QUEUES

Queues: Introduction to queues: properties and operations, implementing queues using arrays and linked lists, circular queue and priority queue Applications of queues in breadth-first search, scheduling.
Deques: Introduction to deques (double-ended queues), Operations on deques and their applications.

INTRODUCTION TO QUEUES

Let us explain the concept of queues using the analogies given below.

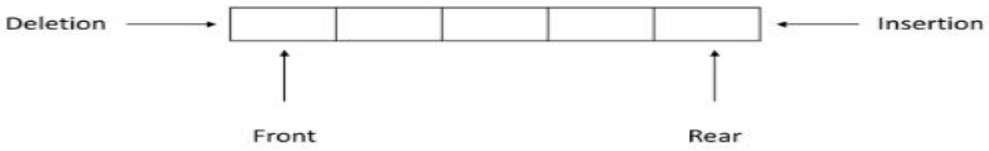
- ✓ People moving on an escalator. The people who got on the escalator first will be the first one to step out of it.
- ✓ People waiting for a bus. The first person standing in the line will be the first one to get into the bus.
- ✓ People standing outside the ticketing window of a cinema hall. The first person in the line will get the ticket first and thus will be the first one to move out of it.
- ✓ Cars lined at a toll bridge. The first car to reach the bridge will be the first to leave.

In all these examples, we see that the element at the first position is served first. Same is the case with queue data structure.
"Queue data structure is a collection of similar data items in which insertion and deletion operations are performed based on FIFO principle"

PROPERTIES AND OPERATIONS

The key property of the queue data structure is **First In, First Out (FIFO)**. This means that elements are processed in the **same order** they are added:

- ✓ **Insertion:** New elements are inserted at the **back** of the queue called as enqueue .
- ✓ **Deletion/Removal:** Elements are removed from the **front** of the queue called as dequeue.
- ✓ **Linear data structure:** Elements are arranged in a linear sequence.
- ✓ **Restricted access:** Items can only be accessed and removed from the front, while insertion happens at the back.
- ✓ **Dynamic size:** The size of the queue can grow or shrink as elements are added or removed.



ARRAY REPRESENTATION & IMPLEMENTATION OF QUEUES

Queues can be easily represented using linear arrays. As stated earlier, every queue has front and rear variables that point to the position from where deletions and insertions can be done, respectively.
The array representation of a queue is shown in Fig.

12	9	7	18	14	36				
0	1	2	3	4	5	6	7	8	9

Figure 1 Queue

12	9	7	18	14	36	45			
0	1	2	3	4	5	6	7	8	9

Figure 2 Queue after insertion of a new element

	9	7	18	14	36	45			
0	1	2	3	4	5	6	7	8	9

Figure 3 Queue after deletion of an element

Operations on Queues

In Fig.1, FRONT = 0 and REAR = 5. Suppose we want to add another element with value 45, then REAR would be incremented by 1 and the value would be stored at the position pointed by REAR.

The queue after addition would be as shown in Fig.2. Here, FRONT = 0 and REAR = 6. Every time a new element has to be added, we repeat the same procedure.

If we want to delete an element from the queue, then the value of FRONT will be incremented. Deletions are done from only this end of the queue. The queue after deletion will be as shown in Fig.3.Here, FRONT = 1 and REAR = 6.

However, before inserting an element in a queue, we must check for overflow conditions. An overflow will occur when we try to insert an element into a queue that is already full. When REAR = MAX – 1, where MAX is the size of the queue, we have an overflow condition. Note that we have written MAX – 1 because the index starts from 0. Similarly, before deleting an element from a queue, we must check for underflow conditions. An underflow condition occurs when we try to delete an element from a queue that is already empty. If FRONT = –1 and REAR = –1, it means there is no element in the queue.

```
Step 1: IF REAR = MAX-1
            Write OVERFLOW
            Goto step 4
        [END OF IF]
Step 2: IF FRONT = -1 and REAR = -1
            SET FRONT = REAR = 0
        ELSE
            SET REAR = REAR + 1
        [END OF IF]
Step 3: SET QUEUE[REAR] = NUM
Step 4: EXIT
```

Algorithm to insert an element in a queue

```
Step 1: IF FRONT = -1 OR FRONT > REAR
            Write UNDERFLOW
        ELSE
            SET VAL = QUEUE[FRONT]
            SET FRONT = FRONT + 1
        [END OF IF]
Step 2: EXIT
```

Algorithm to delete an element from a queue

NOTE: The process of inserting an element in the queue is called enqueue, and the process of deleting an element from the queue is called dequeue.

