note+slide

Lecture 17-21

- Queues

  First-In-First-Out (FIFO)

IT205: Data Structures (AY 2023/24 Sem II Sec B) — **Dr. Arpit Rana**

In general it is homogenous but we also make this hetrogenous for storing a data of hetreogenous type
in c we use union for storing hetreonenous data.

Definition

A Queue is an

- **ordered collection** of elemens are indexded
- **homogeneous** data elements
- where the insertion (a.k.a. **enqueue**) and deletion (a.k.a. **dequeue**) operations take place **at two ends** called the **rear and the front** of the queue.

The maximum number of elements that a queue can accommodate is termed as **LENGTH** of the queue. It operates in First-In-First-Out (FIFO) fashion.

Definition

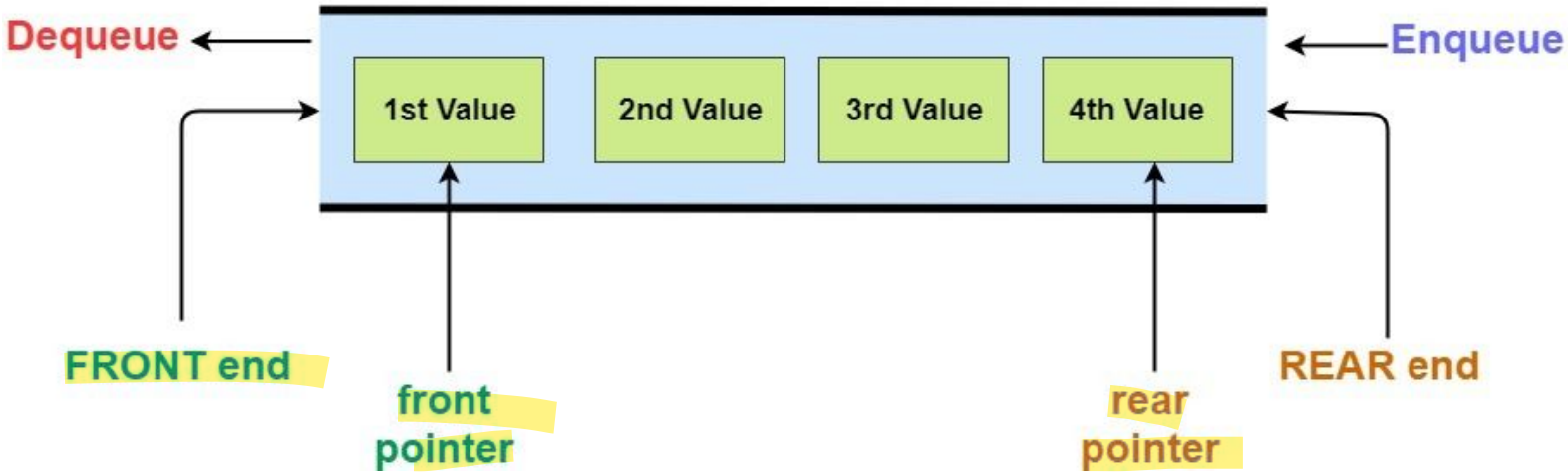


Image Source: https://simplesnippets.tech/

Representation of a Queue

Most commonly, a Queue can be represented in the memory in two ways:

- Using a one-dimensional array
  (better choice when a fixed length queue is required)
- Using a linked list
  (a single/double lists whose size can vary during processing)

deletion is done from front and insertion is done form rare.

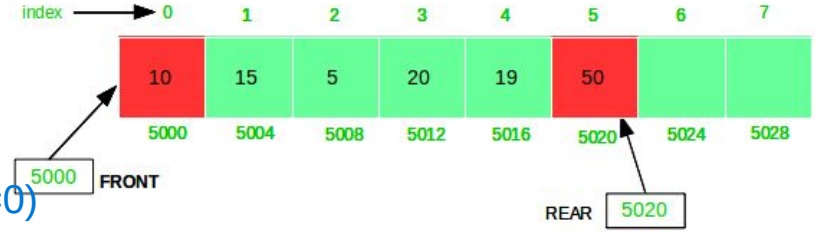
insertion is called enqueue and deletion is called dequeue.

