**4\_ Queue**

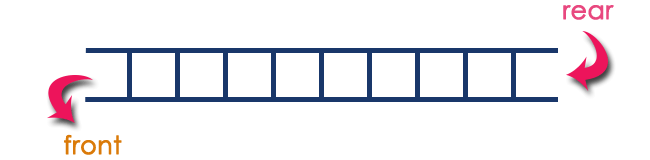
**4.1 Queue in Data Structure | Introduction to Queue**

|  |  |
| --- | --- |
| * It is a linear data structure. * It is a abstract data type * Queue can be implemented by **array, linklist, stack**. * This data algorithm follows **FIFO** (First In First Out) principle or **LILO** | **Rule:**   1. Insertion (will be performed from one end, and the end known as **rear** or **tail**)   insertion from **rear**   1. Deletion (will be performed from another end, and the end known as **head** or **front**)   deletion from **front**  Insertion = enqueue()  Deletion = dequeue() |
| **Logical representation of Queue:**  A whiteboard with writing on it  Description automatically generatedwhatever between front and rear is queue element and except that everything is garbage.  A whiteboard with black text  Description automatically generated  if front = rear (there is one element in the queue) | |
| **Empty queue condition:**   1. front = rear = -1 2. front > rear | **Queue Operation:**   1. enqueue() 2. dequeue() 3. front() / peek() 4. isFull() 5. isEmpty()   these operation time complexity is O(1) |
| **Application of queue:**   1. single shear resource (printer) 2. customer case call 3. processor (sharable resource) |  |

4.1 = done

**Queue Data Structure**

A **Queue Data Structure**is a fundamental concept in computer science used for storing and managing data in a specific order. It follows the principle of “**First in, First out**” **(FIFO)**, where the first element added to the queue is the first one to be removed. Queues are commonly used in various algorithms and applications for their simplicity and efficiency in managing data flow.

A screenshot of a computer

Description automatically generated

*Queue Data Structure*

**What is Queue in Data Structures?**

A queue is a linear data structure that follows the**First-In-First-Out (FIFO)** principle. It operates like a line where elements are added at one end (**rear**) and removed from the other end (**front**).

In a queue data structure, the insertion operation is performed using a function called "**enQueue()**" and deletion operation is performed using a function called "**deQueue()**".

Queue data structure can be defined as follows...

**Queue data structure is a linear data structure in which the operations are performed based on FIFO principle.**

A queue data structure can also be defined as

**"Queue data structure is a collection of similar data items in which insertion and deletion operations are performed based on FIFO principle".**

**Basic Operations of Queue Data Structure**

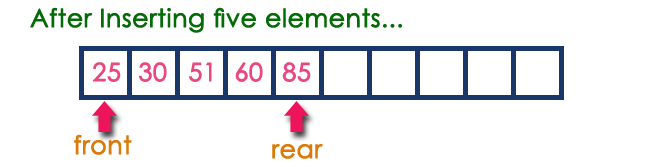
* **Enqueue (Insert)**: Adds an element to the rear of the queue. [ enQueue(value) ]
* **Dequeue (Delete)**: Removes and returns the element from the front of the queue. [ deQueue() ]
* **Peek**: Returns the element at the front of the queue without removing it.
* **isEmpty**: Checks if the queue is empty.
* **isFull**: Checks if the queue is full.
* **display()** - (To display the elements of the queue)

**Applications of Queue**

* Task scheduling in operating systems
* Data transfer in network communication
* Simulation of real-world systems (e.g., waiting lines)
* Priority queues for event processing queues for event processing

**Example**

Queue after inserting 25, 30, 51, 60 and 85.



Queue data structure can be implemented in three ways. They are as follows...

1. **Using Array**
2. **Using Linked List**
3. **Using Stack**

When a queue is implemented using an array, that queue can organize only a limited number of elements. When a queue is implemented using a linked list, that queue can organize an unlimited number of elements.

