue. * **Empty Queue**: A condition where no elements are present in the queue (front = NULL). * **Full Queue**: A condition where the queue cannot accept more elements due to memory constraints (rare in linked list-based implementations). |

| **Algorithm (Student Work Area):**  **1. Create Queue**   * + **Input:** Maximum size of the queue max\_size.   + **Output**: Pointer to the created queue.   + **Algorithm:**     - Allocate memory for the queue structure (Queue).     - Allocate memory for the array (array) inside the queue to hold elements.     - Initialize front and rear to -1.     - Return the pointer to the created queue.   **2. Check if Queue is Full (isFull(Queue \*q))**   * + **Input**: Pointer to the queue q.   + **Output**: true if the queue is full; otherwise, false.   + **Algorithm**:     - Return (q->rear == q->max\_size - 1).   **3. Check if Queue is Empty (isEmpty(Queue \*q))**   * + **Input**: Pointer to the queue q.   + **Output**: true if the queue is empty; otherwise, false.   + **Algorithm**     - Return (q->front == -1 && q->rear == -1).   **4. Enqueue (enqueue(Queue \*q, int value))**   * **Input**: Pointer to the queue q and element value to be added. * **Output**: None. * **Algorithm**:   + Check if the queue is full (isFull(q)).   + If isEmpty(q), set front to 0.   + Increment rear.   + Add value to q->array[q->rear].   **5. Dequeue (dequeue(Queue \*q))**   * **Input**: Pointer to the queue q. * **Output**: Element removed from the front of the queue, or -1 if the queue is empty. * **Algorithm**:   + Check if the queue is empty (isEmpty(q)).   + Get the element at q->array[q->front].   + If front equals rear, set both front and rear to -1.   + Otherwise, increment front.   **6. Peek (peek(Queue \*q))**   * **Input**: Pointer to the queue q. * **Output**: Front element of the queue, or -1 if the queue is empty * **Algorithm**:   + Check if the queue is empty (isEmpty(q)).   + Return q->array[q->front].   **7. Traverse (traverse(Queue \*q))**   * **Input**: Pointer to the queue q. * **Output**: None (prints the elements of the queue). * **Algorithm**:   + Check if the queue is empty (isEmpty(q)).   + Print all elements from q->array[q->front] to q->array[q->rear]   **8. Search (search(Queue \*q, int value))**   * **Input**: Pointer to the queue q and value to search. * **Output**: Index of value in the queue (0-based index relative to front), or -1 if not found. * **Algorithm**:   + Check if the queue is empty (isEmpty(q)).   + Iterate through the queue from q->front to q->rear.   + Return the index of value relative to front, or -1 if not found. |
| --- |

| **Code:** |
| --- |
| **OUTPUT :** |