LINKED REPRESENTATION & IMPLEMENTATION OF QUEUES

We have seen how a queue is created using an array. Although this technique of creating a queue is easy, its drawback is that the array must be declared to have some fixed size.

And in another case we allocate less memory locations for a queue that might end up growing large and large, then a lot of re-allocations will have to be done, thereby creating a lot of overhead and consuming a lot of time. But if the array size cannot be determined in advance, the other alternative, i.e., the linked representation is used.

The storage requirement of linked representation of a queue with n elements is O(n) and the typical time requirement for operations is O(1).

In a linked queue, every element has two parts, one that stores the data and another that stores the address of the next element. The START pointer of the linked list is used as FRONT. Here, we will also use another pointer called REAR, which will store the address of the last element in the queue.

All insertions will be done at the rear end and all the deletions will be done at the front end. If FRONT = REAR = NULL, then it indicates that the queue is empty. The linked representation of a queue is shown in Fig. 5.

Figure 5 Linked queue

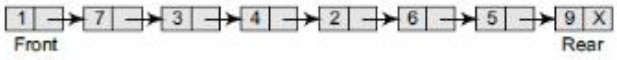


Figure 6 Linked queue after inserting a new node

```
Step 1: Allocate memory for the new node and name
        it as PTR
Step 2: SET PTR -> DATA = VAL
Step 3: IF FRONT = NULL
            SET FRONT = REAR = PTR
            SET FRONT -> NEXT = REAR -> NEXT = NULL
        ELSE
            SET REAR -> NEXT = PTR
            SET REAR = PTR
            SET REAR -> NEXT = NULL
        [END OF IF]
Step 4: END
```

Algorithm to insert an element in a linked queue

Insert Operation

The insert operation is used to insert an element into a queue. The new element is added as the last element of the queue. Consider the linked queue shown in Fig. 5.

To insert an element with value 9, we first check if FRONT=NULL. If the condition holds, then the queue is empty. So, we allocate memory for a new node, store the value in its data part and NULL in its next part. The new node will then be called both FRONT and rear. However, if FRONT != NULL, then we will insert the new node at the rear end of the linked queue and name this new node as rear. Thus, the updated queue becomes as shown in Fig.6.

The algorithm shows that inserting an element in a linked queue. In Step 1, the memory is allocated for the new node. In Step 2, the DATA part of the new node is initialized with the value to be stored in the node. In Step 3, we check if the new node is the first node of the linked queue. This is done by checking if FRONT = NULL. If this is the case, then the new node is tagged as FRONT as well as REAR. Also NULL is stored in the NEXT part of the node (which is also the FRONT and the REAR node). However, if the new node is not the first node in the list, then it is added at the REAR end of the linked queue (or the last node of the queue).

Delete Operation

The delete operation is used to delete the element that is first inserted in a queue, i.e., the element whose address is stored in FRONT. However, before deleting the value, we must first check if FRONT=NULL because if this is the case, then the queue is empty and no more deletions can be done. If an attempt is made to delete a value from a queue that is already empty, an underflow message is printed. Consider the queue shown in Fig. 7.

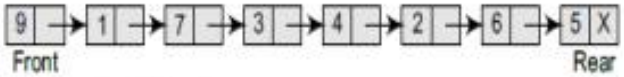


Figure 7 Linked queue

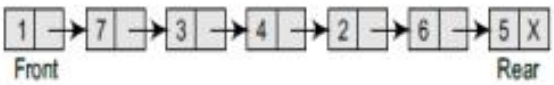


Figure 8 Linked queue after deletion of an element

```
Step 1: IF FRONT = NULL
            Write "Underflow"
            Go to Step 5
        [END OF IF]
Step 2: SET PTR = FRONT
Step 3: SET FRONT = FRONT -> NEXT
Step 4: FREE PTR
Step 5: END
```

APPLICATIONS OF QUEUES

- ✓ Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
- ✓ Queues are used to transfer data asynchronously (data not necessarily received at same rate as sent) between two processes (IO buffers), e.g., pipes, file IO, sockets.
- ✓ Queues are used as buffers on MP3 players and portable CD players, iPod playlist.
- ✓ Queues are used in Playlist for jukebox to add songs to the end, play from the front of the list.
- ✓ Queues are used in operating system for handling interrupts. When programming a real-time system that can be interrupted, for example, by a mouse click, it is necessary to process the interrupts immediately, before proceeding with the current job. If the interrupts have to be handled in the order of arrival, then a FIFO queue is the appropriate data structure.