**4.2 Implementation of Queue using Arrays**

enqueue() and dequeue() operation must have the time complexity of O(1)

Operations with theirs cases

|  |  |
| --- | --- |
| **Insertion**  **Case: 🡪 rear** variable will work   1. **Full empty**   Condition **front** == -1 & **rear** == -1 then add to the array and then increment **rear** variable (**only one will happen this condition)**   1. **Has one or one+**   Else 🡪 add to array and then increment **rear** variable.   1. **Full occupied**   Condition **rear** == **size** -1 print “**Overflowed**” | **Deletion**  **Case**:  **🡪 front** variable will work   1. **Full empty**   Condition **front** == -1 & **rear** == -1 then print “**Underflowed**”   1. **Has one**   Condition rear == front then take these two to index 0   1. **Full occupied**   Return the deleted element or  Print the deleted item.  Increment front variable. |
| **Display**  **Case:**   1. **Check emptiness.**   then print “**Underflowed.**”   1. **Has elements**   Have a loop to print the elements. | **Peek**  **Case**:  **🡪 front** variable will work   1. **Full empty**   Condition **front** == -1 & **rear** == -1 then print “**Underflowed**”   1. **Full occupied / Has one**   Print the **peek** item. |
| **Pros:**  **Easy to Implement:** Arrays provide a straightforward way to implement a queue.  **Efficient Management of Large Data:** Arrays can handle a large amount of data efficiently.  **Fast Operations:** Insertion (enqueue) and deletion (dequeue) operations can be performed quickly as they follow the First In First Out (FIFO) rule. | **Cons:**  **Static Data Structure:** Arrays are static, meaning their size is fixed at the time of creation. This can lead to inefficiency if the queue does not use all the allocated space.  **Memory Utilization:** In an array implementation of a queue, if elements are removed from the front, the space cannot be reused for new elements. This can lead to poor memory utilization. |
| **Possible solution:**  **Circular Queue:** A circular queue is a variation of a queue that eliminates the issue of poor memory utilization in array implementation. In a circular queue, when we reach the end of the array, we circle back to the beginning of the array. This allows us to utilize the space left by dequeued elements.  **Dynamic Array:** Instead of using a static array, we can use a dynamic array that can grow and shrink in size as needed. This can help overcome the limitation of a fixed size.  **Deque (Double Ended Queue):** A Deque is a queue in which insertions and deletions are performed at both ends. This allows more flexibility than a regular queue.  **Priority Queue:** A priority queue is a special type of queue in which each element is associated with a priority and is served according to its priority. If elements with the same priority occur, they are served according to their ordering in the queue. | |

A screenshot of a computer

Description automatically generated

A screenshot of a computer

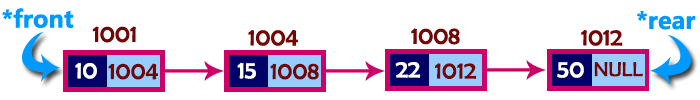
Description automatically generated

4.2 = DONE

**4.3 Queue Implementation using Linked List in C**

enqueue() and dequeue() operation must have the time complexity of O(1)

|  |  |
| --- | --- |
| **Pseudocode for the functions** | |
| Queue implementation with lisnklist required node implementation with two pointer **front** and **rear** as well as it’s destructor | |
| enqueue()   * 1. Create new node with passing data   2. Case – 1   Condition => full empty  Make both of the pointer point to the new node   * 1. Case – 2   Condition => else  Make **rear** point to the new pointer | peek()   1. Case -1   Condition => is the list full empty  Print “empty”   1. Case -2   Condition else  Print the current node data |
| denqueue()   1. Case -1   Condition => is the list full empty  Print “empty”   1. Case -2   Condition => always perform deletion from first node in 4 steps   1. Create a temporary variable and copy current node to it 2. Move one step ahead the current node 3. Point temporary node next to NULL pointer 4. Delete the temporary node   Another condition to check **if(fro == NULL) rear = NULL**  may not be strictly necessary for the dequeue operation itself, it is essential for maintaining the correctness, safety, and consistency of the queue implementation. | display()   1. Case -1   Condition => is the list full empty  Print “empty”   1. Case -2   Condition else  Create a temporary node to copy the current node  Loop 🡪 traverse the full list with the hep of temp and make it one step ahead in each loop and print each data |
| **Pros:**   1. **Dynamic Size:** Unlike arrays, linked lists do not need to have their size specified at creation, which can make them more flexible. 2. **Efficient Operations:** Operations like insertion and deletion (enqueue and dequeue) can be done in constant time, i.e., O(1). 3. **No Memory Wastage**: In a linked list, memory is allocated only when required, which can lead to efficient memory usage. | **Cons:**   1. **Increased Memory Usage:** Each node in a linked list also needs to store the address of the next node, which can lead to increased memory usage compared to arrays. 2. **Access Time:** Accessing elements in the middle of a linked list can be time-consuming as it requires traversal from the start of the list. 3. **Complexity:** Implementing a queue using a linked list can be more complex than using an array due to the need for handling pointers. |
| **Possible Solutions for the cons**   1. **Increased Memory Usage**: This is a trade-off for the dynamic size of linked lists. If memory usage is a concern, an array-based queue might be a better choice. 2. **Access Time:** Queues are typically used in scenarios where access is only needed at the front and rear, so this may not be a significant issue. If random access is required, a different data structure like an array or tree might be more suitable. 3. **Complexity:** Using libraries or built-in data structures available in many languages can help reduce the complexity of implementing a queue with a linked list. | |