Array Representation of a Queue

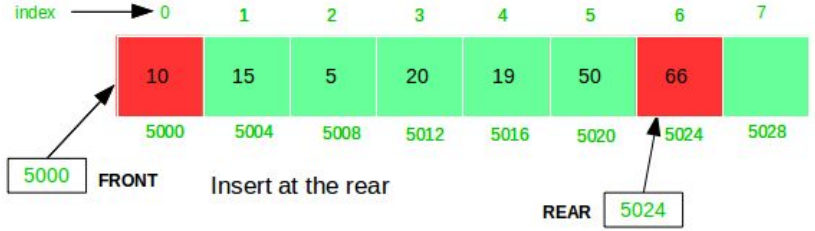
use idea write in notebook

First, a block of sufficient size (1…N) to accommodate the full capacity of the queue is allocated.

in note book (lowerindex=0)

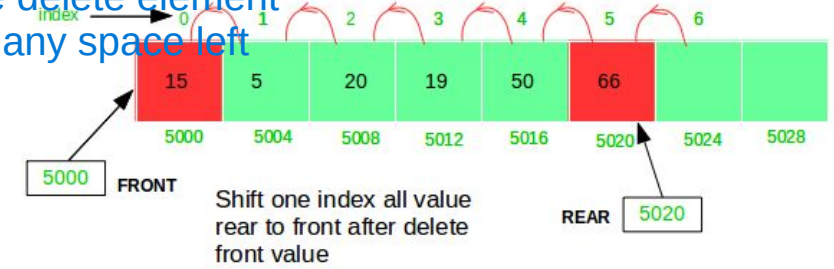


- Queue is **empty**: front==rear;
  FRONT = 0
  REAR = 0 (or -1)



- Queue is **full**: rear=size-1
  FRONT = 1
  REAR = N

here front=1 because when we delete element front move further so behind many space left so it is not full in that condition



- Queue contains **elements >= 1**:
  FRONT <= REAR
  Number of elements: REAR - FRONT + 1

code->queue1.cpp

Image Source: https://www.geeksforgeeks.org/

Operations on Queues

The basic operations required to manipulate a queue are:

- $enqueue(Q, ITEM)$: to insert an item in the queue
- $dequeue(Q)$: to remove an item from the queue
- $empty(Q)$: to know whether the queue is empty

Operations on a Queue

Input: A queue “Q” and a new element ITEM to be inserted in it
Output: A queue “Q” after inserting an element **at the rear of the queue**

Steps to perform *enqueue*(Q, *ITEM*):

```
If REAR == N then                          // Q is full
    Print “Queue is full”
    exit()
Else
    If REAR == 0 AND FRONT == 0 then       // Q is empty
        FRONT = 1
    EndIf
    REAR = REAR + 1
    Q[REAR] = ITEM
EndIf
Stop
```

Operations on a Queue

Input: A Queue with elements
Output: Removes an item from the front of the queue if it is not empty

Steps to perform dequeue(Q)

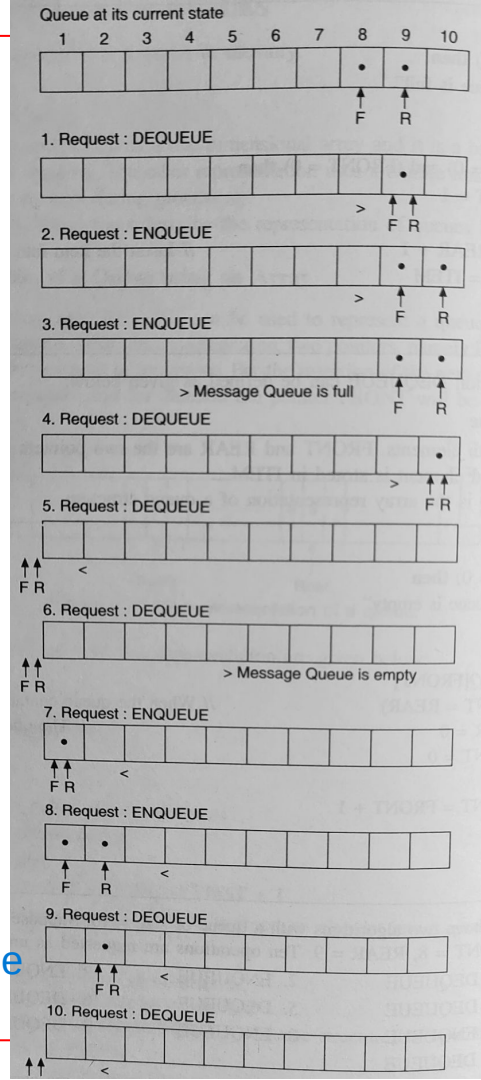
```
If FRONT == 0 then
    Print "Queue is empty"
    exit()
Else
    ITEM = Q[FRONT]
    If FRONT == REAR then    // Q has only one element
        REAR = 0
        FRONT = 0
    Else
        FRONT = FRONT + 1
    EndIf
EndIf
Stop
```