TYPES OF QUEUES

A queue data structure can be classified into the following types:
1. Circular Queue 2.Priority Queue 3. Deque

Circular Queues

In linear queues, we have discussed so far that insertions can be done only at one end called the REAR and deletions are always done from the other end called the FRONT. Look at the queue shown in Fig.9.

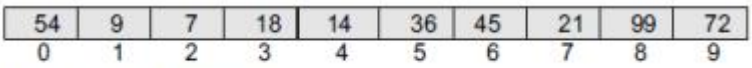


Figure 9 Linear queue

Here, FRONT = 0 and REAR = 9.

Now, if you want to insert another value, it will not be possible because the queue is completely full. There is no empty space where the value can be inserted. Consider a scenario in which two successive deletions are made. The queue will then be given as shown in Fig. 10.

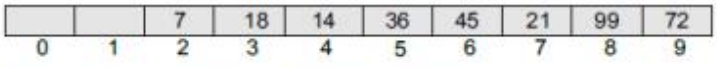


Figure 10 Queue after two successive deletions

Here, front = 2 and REAR = 9.

Suppose we want to insert a new element in the queue shown in Fig.10. Even though there is space available, the overflow condition still exists because the condition rear = MAX – 1 still holds true. This is a major drawback of a linear queue.

To resolve this problem, we have two solutions. First, shift the elements to the left so that the vacant space can be occupied and utilized efficiently. But this can be very time-consuming, especially when the queue is quite large.

The second option is to use a circular queue. In the circular queue, the first index comes right after the last index. Conceptually, you can think of a circular queue as shown in Fig. 11.

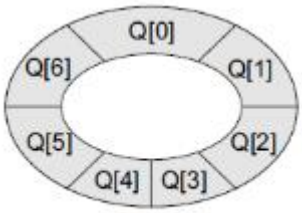


Figure 11 Circular queue

The circular queue will be full only when front = 0 and rear = Max – 1. A circular queue is implemented in the same manner as a linear queue is implemented. The only difference will be in the code that performs insertion and deletion operations.

For insertion, we now have to check for the following three conditions:

- ✓ If front = 0 and rear = MAX – 1, then the circular queue is full. Look at the queue given in Fig. 12 which illustrates this point.
- ✓ If rear != MAX – 1, then rear will be incremented and the value will be inserted as illustrated in Fig.13.
- ✓ If front != 0 and rear = MAX – 1, then it means that the queue is not full. So, set rear = 0 and insert the new element there, as shown in Fig.14.

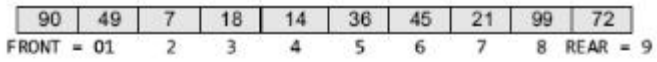


Figure 12 Full queue

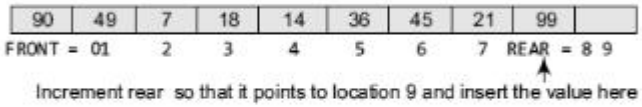


Figure 13 Queue with vacant locations

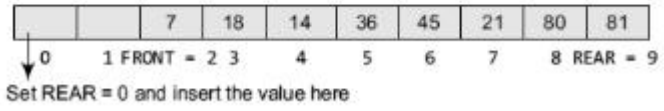


Figure 14 Inserting an element in a circular queue

```
Step 1: IF FRONT = 0 and Rear = MAX - 1
            Write "OVERFLOW"
            Goto step 4
        [End OF IF]
Step 2: IF FRONT = -1 and REAR = -1
            SET FRONT = REAR = 0
        ELSE IF REAR = MAX - 1 and FRONT != 0
            SET REAR = 0
        ELSE
            SET REAR = REAR + 1
        [END OF IF]
Step 3: SET QUEUE[REAR] = VAL
Step 4: EXIT
```

Let us look at the algorithm to insert an element in a circular queue. In Step 1, we check for the overflow condition. In Step 2, we make two checks. First to see if the queue is empty, and second to see if the REAR end has already reached the maximum capacity while there are certain free locations before the FRONT end. In Step 3, the value is stored in the queue at the location pointed by REAR.