A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

4.3 = DONE

**4.4 Circular Queue in Data Structure | Circular Queue using Arrays in C**

For practice do these operations

A white paper with black writing

Description automatically generated

Using mod operator for circular queue “ % ”

|  |  |
| --- | --- |
| **enqueue()**  **Case: 🡪 rear** variable will work   1. **If -> Full empty**   Condition **front** == -1 & **rear** == -1 then add to the array and then increment **rear** variable (**only one will happen this condition)**   1. **Else if -> rear + 1)%N) == fro**   Else 🡪 print **Overflow**   1. **Else ->** 2. rear = (rear + 1) % N;   updating **rear** variable in such a way that it comes back in circle   1. inputting data to queue | **dequeue()**  **Case**:  **🡪 front** variable will work   1. **Full empty**   Condition **front** == -1 & **rear** == -1 then print “**Underflowed**”   1. **Has one**   Condition rear == front then take these two to index 0   1. **Full occupied** 2. Return the deleted element or 3. Print the deleted item. 4. **front = (front + 1) % N** updating front variable to be go back to circle |
| **display()**   1. **Case: if**   **Check emptiness.**  then print “**Underflowed.**”   1. **else**   **Has elements**   1. Have a loop to print the elements. 2. Loop condition i is not equal to rear 3. Additionally, add an extra link to print the last element | **peek()**  **Case**:  **🡪 front** variable will work   1. **if -> Full empty**   Condition **front** == -1 & **rear** == -1 then print “**Underflowed**”   1. **else -> Full occupied / Has one**   Print the **peek** item. |
| **Pros:**   1. **Efficient Space Utilization:** Circular queues make the most of available space by reusing it once items are removed, avoiding wasted memory areas. 2. **No Need to Shift Elements:** When removing or adding items, there’s no need to move other elements around, which simplifies the process. 3. **Constant Time Operations:** Operations like insertion and deletion take a consistent amount of time, making performance predictable and stable. 4. **Reusable Memory Slots:** As items are processed, the space they occupied becomes immediately available for new items, which optimizes memory use. 5. **Suitable for Cyclic Processes:** They are ideal for situations where processes repeat in a cycle, such as scheduling tasks in an operating system. | **Cons:**   1. **Wastes Some Memory Space:** Circular queues can leave unused memory if the queue isn’t full, which isn’t efficient if storage is a concern. 2. **Fixed Size Limits Capacity:** Once created with a set amount of space, a circular queue can’t hold more than its maximum capacity, which can be limiting. 3. **Complexity in Implementation:** Setting up a circular queue involves careful tracking of the start and end points, which can be more complex than other data structures. 4. **Inefficient Memory Use When Full:** When a circular queue is full, it can lead to poor memory utilization as it doesn’t dynamically shrink or grow based on current needs. 5. **Difficult to Increase Size:** If you need a bigger queue than originally planned, expanding a circular queue’s size isn’t straightforward and often requires creating a new larger queue and transferring data. |
| **Possible solution:**   1. **Wastes Some Memory Space:** This can be mitigated by using a dynamic data structure like a linked list to implement the queue, which can grow and shrink as needed. 2. **Fixed Size Limits Capacity:** You can implement a dynamic circular queue that resizes itself when it gets full. This can be done by creating a new larger array and copying the elements from the old array to the new one. 3. **Complexity in Implementation:** Using libraries or built-in data structures available in many languages can help reduce the complexity of implementing a circular queue. 4. **Inefficient Memory Use When Full:** This can be addressed by using a dynamic circular queue that can grow and shrink as needed. 5. **Difficult to Increase Size:** As with the fixed size issue, implementing a dynamic circular queue that can resize itself when it gets full can solve this problem. | |