Exercise

Let's trace the above two algorithms with a queue of size = 10 where FRONT=8 and REAR=9. The operations are requested as below.

- dequeue → enqueue → enqueue → dequeue → dequeue → dequeue → enqueue → enqueue → dequeue → dequeue.

- There is one potential problem with this representation.
  - What is that problem?
  - How can we resolve it?



here problem is in 3 rd case because queue is not full but is showing it is full. so solution is we can make one pointer stuck at one position and we move only second pointer but in this we have one problem that it take O(n) because when we delete element then we should also shift all element by one so it take O(n) time .code->queue1ad.cpp

Linked Representation of a Queue

in this we don't store any data in head

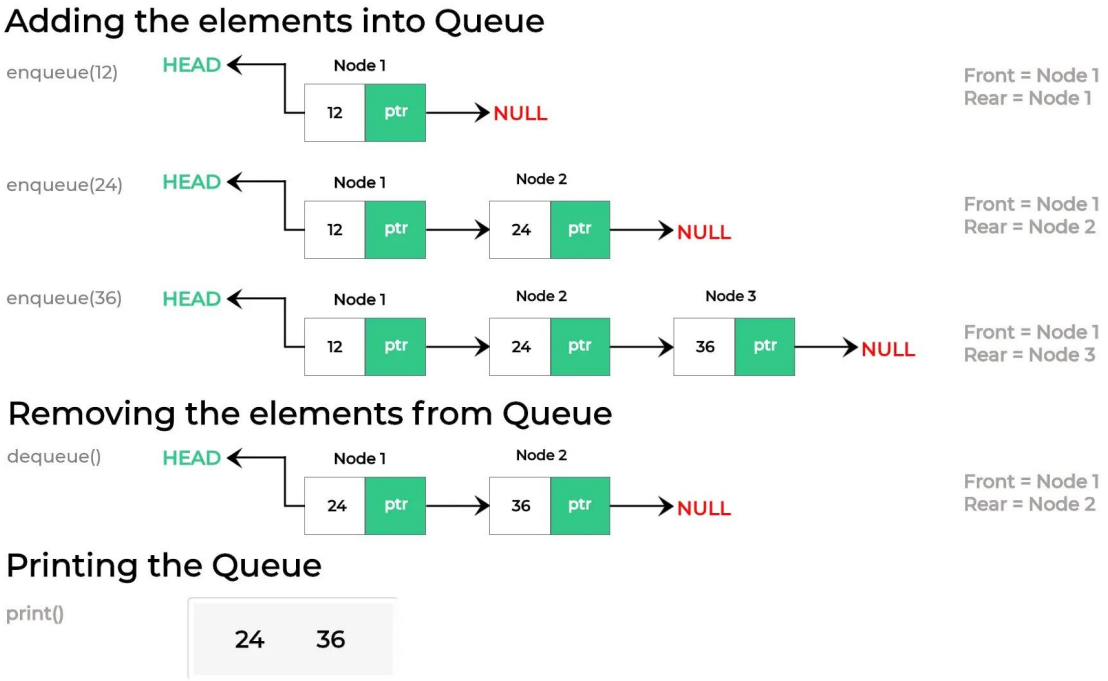
Array representation of queue is easy and convenient but **allows only fixed sized queue**. The solution is linked representation.

- The pointers FRONT and REAR point the first and the last node in the list.
- Queue is **empty**:

  FRONT = REAR = HEAD
  HEAD = NULL
- Queue contains at least one element:

  HEAD != NULL

queue4 and 3.cpp



Exercise

code5.cpp

- Implement a queue using doubly linked lists, outline the advantages and/or disadvantages of it.

Disadvantages of Using Doubly Linked List for Queue:

Increased Memory Overhead:
Doubly linked lists require more memory per node due to the storage of both next and previous pointers. In a queue, where simple First-In-First-Out (FIFO) behavior is often sufficient, the extra memory for backward traversal might be unnecessary.

