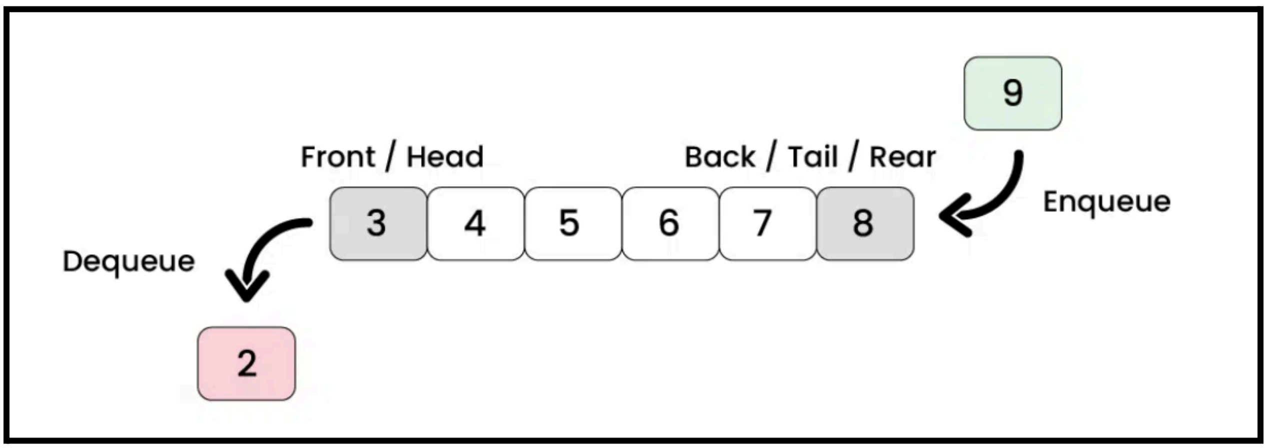
UNIT-IV : QUEUES

Introduction to queues :

A **queue** is a linear data structure where elements are stored in the FIFO (First In First Out) principle where the first element inserted would be the first element to be accessed. A queue is an Abstract Data Type (ADT) similar to stack, the thing that makes queue different from stack is that a queue is open at both its ends. The data is inserted into the queue through one end and deleted from it using the other end.

Representation of queue :



Operations of queues :

These are all built-in operations to carry out data manipulation and to check the status of the queue.

- Enqueue: Insert an element at the end of the queue.
- Dequeue: Simply remove an element from the front of the queue.
- IsEmpty: Used to check if the queue is completely empty.
- IsFull: Used to check if the queue is completely full.
- Peek: Without removing it, obtain the value from the front of the queue.

Queues can be implemented by using arrays and linked list.

Enqueue Operation in Queue:
Enqueue() operation in Queue **adds (or stores) an element to the end of the queue**.Addition of an element to the queue. Adding an element will be performed after checking whether the queue is full or not. If rear < n which indicates that the array is not full then store the element at arr[rear] and increment rear by 1 but if rear == n then it is said to be an Overflow condition as the array is full.
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, return overflow error and exit.
- **Step 3:** If the queue is not full, increment the rear pointer to point to the next empty space.
- **Step 4:** Add the data element to the queue location, where the rear is pointing.
- **Step 5:** return success.