A screenshot of a computer

Description automatically generated

A screenshot of a computer

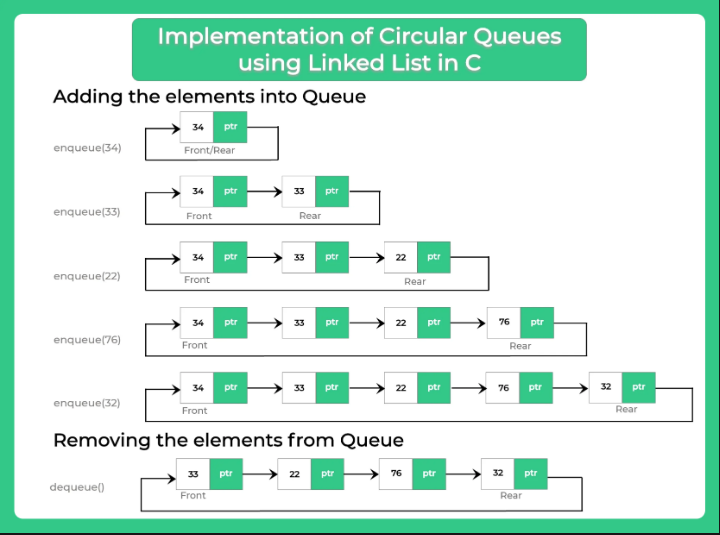
Description automatically generated

4.4 = DONE

**4.5 Circular Queue in Data Structure | Circular Queue using Linked List**

For practice do these operations

A white board with red writing

Description automatically generated 

|  |  |
| --- | --- |
| **Pseudocode for the functions** | |
| Queue implementation with lisnklist required node implementation with two pointer **front** and **rear** as well as it’s destructor | |
| enqueue()  Create new node with passing data   1. Case – 1   Condition => full empty  Make both of the pointer point to the new node  Make rear point to front   1. Case – 2   Condition => else  Make **rear** point to the new pointer  Make rear one stem ahead  Make rear point to front | peek()   1. Case -1   Condition => is the list full empty  Print “empty”   1. Case -2   Condition else  Print the current node data |
| denqueue()   1. Case -1 if   Condition => is the list full empty  Print “empty”   1. Case -2 else if -> rear is equal to front 2. Place NULL to both rear and front 3. Make temp’s next point to NULL 4. Delete temporary nide 5. Print “deleted value” 6. Make front equal to front next 7. Make rear next point to front 8. Make temporary node next point to NULL 9. And delete temporary node | display()   1. Case -1   Condition => is the list full empty  Print “empty”   1. Case -2   Condition else   1. Create a temporary node to copy the current node 2. Loop 🡪 until temp next is not equal to front 3. With each step forward print each data 4. Outside of the loop print the last data seperatly |
| **Pros:**   1. **Efficient Operations**:    1. Circular queues allow for quick and efficient traversal because the last node points back to the first node.    2. Useful for applications that require frequent traversal, such as queue and hash table implementations. 2. **Space Utilization**:    1. Circular queues make efficient use of available space by reusing it once items are removed.    2. This avoids wasted memory areas and ensures better space utilization. | **Cons:**   1. **Infinite Loop Risk**:    * If not handled with proper validation, circular queues can lead to infinite loops.    * Careful implementation is necessary to prevent this issue. 2. **Reversing the Queue**:    * Reversing a circular queue is not straightforward.    * Unlike regular queues, where reversing is relatively simple, circular queues require additional steps to achieve the same result. 3. **Finding the End of the List**:    * Locating the end of a circular queue can be harder compared to linear queues.    * Loop control and boundary conditions need careful consideration. |
| **Possible Solutions:**   1. **Sentinel Node**:    * Introduce a sentinel node (dummy node) at the end of the circular queue.    * This sentinel node simplifies operations like finding the end of the queue and handling boundary conditions. 2. **Doubly Linked Circular Queue**:    * Use a doubly linked list for the circular queue.    * Each node has pointers to both the next and previous nodes.    * This facilitates easier traversal in both directions and simplifies reversing the queue. 3. **Circular Queue with Dynamic Resizing**:    * Implement a circular queue that dynamically resizes itself when needed.    * When the queue becomes full or empty, allocate or deallocate memory as required.    * This ensures efficient space utilization and avoids overflow/underflow issues. | |

