Queue

Queue is a specialized data storage structure (Abstract data type). Unlike, arrays access of elements in a Queue is restricted. It has two main operations **enqueue** and **dequeue**. Insertion in a queue is done using **enqueue** function and removal from a queue is done using **dequeue** function. An item can be inserted at the end ('rear') of the queue and removed from the front ('front') of the queue. It is therefore, also called First-In-First-Out (FIFO) list. Queue has five properties - capacity stands for the maximum number of elements Queue can hold, size stands for the current size of the Queue, elements is the array of elements, front is the index of first element (the index at which we remove the element) and rear is the index of last element (the index at which we insert the element).

Algorithm:

Queue structure is defined with fields capacity, size, *elements (pointer to the array of elements), front and rear.

Functions –

1. createQueue function– This function takes the maximum number of elements (maxElements) the Queue can hold as an argument, creates a Queue according to it and returns a pointer to the Queue. It initializes Queue Q using malloc function and its properties.
   elements = (int *)malloc(sizeof(int)*maxElements).
   Q->size = 0, current size of the Queue Q.
   Q->capacity = maxElements, maximum number of elements Queue Q can hold.
   Q->front = 0
   Q->rear = -1
2. enqueue function - This function takes the pointer to the top of the queue Q and the item (element) to be inserted as arguments. Check for the emptiness of queue
   If Q->size is equal to Q->capacity, we cannot push an element into Q as there is no space for it.
   Else, enqueue an element at the end of Q, increase its size by one. Increase the value of Q->rear to Q->rear + 1. As we fill the queue in circular fashion, if Q->rear is equal to Q->capacity make Q->rear = 0. Now, Insert the element in its rear side

   Q->elements[Q->rear] = element

3. dequeue function - This function takes the pointer to the top of the stack S as an argument.
   If Q->size is equal to zero, then it is empty. So, we cannot dequeue.
   Else, remove an element which is equivalent to incrementing index of front by

one. Decrease the size by 1. As we fill elements in circular fashion, if Q->front is equal to Q->capacity make Q->front=0.

4. front function – This function takes the pointer to the top of the queue Q as an argument and returns the front element of the queue Q. It first checks if the queue is empty (Q->size is equal to zero). If it's not it returns the element which is at the front of the queue.

Q->elements[Q->front]

Property:

1. Each function runs in O(1) time.
2. It has two basic implementations

   Array-based implementation – It's simple and efficient but the maximum size of the queue is fixed.

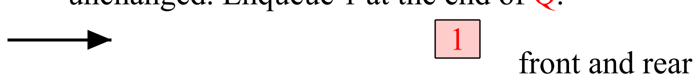
   Singly Linked List-based implementation – It's complicated but there is no limit on the queue size, it is subjected to the available memory.

Example:

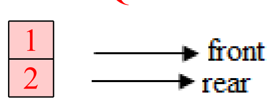
Create the queue Q using createQueue function, where Q is the pointer to the structure Queue. The maximum number of elements (maxElements) = 5.

Initially Q->size = 0, Q->capacity = 5, Q->front = 0, Q->rear = -1.

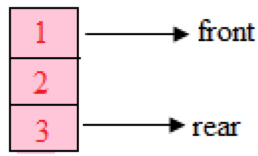
1. enqueue(Q,1): Since, Q->size ≠ Q->capacity. Increase its size by one and also Q->rear by 1. Now, size = 1 and Q->rear = 0. Since, Q->rear = Q->capacity, Q->rear remains unchanged. Enqueue 1 at the end of Q.



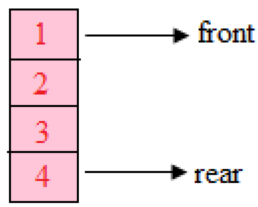
2. enqueue(Q,2): Since, Q->size ≠ Q->capacity. Increase its size by one and also Q->rear by 1. Now, size = 2 and Q->rear = 1. Since, Q->rear = Q->capacity, Q->rear remains unchanged. Enqueue 2 at the end of Q.



3. enqueue(Q,3): Since, Q->size ≠ Q->capacity. Increase its size by one and also Q->rear by 1. Now, size = 3 and Q->rear = 2. Since, Q->rear = Q->capacity, Q->rear remains unchanged. Enqueue 3 at the end of Q.

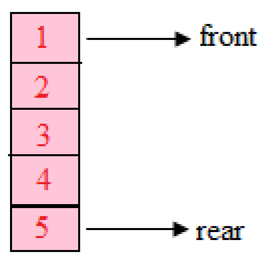


4. enqueue(Q,4): Since, Q->size ≠ Q->capacity. Increase its size by one and also Q->rear by 1. Now, size = 4 and Q->rear = 3. Since, Q->rear = Q->capacity, Q->rear remains unchanged. Enqueue 4 at the end of Q.

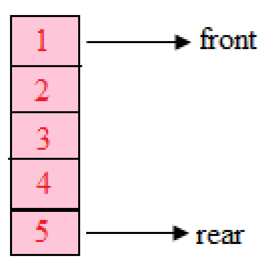


5. front(Q): Since, Q->size(=4) is not equal to zero. It returns the element at the front end of the queue i.e. 1. Output = '1'.

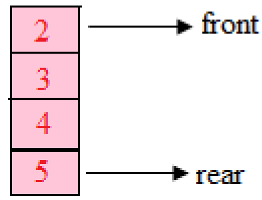
6. enqueue(Q,5): Since, Q->size ≠ Q->capacity. Increase its size by one and also Q->rear by 1. Now, size = 5 and Q->rear = 4. Since, Q->rear = Q->capacity, Q->rear remains unchanged. Enqueue 5 at the end of Q.



7. enqueue(Q,6): Since, Q->size = Q->capacity, we cannot push an element as there is no space for it. size = 5 and Q->rear = 4.



8. dequeue(Q): Since, Q->size(=5) is not equal to zero. It removes the front most element i.e. 1 by simply incrementing index of front by one. Decrease the size by 1. Now, size = 4 and Q->front = 1. Since, Q->front is not equal Q->capacity, Q->front remains unchanged.



9. front(Q): Since, Q->size(=4) is not equal to zero. It returns the element at the front end of the queue i.e. 2. Output = ‘2’.
10. dequeue(Q): Since, Q->size(=4) is not equal to zero. It removes the front most element i.e. 1 by simply incrementing index of front by one. Decrease the size by 1. Now, size = 3 and Q->front = 2. Since, Q->front is not equal Q->capacity, Q->front remains unchanged.