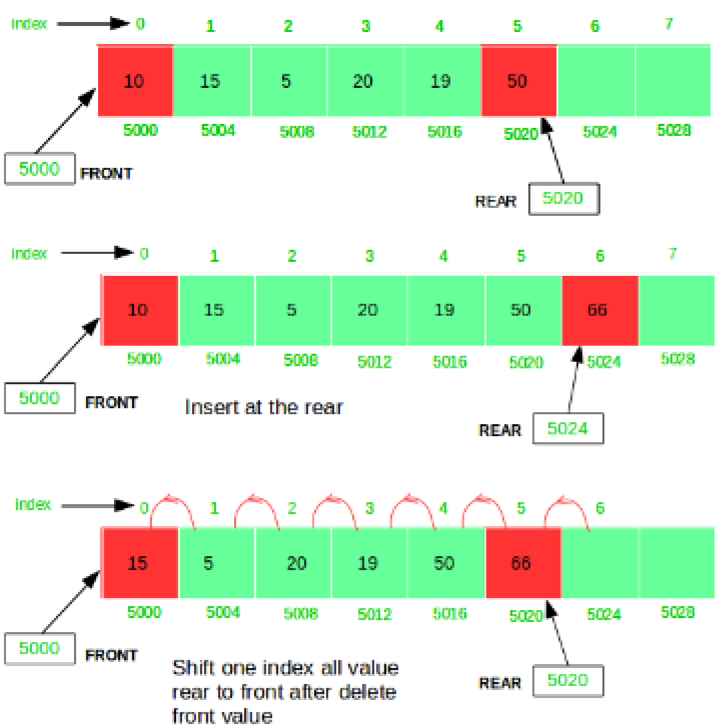
Algorithm

- Step 1: IF REAR = MAX - 1
  Write OVERFLOW
  Go to step [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

Dequeue Operation in Queue:
Removes (or access) the first element from the queue.

Removal of an element from the queue. An element can only be deleted when there is at least an element to delete i.e. rear > 0. Now, the element at arr[front] can be deleted but all the remaining elements have to shift to the left by one position in order for the dequeue operation to delete the second element from the left on another dequeue operation.
The following steps are taken to perform the dequeue operation:

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



Algorithm

- IF(FRONT==-1||FRONT==REAR+1)
- RETURN
- ELSE
- QUEUE[FRONT]=0
- FRONT=FRONT+1

Implementation of queue using arrays

Advantages of implementing queues using arrays:

- Memory Efficient: Array elements do not hold the next elements address like linked list nodes do.
- Easier to implement and understand: Using arrays to implement queues require less code than using linked lists, and for this reason it is typically easier to understand as well

Implementing queues using arrays

```
#include<stdio.h>

#include<stdlib.h>

#define maxsize 5

void insert();

void delete();

void display();

int front = -1, rear = -1;
```

```
int queue[maxsize];
void main ()
{
    int choice;
    while(choice != 4)
    {
        printf("\n1.insert an element\n2.Delete an element\n3.Display the queue\n4.Exit\n");
        printf("\nEnter your choice ?");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
            insert();
            break;
            case 2:
            delete();
            break;
            case 3:
            display();
            break;
            case 4:
            exit(0);
            break;
            default:
            printf("\nEnter valid choice??\n");
        }
```

```
    }
}

void insert()
{
    int item;
    printf("\nEnter the element\n");
    scanf("\n%d",&item);
    if(rear == maxsize-1)
    {
        printf("\nOVERFLOW\n");
        return;
    }
    if(front == -1 && rear == -1)
    {
        front = 0;
        rear = 0;
    }
    else
    {
        rear = rear+1;
    }
    queue[rear] = item;
    printf("\nValue inserted ");

}
```

```
void delete()
{
    int item;
    if (front == -1 || front > rear)
    {
        printf("\nUNDERFLOW\n");
        return;
    }
    else
    {
        item = queue[front];  //copy the element first before overwritten
        if(front == rear)
        {
            front = -1;
            rear = -1 ;
        }
        else
        {
            front = front + 1;
        }
        printf("\nvalue deleted ");
    }
}
```

```
void display()
{
    int i;
    if(rear == -1)
    {
        printf("\nEmpty queue\n");
    }
    else
    {   printf("\nprinting values .....\n");
        for(i=front;i<=rear;i++)
        {
            printf("\n%d\n",queue[i]);
        }
    }
}
```

Disadvantages of implementing queues using linked list:

Fixed size: An array occupies a fixed part of the memory. This means that it could take up more memory than needed, or if the array fills up, it cannot hold more elements. And resizing an array can be costly.

- Shifting cost: Dequeue causes the first element in a queue to be removed, and the other elements must be shifted to take the removed elements' place. This is inefficient and can cause problems, especially if the queue is long.
- Alternatives: Some programming languages have built-in

Queues using linked list

There are two basic operations which can be implemented on the linked queues. The operations are Insertion and Deletion.