A screenshot of a computer

Description automatically generated

A screenshot of a computer

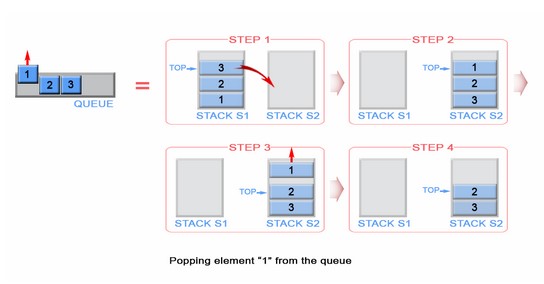
Description automatically generated

A screenshot of a computer program

Description automatically generated

4.5 = DONE

**4.6 Implement Queue using Stack in C**



|  |  |
| --- | --- |
| There are 2 methods of implementing queue using stack  🡪 One stack -> recursion  🡪Two stack -> normal approach (stack1 and stack2(temporary / supporting))    1st method dequeue is costly  2nd method enqueue is costly  We will have 1st approach |  |
| **Pros:**   * **Space Efficiency:** This method is space efficient as it uses only two stacks for implementing a queue, and no extra space is required. * **Flexibility:** Stacks are dynamic data structures, and they can grow as needed. | **Cons:**   * **Time Complexity:** The dequeue operation in this approach has a time complexity of O(n), which is not efficient for large datasets. |
| **Applications:**   * This algorithm can be used in scenarios where the order of execution is important. For example, in CPU scheduling, disk scheduling. * It can also be used in certain algorithms like breadth-first search(BFS) for graph data structures. | **Possible Solution for Cons:**   * **Amortized Analysis:** Although the worst-case time complexity for dequeue operation is O(n), if we take into account the number of times each element is manipulated (pushed or popped) in the entire process of enqueue and dequeue operations, the average time complexity is O(1), which is efficient. |

**Operation’s pseudocode**

|  |  |
| --- | --- |
| Create a class for Queue algorithm  Have **top1**, **top2** and **count** variable  And **stack1**, **stack2** as arrays | |
| **push1()**   1. Condition: if( top1 == N -1) 🡪 checking if the queue is full   Print the massage   1. Condition: else   Increment **top1**  Insert the passing value into **stack1** | **push2()**   1. Condition: if( top2 == N -1) 🡪 checking if the queue is full   Print the massage   1. Condition: else   Increment **top2**  Insert the passing value into **stack2** |
| **pop1()**  inserting **top1** inside of **stack1** and  then decrementing **top1** | **pop2()**  inserting **top2** inside of **stack2** and  then decrementing **top2** |
| **enqueue()**  call push1 function with a passing value  increment count variable | **dequeue()**  **initialize a and b variable**   1. Condition: if ( top1 == -1 && top2 == -1) checking if the queue is empty   Print the massage   1. Condition: else   For loop i < count   * a = pop1() * push2(a)   B = pop2()  Decrement count  For loop i < count   * a = pop2() * push1(a) |
| **display()**  same as usual  but the condition is I <= top1  can’t use count in the condition |  |





DONE = 4.6

**4.7 DEQUE in Data Structure | Introduction to DEQue - Double Ended Queue**

**Deque** (Double Ended Queue) is a type of queue in which insertion and removal of elements can be performed from both the front and the rear. It does not follow the FIFO (First In First Out) rule.

**How it is different from linear and circular queue**

**Linear Queue:**

* A linear queue is a **FIFO (First In First Out)** data structure[**1**](https://www.geeksforgeeks.org/difference-between-queue-and-deque-queue-vs-deque/).
* Elements can only be inserted at the **end** (rear) of the queue.
* Elements can only be removed from the **front** (head) of the queue.
* A linear queue can be implemented using an array or a linked list.

**Circular Queue:**

* A circular queue is a variation of the linear queue where the last element points to the first element, making a **circular link**.
* The main advantage of a circular queue over a simple queue is **better memory utilization**. If the last position is full and the first position is empty, we can insert an element in the first position.
* This action is not possible in a simple queue.

**Deque (Double Ended Queue):**

* A deque (double-ended queue) is a linear data structure that stores a collection of elements, with operations to add and remove elements from **both ends** of the deque.
* Deque can be used to implement the functionalities of both Stack (LIFO approach i.e., Last In First Out) and Queue (FIFO approach i.e., First In First Out).
* Deque can be implemented using a **circular array** or a **doubly linked list**.
* There are two types of deque:

1. **Input restricted deque** (deletion can be done from both the ends but insertion from the front will not be allowed) and
2. **Output restricted deque** (insertion can be done from both the ends but deletion from the rear will not be allowed).