Complexity and Overhead:
Implementing and maintaining a doubly linked list adds complexity to the code. The additional pointers and operations to keep the list doubly linked may introduce overhead, especially when compared to a simpler data structure like a singly linked list.

Potential for Bugs:
The bidirectional nature of doubly linked lists introduces additional opportunities for errors in pointer manipulation. If not managed carefully, it could lead to bugs such as dangling pointers or inconsistent state in the data structure.

Overkill for Basic Queuing:
For a basic queue with FIFO behavior, a doubly linked list might be overkill. Using a simpler data structure like a singly linked list or even an array could provide the necessary functionality without the added complexity and memory overhead.

Various Queue Structures

- Circular queue
- Deque
- Priority queue

for circular Queue:

for index starting with 0

starting index ->front =rear=-1

enqueue:front=(rear+1)%size ==>Queue is full

dequeue:front=rear=-1 ==>Queue is empty

for first element insertion and last element deletion condition are different

code=>queue6.cpp

circular queue is also called as link buffer.In circular queue insertion and deletion take constant time. also in circular queue memmroy does not waste.

Circular Queue

here consider starting index=>1

Physically, a circular array is the same as an ordinary array, say Q[1… N], but logically it implies that Q[1] comes after Q[N].

- Both pointers will move in clockwise direction.
- This is controlled by **MOD** operation:
  - p points to the current location (1 ≤ p ≤ LENGTH)
  - next location = (p **MOD** LENGTH) + 1
- Circular Queue is **empty**:
  FRONT = 0; REAR = 0
- Circular Queue is **full**:
  FRONT = (REAR **MOD** LENGTH) + 1

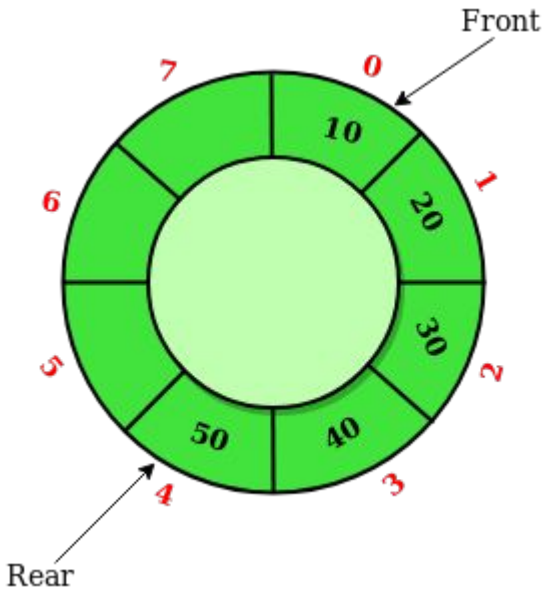


Image Source: https://www.geeksforgeeks.org/

Operations on a Circular Queue

The following image describes the insertion and deletion of elements from a circular queue.