Process

1. First allocate the memory to a new node pointer.
2. Inserting an element has two scenarios
   2.1 the condition **front = NULL** becomes true. Now, the new element will be added as the only element of the queue and the next pointer of front and rear pointer both, will point to NULL.

```
                 +----------+
front ------->   |  Node 1  |   rear -----> NULL
                 +----------+
```

   2.2 The condition front = NULL becomes false. In this scenario, we need to update the end pointer rear so that the next pointer of rear will point to the new node ptr. Since, this is a linked queue, hence we also need to make the rear pointer point to the newly added node **ptr**. We also need to make the next pointer of the rear point to NULL.

```
                +----------+       +----------+
front ------->  |  Node 1  | --->  |  Node 2  |   rear -----> NULL
                +----------+       +----------+
```

Algorithm

- **Step 1:** Allocate the space for the new node 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

```
#include<stdio.h>

#include<stdlib.h>

struct node

{

    int data;

    struct node *next;

};

struct node *front;

struct node *rear;

void insert();

void delete();

void display();

void main ()

{

    int choice;

    while(choice != 4)

    {           printf("\n1.insert an element\n2.Delete an element\n3.Display the queue\n4.Exit\n");

        printf("\nEnter your choice ?");

        scanf("%d",& choice);

        switch(choice)

        {

            case 1:

            insert();

            break;

            case 2:

            delete();
```

```
            break;
            case 3:
            display();
            break;
            case 4:
            exit(0);
            break;
            default:
            printf("\nEnter valid choice??\n");
        }
    }
}
void insert()
{
    struct node *ptr;
    int item;
    ptr = (struct node *) malloc (sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW\n");
        return;
    }
    else
    {
        printf("\nEnter value?\n");
        scanf("%d",&item);
```

```
        ptr -> data = item;
        if(front == NULL)
        {
            front = ptr;
            rear = ptr;
            front -> next = NULL;
            rear -> next = NULL;
        }
        else
        {
            rear -> next = ptr;
            rear = ptr;
            rear->next = NULL;
        }
    }
}
void delete ()
{
    struct node *ptr;
    if(front == NULL)
    {
        printf("\nUNDERFLOW\n");
        return;
    }
    else
    {
```

```
        ptr = front;
        front = front -> next;
        free(ptr);
    }
}
void display()
{
    struct node *ptr;
    ptr = front;
    if(front == NULL)
    {
        printf("\nEmpty queue\n");
    }
    else
    {   printf("\nprinting values .....\n");
        while(ptr != NULL)
        {
            printf("\n%d\n",ptr -> data);
            ptr = ptr -> next;
        }
    }
}
```

Applications of queues in breadth-first search

Breadth-first search (BFS) is a strategy for searching in a graph when search is limited to essentially two operations: (a) visit and inspect a node of a graph; (b) gain access to visit the nodes that neighbour the currently inspected node. The BFS begins at a root node and inspects all the neighbouring nodes. Then for each of those neighbour nodes in turn, it inspects their neighbour nodes which are unvisited, and so on. This is where a queue comes in handy.

A queue is a data structure that follows the First-In-First-Out (FIFO) principle. In the context of BFS, we initialise a queue with a starting node. Then we enter a loop where we dequeue a node from the front of the queue, visit that node, and enqueue all its unvisited neighbours at the back of the queue. This process continues until the queue is empty, which means all nodes reachable from the starting node have been visited.
all nodes at distance d from the start node are visited before any nodes at distance d+1. This is because when we visit a node and enqueue its neighbours, those neighbours are at a greater distance from the start node than the current node.

Algorithm

The steps involved in the BFS algorithm to explore a graph are given as follows -

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Enqueue the starting node A and set its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until QUEUE is empty

Step 4: Dequeue a node N. Process it and set its STATUS = 3 (processed state).

Step 5: Enqueue all the neighbours of N that are in the ready state (whose STATUS = 1) and set