Let us now discuss how **deletions** are performed in this case. To delete an element, again we check for three conditions.

- ✓ Look at Fig.15. If front = –1, then there are no elements in the queue. So, an underflow condition will be reported.
- ✓ If the queue is not empty and front = rear, then after deleting the element at the front the queue becomes empty and so front and rear are set to –1. This is illustrated in Fig. 16.
- ✓ If the queue is not empty and front = MAX–1, then after deleting the element at the front, front is set to 0. This is shown in Fig. 17.

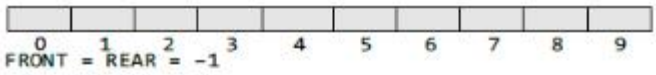


Figure 15 Empty queue

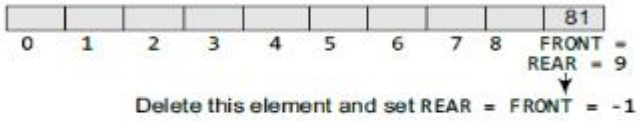


Figure 16 Queue with a single element

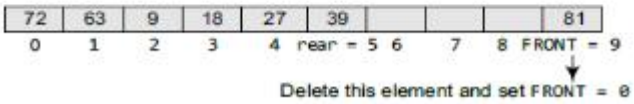


Figure 17 Queue where FRONT = MAX-1 before deletion

```
Step 1: IF FRONT = -1
            Write "UNDERFLOW"
            Goto Step 4
        [END of IF]
Step 2: SET VAL = QUEUE[FRONT]
Step 3: IF FRONT = REAR
            SET FRONT = REAR = -1
        ELSE
            IF FRONT = MAX -1
                SET FRONT = 0
            ELSE
                SET FRONT = FRONT + 1
            [END of IF]
        [END OF IF]
Step 4: EXIT
```

Let us look at the algorithm to delete an element from a circular queue. In Step 1, we check for the underflow condition. In Step 2, the value of the queue at the location pointed by FRONT is stored in VAL. In Step 3, we make two checks. First to see if the queue has become empty after deletion and second to see if FRONT has reached the maximum capacity of the queue. The value of FRONT is then updated based on the outcome of these checks.

Priority Queues

A priority queue is a data structure in which each element is assigned a priority. The priority of the element will be used to determine the order in which the elements will be processed. The general rules of processing the elements of a priority queue are

- ✓ An element with higher priority is processed before an element with a lower priority.
- ✓ Two elements with the same priority are processed on a first-come-first-served (FCFS) basis.

A priority queue can be thought of as a modified queue in which when an element has to be removed from the queue, the one with the highest-priority is retrieved first. The priority of the element can be set based on various factors. Priority queues are widely used in operating systems to execute the highest priority process first. The priority of the process may be set based on the CPU time it requires to get executed completely.

Implementation of a Priority Queue

There are two ways to implement a priority queue. We can either use a sorted list to store the elements so that when an element has to be taken out, the queue will not have to be searched for the element with the highest priority or we can use an unsorted list so that insertions are always done at the end of the list.

Every time when an element has to be removed from the list, the element with the highest priority will be searched and removed. While a sorted list takes O(n) time to insert an element in the list, it takes only O(1) time to delete an element. On the contrary, an unsorted list will take O(1) time to insert an element and O(n) time to delete an element from the list.

Practically, both these techniques are inefficient and usually a blend of these two approaches is adopted that takes roughly O(log n) time or less.

APPLICATIONS OF QUEUES IN BREADTH-FIRST SEARCH:

Breadth-first search is a graph traversal algorithm that starts traversing the graph from the root node and explores all the neighboring nodes. Then, it selects the nearest node and explores all the unexplored nodes. While using BFS for traversal, any node in the graph can be considered as the root node.

Algorithm

Step 1: Create an empty queue to store the nodes to be visited and a boolean vector to track the visited nodes. Initially, no node is visited.

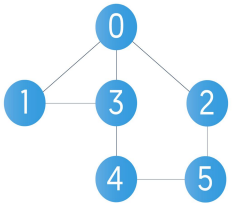
Step 2: Choose a starting node, mark it visited, and add it to the queue.

Step 3: Extract the front item of the queue, add it to the BFS order, and traverse through the list of that node's adjacent nodes.

Step 4: If a node in that list has not been already visited, mark it visited, and add it to the back of the queue.