|  |  |
| --- | --- |
| It supports both stack and queue property | There are two type of deque   1. Input – restricted  * Insertion is only allowed from one end * Deletion is allowed from both ends  1. Output-restricted  * Insertion is only allowed from both ends * Deletion is allowed from one end |
| **Deque operations**   1. insertAtFront () / enqueueAtFront() 2. deleteFromFront () / dequeueFromFront() 3. insertAtRear () / enqueueAtRear() 4. deleteFromRear () / dequeueFromRear() 5. getFront () 6. getRear () 7. isFull () 8. isEmpty () | implementation of Deque   1. circular array 2. doubly linklist   all the operation should take time complexity O(1) |
| **## Pros of Deque:**   1. **Versatility:** You can add and remove items from both the front and back of the queue. 2. **Speed:** Deques are faster in adding and removing elements to the end or beginning. 3. **Rotation Operations:** The clockwise and anti-clockwise rotation operations are faster in a deque. 4. **Dynamic Size:** Deques can grow or shrink dynamically. | **## Cons of Deque:**   1. **Memory Efficiency:** Deques are less memory efficient than a normal queue. |
| **## Applications of Deque:**   1. **Job Scheduling Algorithms:** It is used in job scheduling algorithms. 2. **Web Browser History:** Recently visited URLs are added to the front of the deque and the URL at the back of the deque is removed after some specified number of operations of insertions at the front. 3. **Software Application's List of Undo Operations:** Storing a software application’s list of undo operations. 4. **Graph Traversal Algorithms:** In graph traversal algorithms such as breadth-first search (BFS). BFS uses a deque to store nodes and performs operations such as adding or removing nodes from both ends of the deque. 5. **Task Management Systems:** In task management systems to manage the order and priority of incoming tasks. Tasks can be added to the front or back of the deque depending on their priority or deadline. 6. **Queueing Systems:** In queueing systems to manage the order of incoming requests. Requests can be added to the front or back of the deque depending on their priority or arrival time. 7. **Caching Systems:** In caching systems to cache frequently accessed data. Deques can be used to store cached data and efficiently support operations such as adding or removing data from both ends of the deque. | **## Possible Solutions to Cons:**   1. **Overflow and Underflow Conditions:** Always check for the underflow condition before performing delete operations. Attempting to delete from an empty deque can lead to errors. Similarly, check for the overflow condition while adding elements, if the deque has a size limit. |

DONE = 4.7

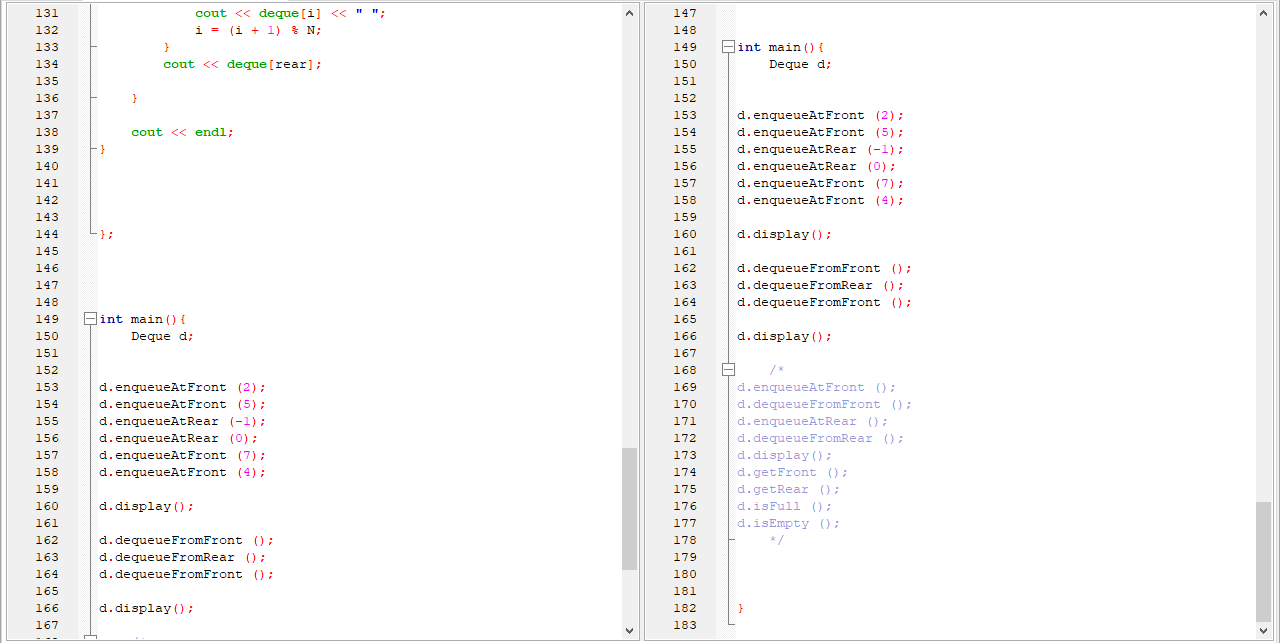
**4.8 Implementation of DEQUE using Circular Array**