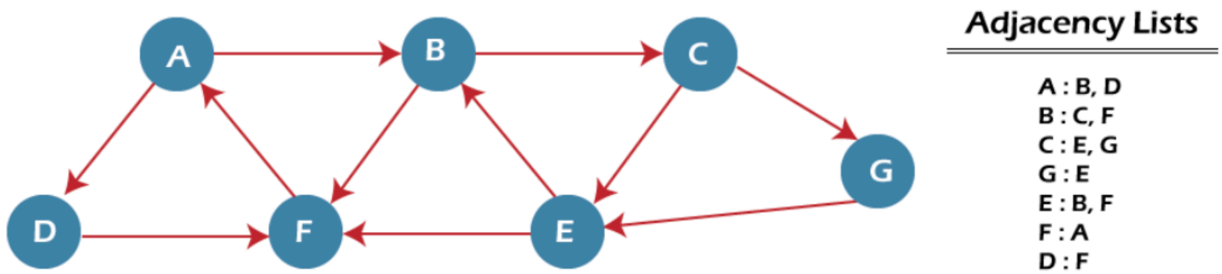
their STATUS = 2

(waiting state)

[END OF LOOP]

Step 6: EXIT

Working of BFS algorithm



In the above graph, minimum path 'P' can be found by using the BFS that will start from Node A and end at Node E. The algorithm uses two queues, namely QUEUE1 and QUEUE2. QUEUE1 holds all the nodes that are to be processed, while QUEUE2 holds all the nodes that are processed and deleted from QUEUE1.

The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.

Applications of queues in Scheduling

Process scheduling is an essential part of a Multiprogramming operating system. Such operating systems allow more than one process to be loaded into the executable memory at a time and loaded process shares the CPU using time multiplexing.

Scheduling Queues

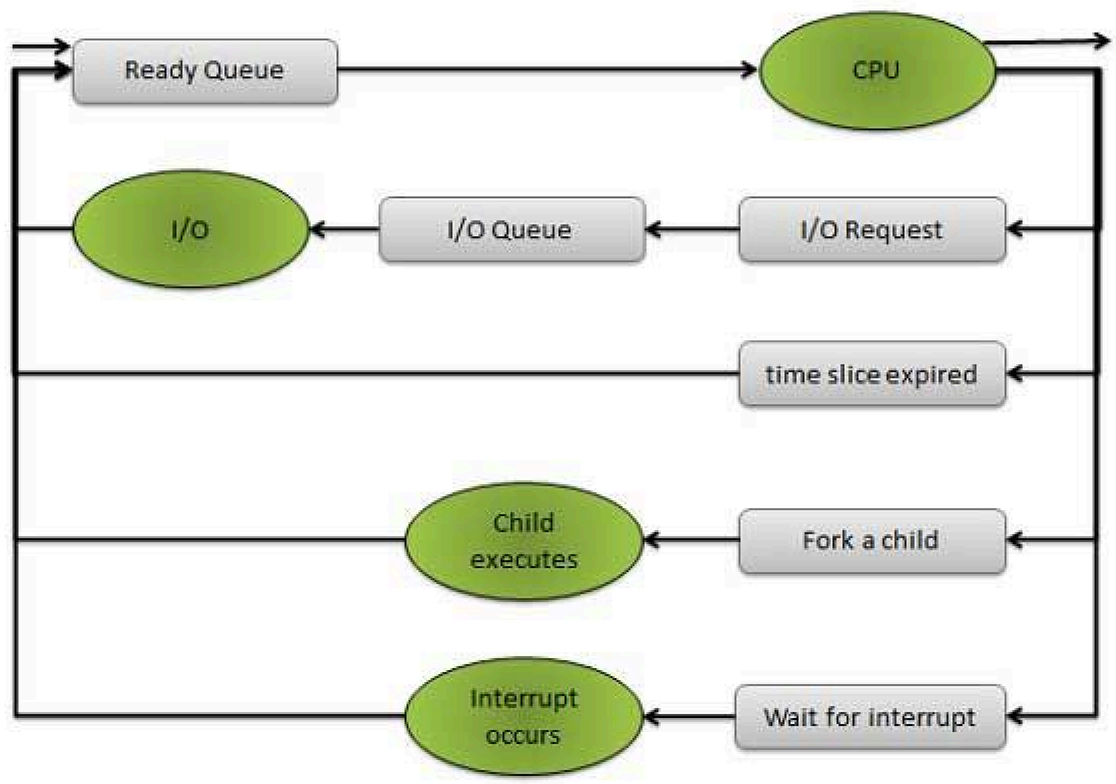
Scheduling queues refers to queues of processes or devices. When the process enters into the system, then this process is put into a job queue. This queue consists of all processes in the system. The operating system also maintains other queues such as device queue. Device queue is a queue for which multiple processes are waiting for a particular I/O device. Each device has its own device queue.