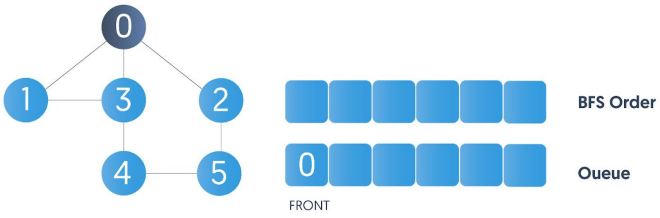
Step 5: Keep repeating steps 3 and 4 until the queue is empty.

Let's take an example. We'll use an undirected graph with six vertices.



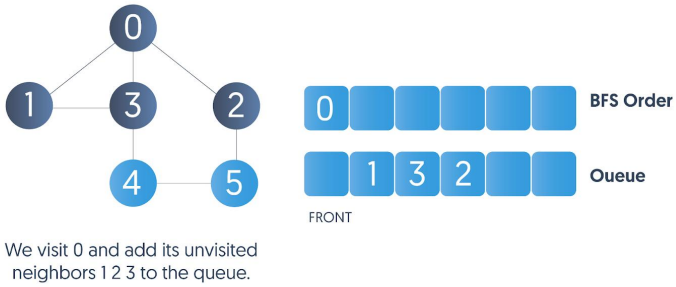
Graph with six nodes

We first choose a node to start BFS and put it in the queue after marking it "visited." In this case, we have chosen 0.



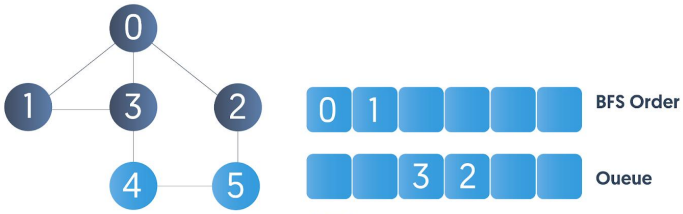
The queue only has the starting node in the beginning.

We will repeatedly pop the front element of the queue, traverse its neighboring nodes, and add it to "BFS Order." Currently, we have 0 at the front of the queue, and none of its neighbors are visited. So, we'll mark them as visited and add all of them to the queue.



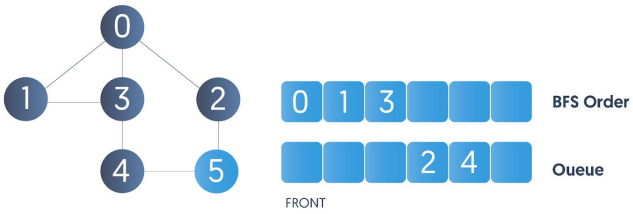
We visit 0 and add its unvisited neighbors 1 2 3 to the queue.

Now, we have node 1 at the front of the queue. It has only two neighbors, 0 and 3, which have been marked visited in a previous step. So, we'll move on to the next step.



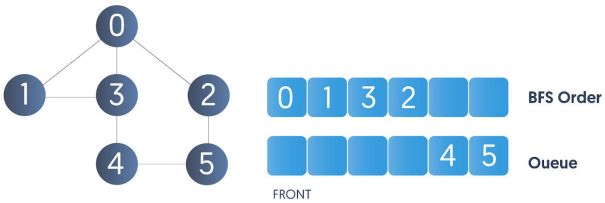
We visit 1 and but it doesn't have any unvisited neighbor

Here, we have 3 at the front of the queue. It has only one unvisited neighbor, 4. So, we will mark it visited and add it to the queue.



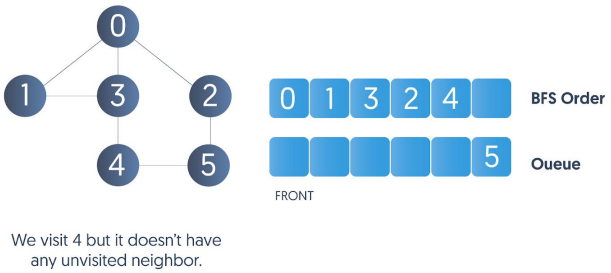
We visit 3 and add its only unvisited neighbour 4 to the queue.

For node 2, we add its unvisited neighbor 5 to the queue.