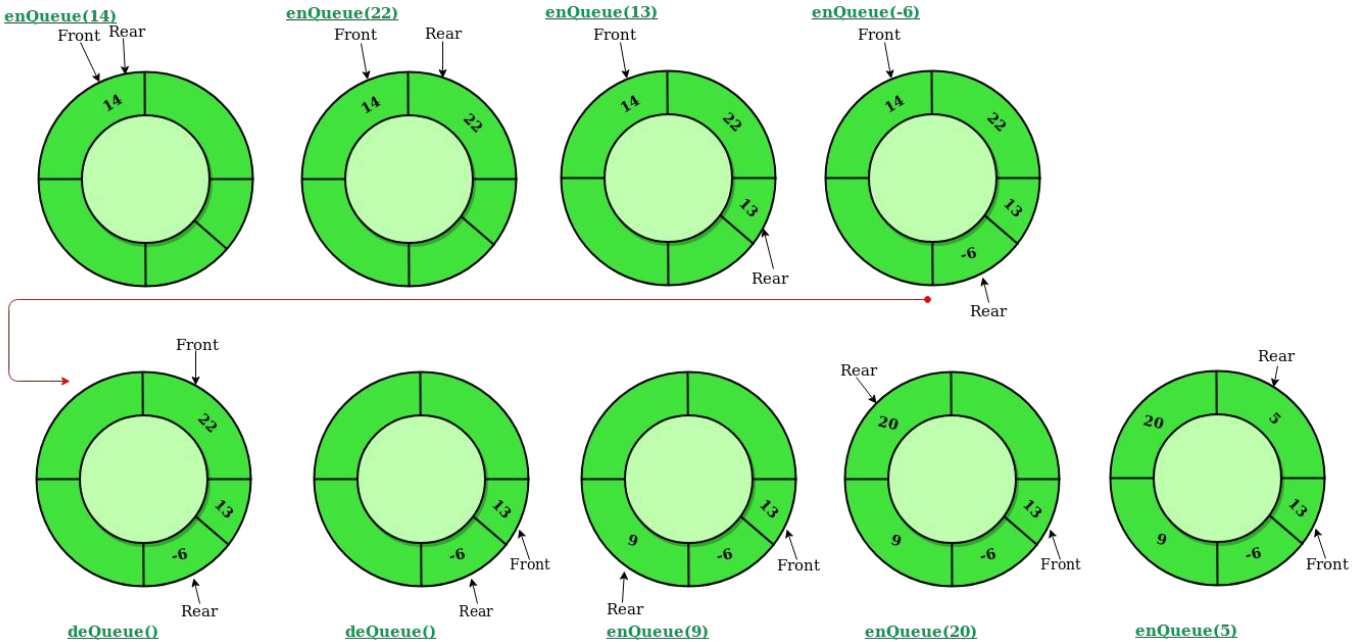


Image Source: https://www.geeksforgeeks.org/

Operations on a Circular Queue

Input: A circular queue “CQ” and a new element ITEM to be inserted in it
Output: A circular queue “CQ” after inserting an element **at the rear of the queue**

Steps to perform *enqueue*(CQ, ITEM):

```
If FRONT == 0 then                          // CQ is empty
    FRONT = REAR = 1
    CQ[FRONT] = ITEM
Else
    next = (REAR MOD LENGTH) + 1
    If next != FRONT then                   // If CQ is not full
        REAR = next
        CQ[REAR] = ITEM
    Else
        Print “Queue is full”
    EndIf
EndIf
Stop
```

Operations on a Circular Queue

Input: A Circular Queue “CQ” with elements
Output: Removes an item from the front of the queue if it is not empty

Steps to perform dequeue(CQ)

```
If FRONT == 0 then
    Print “Queue is empty”
    exit()
Else
    ITEM = CQ[FRONT]
    If FRONT == REAR then    // Q has only one element
        REAR = 0
        FRONT = 0
    Else
        FRONT = (FRONT MOD LENGTH) + 1
    EndIf
EndIf
Stop
```

Various Queue Structures

- Circular queue
- **Deque**
- Priority queue

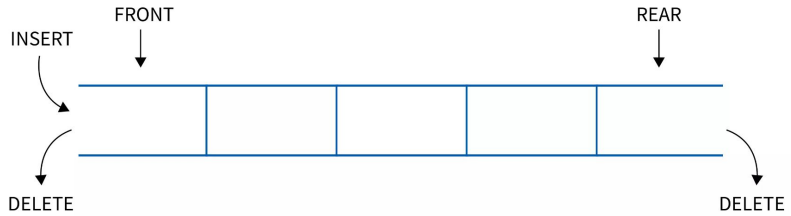
Double Ended Queue (a.k.a. Deque)

Unlike a queue, in deque, both insertion and deletion operations can be performed at either end of the structure.

A Deque Structure

Input-restricted Deque

Allows insertion at one end (e.g., at FRONT) only



Output-restricted Deque

Allows deletion at one end (e.g., at REAR) only

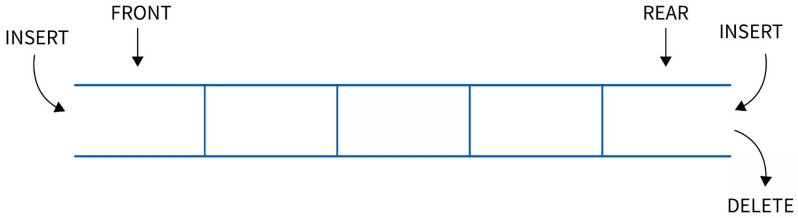


Image Source: https://www.scaler.com/topics/dequeue-in-data-structure//

Double Ended Queue (a.k.a. Deque)

A deque can be represented as a stack as well as a queue. It can be represented by using a doubly linked list or as a circular array (as used in a circular queue).

