Queue Data Structure

Queue

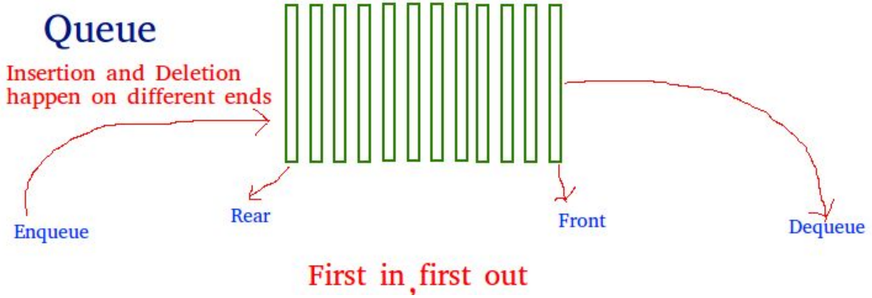
Queue is a linear data structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO).

Queue is an abstract data structure, somewhat similar to Stack. Unlike stack, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

LAST IN LAST OUT
ONE WAY
FIRST IN FIRST OUT

A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first.

- Queue is a data structure in which new elements are inserted at one end called ‘**rear of queue**’, and from which elements are removed at the other end called ‘**front of queue**’.
- Queue is also called First-In-First-Out (FIFO) list.
- In queue, the first inserted element will be the first removed element.
- Two basic queue operations are:
  - enqueue() − add (store) an item to the queue.
  - dequeue() − remove (access) an item from the queue.



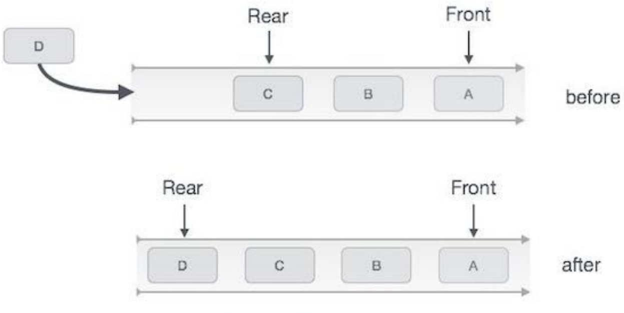
As in stack, a queue can also be implemented using Array and Linked List. Queue can either be a fixed size one (using array) or it may have a sense of dynamic resizing (using linked list).

Implementation of Queue using Array:

Enqueue Operation

Queues maintain two data pointers, front and rear. The following steps should be taken to enqueue (insert) data into a queue −

- Step 1 − Check if the queue is full.
- Step 2 − If the queue is full, produce overflow error and exit.
- Step 3 − If the queue is not full, increment rear pointer to point the next empty space.
- Step 4 − Add data element to the queue location, where the rear is pointing.
- Step 5 − return success.

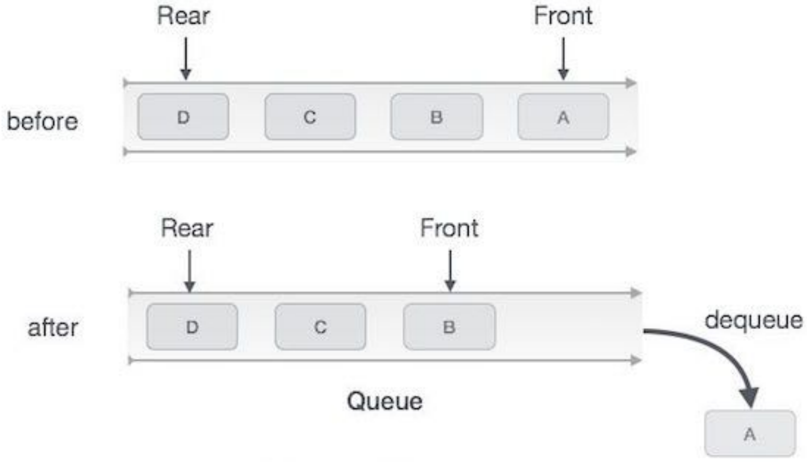


Queue Enqueue

Dequeue Operation

Accessing data from the queue is a process of two tasks − access the data where front is pointing and remove the data after access. The following steps are taken to perform dequeue operation −

- Step 1 − Check if the queue is empty.
- Step 2 − If the queue is empty, produce underflow error and exit.
- Step 3 − If the queue is not empty, access the data where front is pointing.
- Step 4 − Increment front pointer to point to the next available data element.
- Step 5 − Return success.



Queue Dequeue

Implementation of Enqueue Operation in C

```c
void enqueue(int data) {

   if(rear == MAXSIZE-1)

      printf("Could not insert data, Queue is full.\n");

    else {

        rear = rear + 1;

        queue[rear] = data;

    }

}
```

Implementation of Dequeue Operation in C

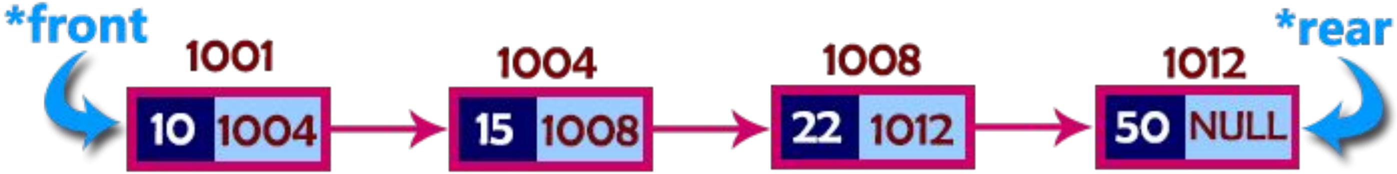
```
int dequeue() {

    int data = -1;

    if(front > rear)

         printf("Could not retrieve data, Queue is empty.\n");

    else {

        data = queue[front];

        front = front + 1;

    }

    return data;

}
```

Implementation of Queue using Linked list:

In a Queue data structure, we maintain two pointers, *front* and *rear*. The *front* points the first item of queue and *rear* points to last item.

enQueue() This operation adds a new node after *rear* and moves *rear* to the next node.

deQueue() This operation removes the front node and moves *front* to the next node.



In above example, the last inserted node is 50 and it is pointed by '**rear**' and the first inserted node is 10 and it is pointed by '**front**'. The order of elements inserted is 10, 15, 22 and 50.

enQueue() operation

```
void enQueue(int value)
{
   struct Node *newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = value;
   newNode -> next = NULL;

   if(front == NULL)
      front = rear = newNode;
   else
   {
      rear -> next = newNode;
      rear = newNode;
   }
}
```

deQueue() operation

```
int deQueue()
{
   struct Node *temp = front;
   int data = -1;

   if(front == NULL)
      printf("\nQueue is Empty!!!\n");
   else{
      front = front -> next;
      data = temp->data;
      free(temp);
   }
   return(data);
}
```