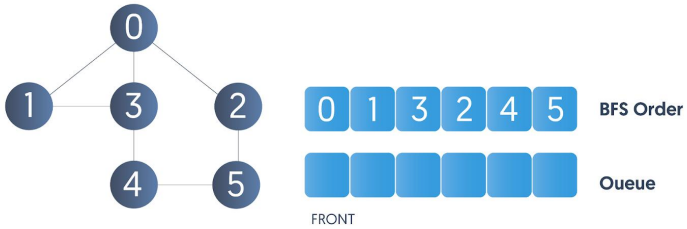
We visit 2 and add its only neighbour 5 to the queue

For node 4, all its neighbors are already visited — we move on to the next step.



We visit 4 but it doesn't have any unvisited neighbor.

After extracting 5 from the queue, all the graph elements have been visited, and there are no elements left in the queue. BFS on this graph is complete.



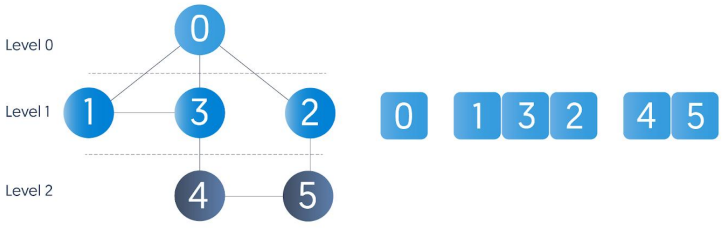
We visit 5 and now all nodes have been visited and the queue is empty.

If you look closely, you will notice that we have visited the nodes layerwise, that is, in the order of their level. Let's understand what that means.

A level structure of an undirected graph is a partition of the vertices into subsets that are at the same distance from a given root vertex. Here:

- On level 0, we have only node 0, the root vertex
- On level 1, we have 1, 3, and 2
- On the last level, we have 4 and 5

Now, if you observe the obtained BFS Order, we have visited the nodes in the order of their levels.



The nodes have been visited in order to their level.

SCHEDULING:

Queues can be used to schedule jobs and ensure that they are executed in the correct order. Scheduling of processes/work is done to finish the work on time. **CPU Scheduling** is a process that allows one process to use the CPU while another process is delayed (in standby) due to unavailability of any resources such as I / O etc, thus making full use of the CPU. The purpose of CPU Scheduling is to make the system more efficient, faster, and fairer.

There are mainly two types of scheduling methods:

- ✓ **Preemptive Scheduling:** Preemptive scheduling is used when a process switches from running state to ready state or from the waiting state to the ready state.
- ✓ **Non-Preemptive Scheduling:** Non-Preemptive scheduling is used when a process terminates , or when a process switches from running state to waiting state.

First Come First Serve:

FCFS considered to be the simplest of all operating system scheduling algorithms. First come first serve scheduling algorithm states that the process that requests the CPU first is allocated the CPU first and is implemented by using FIFO queue.

Characteristics of FCFS:

- FCFS supports non-preemptive and preemptive CPU scheduling algorithms.
- Tasks are always executed on a First-come, First-serve concept.
- FCFS is easy to implement and use.
- This algorithm is not much efficient in performance, and the wait time is quite high.

Given n processes with their burst times, the task is to find average waiting time and average turn around time using FCFS scheduling algorithm.
Here we are considering that arrival time for all processes is 0.

How to compute below times in Round Robin using a program?

FCFS (Example)

Process	Duration	Oder	Arrival Time
P1	24	1	0
P2	3	2	0
P3	4	3	0

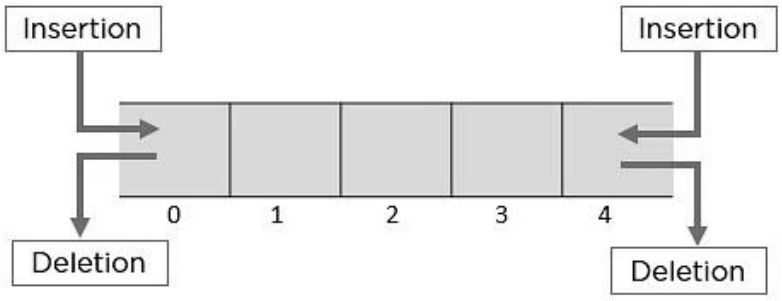
Gantt Chart :