The following four operations are possible on a deque which consists of a list of items.

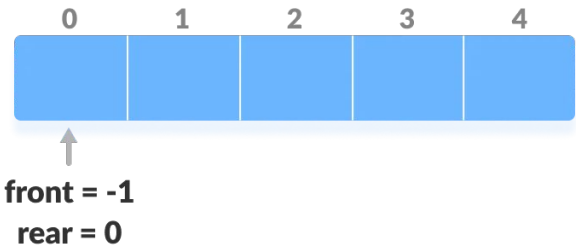
- **Push_deque(ITEM)**: to insert ITEM at the FRONT of a deque
- **Pop_deque()**: to remove the ITEM from the FRONT of a deque | ***same as dequeue(CQ)***
- **Inject(ITEM)**: to insert ITEM at the REAR of a deque | ***same as enqueue(CQ, ITEM)***
- **Eject()**: to remove the ITEM from the REAR of a deque

Assuming a deque is implemented using a circular array of length LENGTH. Let the array be DQ[1, … , LENGTH].

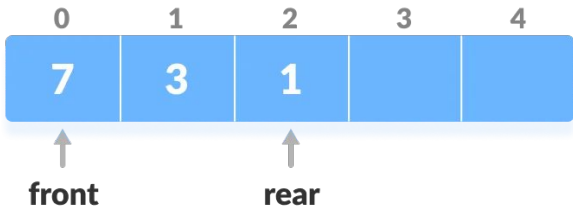
code->queue7.cpp this code written with use of circular queue.
with use of doubly linked list baki.

Operations on a Deque: Push at Front

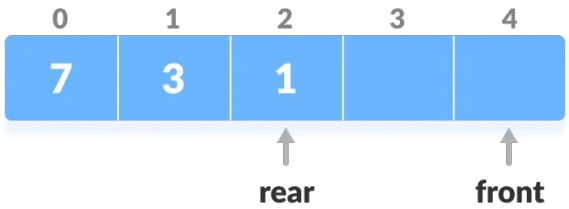
- Initialize an array and pointers for deque



- Check the position of front



- Shift front to the end if front < 1



- Else, decrease front by 1. Finally, insert the element

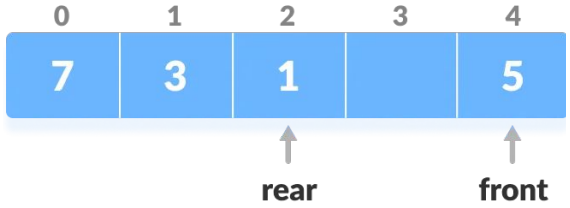


Image Source: https://www.programiz.com/dsa/deque

Operations on a Deque: Push at Front

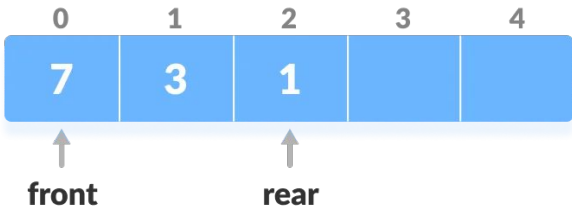
Input: A deque "DQ" and a new element ITEM to be inserted at the FRONT of it
Output: A deque "DQ" after inserting an element **at the FRONT of the queue**

Steps to perform Push_deque(*ITEM*):

```
If FRONT == 1 then                          // FRONT is at extreme left
    next = LENGTH
Else                                        // FRONT is at extreme right or the deque is empty
    If (FRONT == LENGTH) or (FRONT == 0)then
        next = 1
    Else                                    // FRONT is at an intermediate position
        next = FRONT - 1
    EndIf
    If next == REAR then
        Print "Deque is full"; exit()
    Else                                    // Push the ITEM
        FRONT = next; DQ[FRONT] = ITEM
    EndIf
EndIf
Stop
```

Operations on a Deque: Eject from Rear

- Check if deque is empty



- If the deque is empty (i.e. front = -1), deletion cannot be performed (underflow condition).

- If the deque has only one element (i.e. front = rear), set front = -1 and rear = -1, else follow the steps below.

- If rear is at the front (i.e. rear = 0), set it to rear = n - 1.