**Operation’s pseudocode**

|  |  |
| --- | --- |
| Create a Deque class within **front**, **rear** variable and an array named **deque** add constructor | |
| insertAtFront () / enqueueAtFront()   1. Condition: if(front == rear + 1) || (front == 0 && rear == N-1)    1. checking if the queue is full    2. print the massage 2. Condition: if( **front** == -1 && **rear** == -1) checking if the queue is empty    1. **front = rear = 0**    2. **deque[front] = data** 3. Condition: else if( **front** == **0**)    1. **front = N -1;**    2. **deque[front] = data** 4. Condition: else    1. **front – -**    2. **deque[front] = data** | insertAtRear () / enqueueAtRear()   1. Condition: if(front == rear + 1) || (front == 0 && rear == N-1)    1. checking if the queue is full    2. print the massage 2. Condition: if( **front** == -1 && **rear** == -1) checking if the queue is empty    1. **front = rear = 0**    2. **deque[rear] = data** 3. Condition: else if( **front** == **0**)    1. **rear = 0**    2. **deque[rear] = data** 4. Condition: else    1. **rear ++**    2. **deque[rear] = data** |
| deleteFromFront () / dequeueFromFront()   1. Condition: if( **front** == -1 && **rear** == -1) checking if the queue is empty 2. Condition: else if( **front** == **rear** ) it has only one element   Print the deleted value **deque[front]**  **front = rear = -1**   1. Condition: else if( **front** == **N -1**)   Print the deleted value **deque[front]**  **front = 0**   1. Condition: else   Print the deleted value **deque[front]**  **front ++** | deleteFromRear () / dequeueFromRear()   1. Condition: if( **front** == -1 && **rear** == -1) checking if the queue is empty 2. Condition: else if( **front** == **rear** ) it has only one element   Print the deleted value **deque[rear]**  **front = rear = -1**   1. Condition: else if( **rear** == **0**)   Print the deleted value **deque[rear]**  **Rear = N -1**   1. Condition: else   Print the deleted value **deque[rear]**  **Rear--** |
| getFront ()   1. Condition: if( **front** == -1 && **rear** == -1) checking if the queue is empty 2. Condition: else   Print the **front** value of the **deque** array | getRear ()   1. Condition: if( **front** == -1 && **rear** == -1) checking if the queue is empty 2. Condition: else   Print the **rear** value of the **deque** array |
| display()   1. Condition: if( **front** == -1 && **rear** == -1) checking if the queue is empty 2. Condition: else   **i** = **front**  while loop   * condition **i** is not equals to **rear** * print each element * increment **i** with circular queue method( which include modulo operator)   print the last element of the queue | **Shortcut**  **enqueueAtRear 🡪 rear++**  **dequeueFromFront 🡪 front++**  **enqueueAtFront🡪front--**  **dequeueFromRear 🡪rear --** |

****





DONE = 4.8

DONE

**4\_ Queue Problems**

|  |  |  |  |
| --- | --- | --- | --- |
| Easy | | | |
| 1. Reverse First k Elements of Queue |  | 1. Level with maximum number of nodes |  |
| 1. Implement a Queue using an Array |  | 1. Breadth First Search or BFS for a Graph |  |
| 1. Print all elements of a queue in a new line |  | 1. Find Minimum Depth of a Binary Tree |  |
| Medium | | | |
| 1. Implement a Deque |  | 1. Detect cycle in an undirected graph using BFS |  |
| 1. Implement a Circular Queue |  | 1. Find next right node of a given key |  |
| 1. Check if a queue can be sorted into another queue using a stack |  | 1. Minimum steps to reach target by a Knight |  |
| 1. Implement Stack using Queues |  | 1. Islands in a graph using BFS |  |
| 1. Implement Stack using Two Queues |  | 1. Flood Fill Algorithm |  |
| 1. Implement Queue using Two Stacks |  | 1. Minimum steps to reach target by a Knight |  |
| 1. Design a Queue data structure to get minimum or maximum in O(1) time |  | 1. First negative integer in every window of size k |  |
| 1. Check whether a given graph is Bipartite or not |  | 1. Level order traversal in spiral form |  |
| 1. Print Right View of a Binary Tree |  | 1. Minimum time required to rot all oranges |  |
| 1. An Interesting Method to Generate Binary Numbers from 1 to n |  | 1. Queue based approach or first non-repeating character in a stream |  |
| 1. Implement a Queue using a Stack |  | 1. Shortest distance in a maze |  |
| 1. Reverse a queue using recursion |  | 1. Geek in a Maze |  |
| 1. Implement Priority Queue using Linked List |  | 1. Find shortest safe route in a path with landmines |  |
| 1. Implement Queue using Deque |  | 1. Find the first circular tour that visits all petrol pumps |  |
| 1. Flatten a multilevel linked list |  | 1. Connect Nodes at Same Level |  |
| Hard | | | |
| 1. Find the first non-repeating character from a stream of characters |  | 1. Maximum cost path from source node to destination |  |
| 1. Maximum of all subarrays of size k using a queue |  | 1. Trapping Rain Water |  |
| 1. Implement LRU Cache using Queue |  | 1. Maximum cost path from source node to destination |  |
| 1. Design a Queue data structure to get the maximum or minimum of sliding window |  | 1. Snake and Ladder Problem |  |
| 1. Find if there is a path between two vertices in a directed graph |  | 1. Minimum Cost Path in a directed graph via given set of intermediate nodes |  |
| 1. Design a Data Structure for LRU Cache |  | 1. Turn a Queue into a Priority Queue |  |
| 1. Trapping Rain Water |  | 1. Interchange elements of Stack and Queue without changing order |  |