P1 waiting time : 0
P2 waiting time : 24
P3 waiting time : 27

The Average waiting time :
(0+24+27)/3 = 17

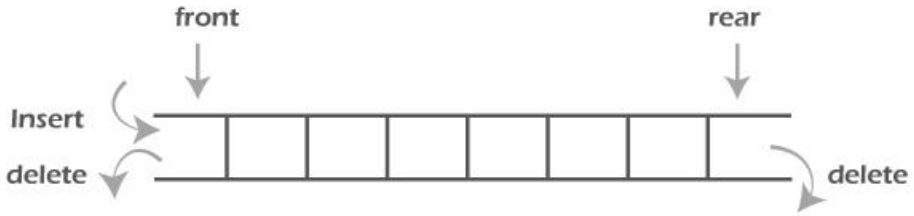
DEQUES

A deque (or double-ended queue) is a list in which the elements can be inserted or deleted at either end. It is also known as a head-tail linked list because elements can be added to or removed from either the front (head) or the back (tail) end.



There are two variants of a double-ended queue. They include

- ✓ Input restricted deque In this dequeue, insertions can be done only at one of the ends, while deletions can be done from both ends.



input restricted double ended queue

- ✓ Output restricted deque In this dequeue, deletions can be done only at one of the ends, while insertions can be done on both ends.

Output restricted double ended queue

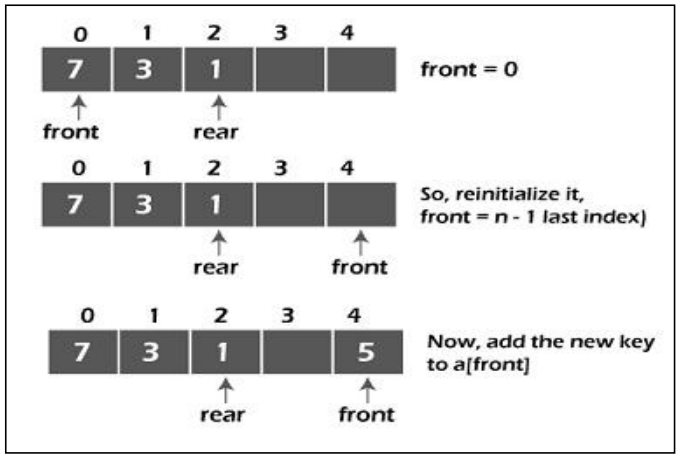
Operations on Deque

Four basic operations are performed on deque, they are as follows:

- Insertion at Rear
- Insertion at Front
- Deletion at Front
- Deletion at Rear

Insertion at the front end

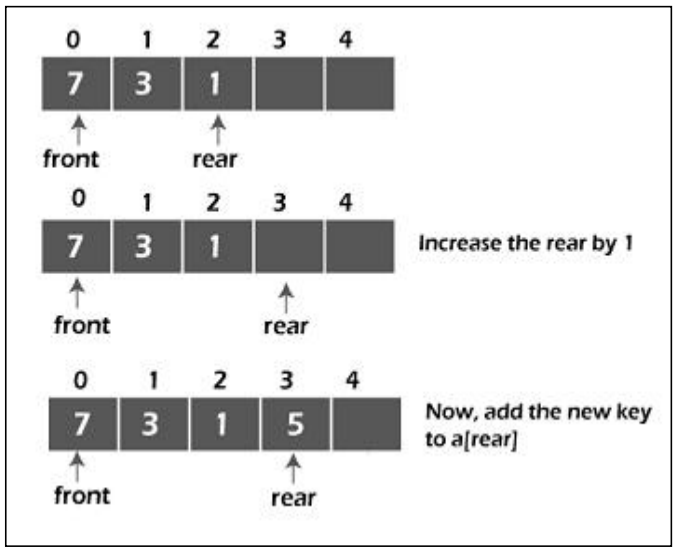
In this operation, the element is inserted from the front end of the queue. Before implementing the operation, we first have to check whether the queue is full or not. If the queue is not full, then the element can be inserted from the front end by using the below conditions -
If the queue is empty, both rear and front are initialized with 0. Now, both will point to the first element.
Otherwise, check the position of the front if the front is less than 1 (front < 1), then reinitialize it by front = n - 1, i.e., the last index of the array.



Insertion at the rear end

In this operation, the element is inserted from the rear end of the queue. Before implementing the operation, we first have to check again whether the queue is full or not. If the queue is not full, then the element can be inserted from the rear end by using the below conditions -

- If the queue is empty, both rear and front are initialized with 0. Now, both will point to the first element.
- Otherwise, increment the rear by 1. If the rear is at last index (or size - 1), then instead of increasing it by 1, we have to make it equal to 0.