- Else, rear = rear - 1.

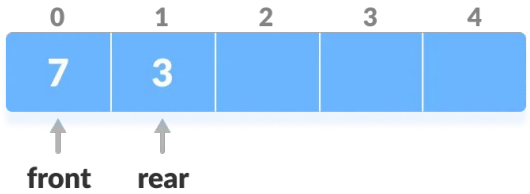


Image Source: https://www.programiz.com/dsa/deque

Operations on a Deque: Eject from Rear

Input: A Deque “DQ” with elements
Output: Removes an item from the REAR of the queue if it is not empty

Steps to perform Eject()

```
If FRONT == 0 then
    Print “Queue is empty”; exit()
Else
    If FRONT == REAR then          //Deque contains single element
        ITEM = DQ[REAR]
        FRONT = REAR = 0
    Else
        If REAR == 1 then              // REAR is at extreme left
            ITEM = DQ[REAR]
            REAR = LENGTH
        Else
            If REAR == LENGTH then     // REAR is at extreme right
                ITEM = DQ[REAR]
                REAR = 1
```

Operations on a Deque

contd…

```
                Else                // REAR is at an intermediate position
                    ITEM = DQ[REAR]
                    REAR = REAR - 1
                EndIf
            EndIf
        EndIf
    EndIf
    Stop
```

Various Queue Structures

- Circular queue
- Deque
- **Priority queue**

A priority queue is a data structure that stores a collection of elements, each associated with a priority. The basic idea is that elements with higher priorities are dequeued before elements with lower priorities. In other words, when you dequeue an element from a priority queue, you get the element with the highest priority.

Priority queues are commonly used in various algorithms and applications where you need to process elements in order of their priority. They are essential in tasks such as task scheduling, graph algorithms (like Dijkstra's algorithm for shortest paths), and Huffman coding in compression algorithms.

There are different implementations of priority queues, such as binary heaps, Fibonacci heaps, and binomial heaps. The choice of implementation depends on the specific requirements and performance characteristics needed for a particular application.

Here are two main operations supported by a priority queue:

Insertion: Adds an element to the priority queue with a specified priority.

Deletion (or Extraction): Removes and returns the element with the highest priority.

The efficiency of these operations depends on the underlying data structure used to implement the priority queue. In general, priority queues are designed to provide efficient access to the element with the highest priority, making them valuable in a wide range of computer science and algorithmic applications.

Priority Queue

A priority queue is a special type of queue in which–

- each element is associated with a **priority value**,
- elements are served on the basis of their priority, i.e., **higher priority elements are served first**.
- if elements with the same priority occur, they are served according to their order in the queue.

Queues implements FIFO rule whereas, in a priority queue, the values are removed on the basis of priority. The element with the highest priority is removed first.

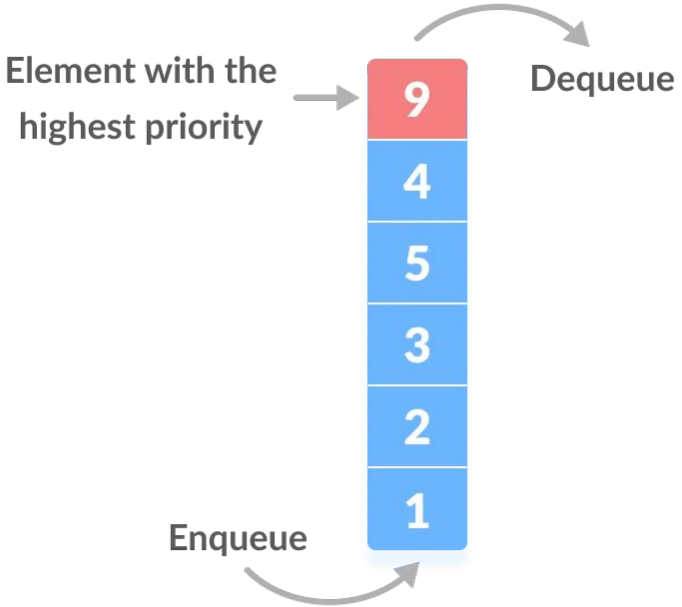
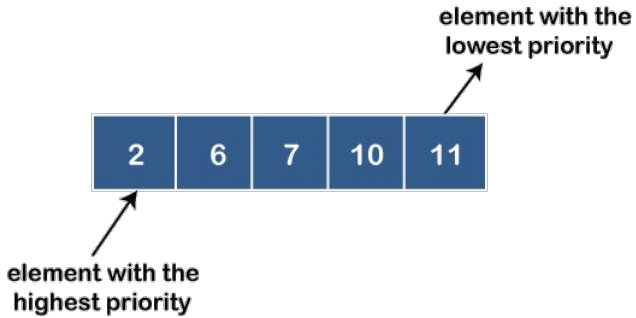