|  |
| --- |
| Link: <https://www.geeksforgeeks.org/top-50-problems-on-queue-data-structure-asked-in-sde-interviews/> |

**Queue problem given by lab teacher.**

**Class Example :**

**Question 1 - Queue Implementation with Array:**

You are required to implement a queue data structure using an array in C. The queue should

have the following functionalities:

a) void enqueue(int item): This function should enqueue an integer item into the queue.

b) int dequeue(): This function should remove and return the front item from the queue.

c) int front(): This function should return the front item from the queue without removing it.

d) int isEmpty(): This function should return 1 if the queue is empty, and 0 otherwise.

Write the C code for the queue implementation with an array and demonstrate its usage by

enqueuing a few elements, dequeuing elements, and checking whether the queue is empty.

**Question 2 - Queue Implementation with Linked List:**

You are required to implement a queue data structure using a singly linked list in C. The queue

should have the following functionalities:

a) void enqueue(int item): This function should enqueue an integer item into the queue.

b) int dequeue(): This function should remove and return the front item from the queue.

c) int front(): This function should return the front item from the queue without removing it.

d) int isEmpty(): This function should return 1 if the queue is empty, and 0 otherwise.

Write the C code for the queue implementation with a linked list and demonstrate its usage by

enqueuing a few elements, dequeuing elements, and checking whether the queue is empty.

Please ensure that your implementations handle edge cases appropriately and provide the

correct output for different scenarios.

**Question 3 - Circular Queue Implementation with Array:**

You are required to implement a circular queue data structure using an array in C. The circular

queue should have the following functionalities:

a) void enqueue(int item): This function should enqueue an integer item into the circular queue.

b) int dequeue(): This function should remove and return the front item from the circular queue.

c) int front(): This function should return the front item from the circular queue without removing

it.

d) int isEmpty(): This function should return 1 if the circular queue is empty, and 0 otherwise.

e) int isFull(): This function should return 1 if the circular queue is full, and 0 otherwise.

Write the C code for the circular queue implementation with an array and demonstrate its usage

by enqueuing a few elements, dequeuing elements, checking whether the queue is empty, and

verifying if the queue is full.

Ensure that your implementation handles the circular nature of the queue correctly and provides

the correct output for various scenarios.

**Question 4 - Implement a Queue using Stacks:**

You are tasked with designing a queue data structure using two stacks. Each stack should have

push, pop, and isEmpty functions. The queue should support the following operations:

enqueue(int x): Add element x to the back of the queue.

dequeue(): Remove and retrieve the element from the front of the queue.

Dequeue constant : https://www.geeksforgeeks.org/queue-using-stacks/

Enqueue constant : Code uploaded

**Practice problem from different section**

1. Implement a stack using queue.
2. Reverse the elements of a queue.

**Input**: q[ ] = {10, 20, 30, 40, 50, 60}

**Output**: q[ ] = {60, 50, 40, 30, 20, 10}

**3.** Sort the elements of a queue without using any extra space.

**Input**: q[ ] = {20, 40, 10, 60, 50, 30}

**Output**: q[ ] = {10, 20, 30, 40, 50, 60}

**4.** Given a queue q[ ] and an integer K, remove the integer K from the queue. If

multiple same elements exist, remove the first one.

**Input**: q[ ] = {10, 20, 30, 40, 50, 60}, K = 30

**Output**: {10, 20, 40, 50, 60}

**Input**: q[] = {1, 2, 3, 3}, K = 3

**Output**: {1, 2, 3}