Deletion at the front end

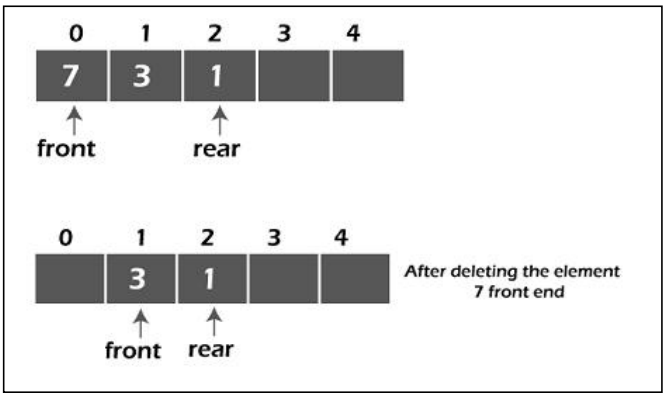
In this operation, the element is deleted from the front end of the queue. Before implementing the operation, we first have to check whether the queue is empty or not.

If the queue is empty, i.e., front = -1, it is the underflow condition, and we cannot perform the deletion. If the queue is not full, then the element can be inserted from the front end by using the below conditions -

if the deque has only one element, set rear = -1 and front = -1.

Else if front is at end (that means front = size - 1), set front = 0.

Else increment the front by 1, (i.e., front = front + 1).



Deletion at the rear end

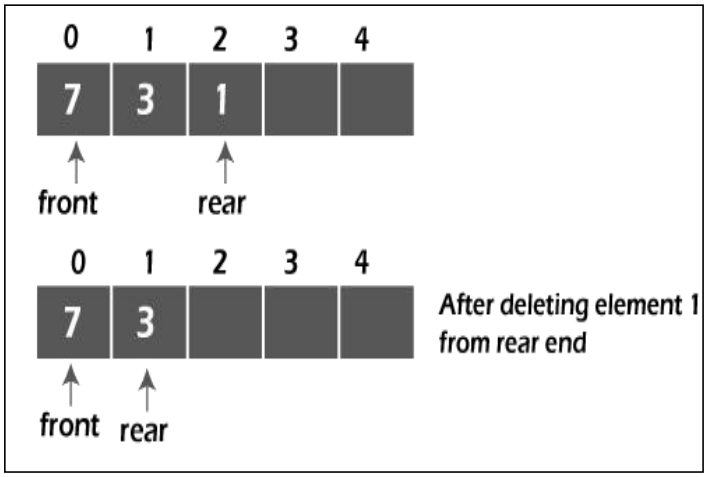
In this operation, the element is deleted from the rear end of the queue. Before implementing the operation, we first have to check whether the queue is empty or not.

If the queue is empty, i.e., front = -1, it is the underflow condition, and we cannot perform the deletion.

If the deque has only one element, set rear = -1 and front = -1.

If rear = 0 (rear is at front), then set rear = n - 1.

Else, decrement the rear by 1 (or, rear = rear -1).



Check empty

This operation is performed to check whether the deque is empty or not. If front = -1, it means that the deque is empty.

Check full

This operation is performed to check whether the deque is full or not. If front = rear + 1, or front = 0 and rear = n - 1 it means that the deque is full.

The time complexity of all of the above operations of the deque is O(1), i.e., constant.

Applications of Deque in Data Structure

The deque can act as both the queue and stack. Thus, all the applications of queue and stacks can be implemented using a double-ended queue. Given below are some critical applications of deque in data structure:

Palindrome Checker

The string or a number that reads the same backward as forward is known as a palindrome. The program to check if the given string is palindrome or not can be implemented using deque in a data structure. If the string reads the same from both the deque ends, it will consider it a palindrome. The image given below is an illustration of a palindrome checker implementation using a double-ended queue.

Multiprocessor Scheduling

When multiple processes are being executed by multiple processors (CPU, Core) in a system, then that system utilizes a multiprocessor scheduling algorithm. In this algorithm, the deque is used by each processor to store different threads of processes. This algorithm is also called the A-Steal algorithm for scheduling. The illustration given below represents the process of multiprocessor scheduling.