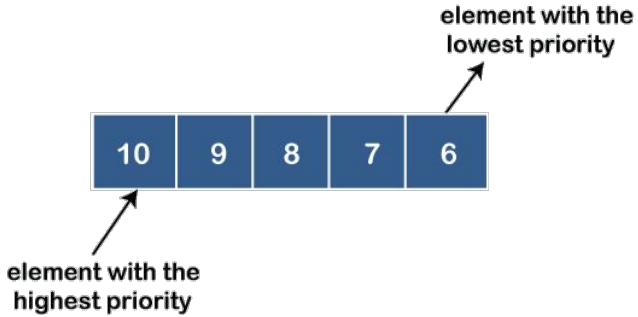
Image Source: https://www.programiz.com/dsa/priority-queue

Types of Priority Queue

- Ascending Order Priority Queue: a lower priority number is given as a higher priority in a priority.



- Descending order Priority Queue: a higher priority number is given as a higher priority in a priority.



Priority Queue

Implementation of Priority Queue:

Priority queue can be implemented using

- an array (simple/circular),
- a linked list,
- a heap data structure, or
- a binary search tree.

we first write code for Priority queue with use of linked list

Among these data structures, heap data structure provides an efficient implementation of priority queues.

Priority Queue using an Array

An array is maintained to hold the item and its priority value. The element is inserted at the REAR end as usual and the deletion is performed in either of the following ways.

- Search the element of the highest priority, delete the element, and shift the trailing elements to fill up the vacant position. OR
- Add the element, sort the queue to keep the highest priority element at the front, and delete it from the front.

operation	argument	return value	size	contents (unordered)	contents (ordered)
insert	P		1	P	P
insert	Q		2	P Q	P Q
insert	E		3	P Q E	E P Q
remove max		Q	2	P E	E P
insert	X		3	P E X	E P X
insert	A		4	P E X A	A E P X
insert	M		5	P E X A M	A E M P X
remove max		X	4	P E M A	A E M P
insert	P		5	P E M A P	A E M P P
insert	L		6	P E M A P L	A E L M P
insert	E		7	P E M A P L E	A E E L M
remove max		P	6	E E M A P L	A E E L M

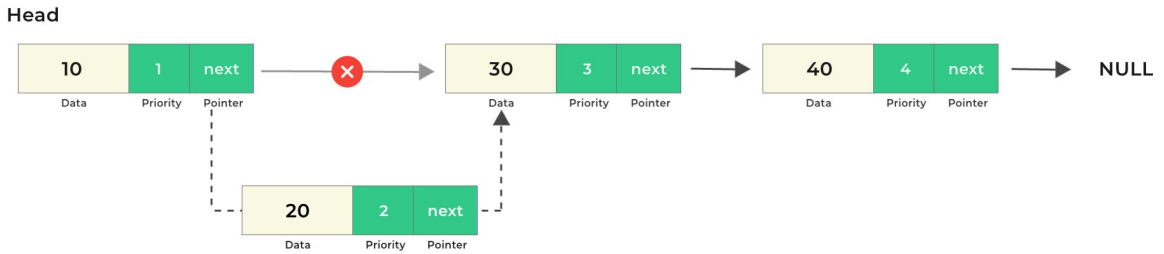
A sequence of operations on a priority queue

Priority Queue using a Linked List

Implementation of priority Queue data by using a linked list is more efficient as compared to arrays. Here all the nodes are in sorted order of their priority.

Queue8.cpp

Insertion



Deletion

Applications of Queue

- Used in the Dijkstra's shortest path algorithm
- CPU Scheduling in a Multiprogramming Environment
- Used in the operating system for load balancing and interrupt handling
- Used as a buffer for input devices like the Keyboard.
- Widely used for spooling in Printers.

when you have less time and more work that time you use priority . so application in which time is less but work is more then that place priority queue used

Next Lecture

- Tables