This figure shows the queuing diagram of process scheduling.

Queue is represented by rectangular box.

The circles represent the resources that serve the queues.

The arrows indicate the process flow in the system.



Queues are of two types

Ready queue
Device queue

A newly arrived process is put in the ready queue. Processes waits in ready queue for allocating the CPU. Once the CPU is assigned to a process, then that process will execute. While executing the process, any one of the following events can occur.
The process could issue an I/O request and then it would be placed in an I/O queue.
The process could create new sub process and will wait for its termination.
The process could be removed forcibly from the CPU, as a result of interrupt and put back in the ready queue.

Schedulers

Schedulers are special system softwares which handles process scheduling in various ways.Their main task is to select the jobs to be submitted into the system and to decide which process to run. Schedulers are of three types

Long Term Scheduler
Short Term Scheduler
Medium Term Scheduler

Long Term Scheduler

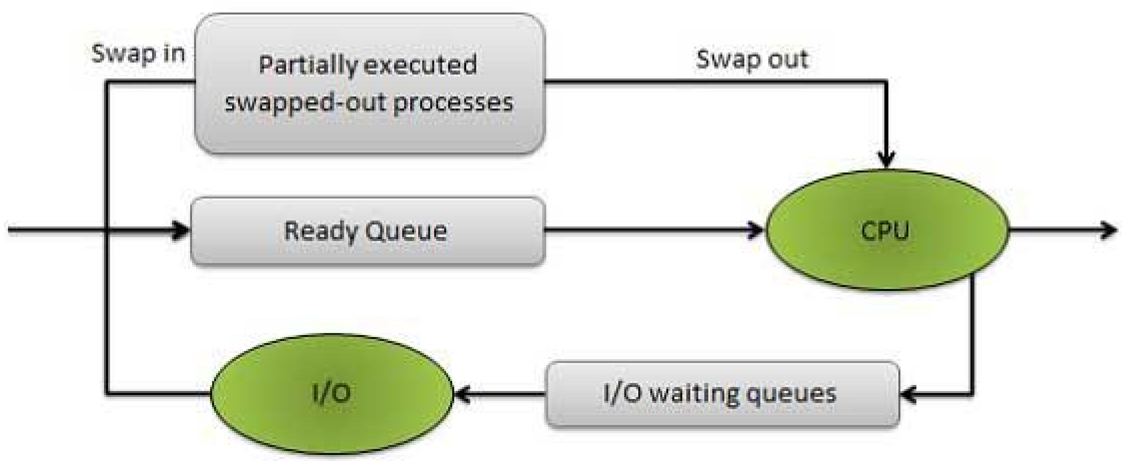
It is also called job scheduler. Long term scheduler determines which programs are admitted to the system for processing. Job scheduler selects processes from the queue and loads them into memory for execution. Process loads into the memory for CPU scheduling. The primary objective of the job scheduler is to provide a balanced mix of jobs, such as I/O bound and processor bound. It also controls the degree of multiprogramming. If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system.
On some systems, the long term scheduler may not be available or minimal. Time-sharing operating systems have no long term scheduler. When process changes the state from new to ready, then there is use of long term scheduler.

Short Term Scheduler

It is also called CPU scheduler. Main objective is increasing system performance in accordance with the chosen set of criteria. It is the change of ready state to running state of the process. CPU scheduler selects process among the processes that are ready to execute and allocates CPU to one of them.
Short term scheduler also known as dispatcher, execute most frequently and makes the fine grained decision of which process to execute next. Short term scheduler is faster than long term scheduler.

Medium Term Scheduler

Medium term scheduling is part of the swapping. It removes the processes from the memory. It reduces the degree of multiprogramming. The medium term scheduler is in-charge of handling the swapped out-processes.



Running process may become suspended if it makes an I/O request. Suspended processes cannot make any progress towards completion. In this condition, to remove the process from memory and make space for other process, the suspended process is moved to the secondary storage. This process is called swapping, and the process is said to be swapped out or rolled out. Swapping may be necessary to improve the process mix.

Deque

Deque is a type of queue in which insert and deletion can be performed from either **front** or **rear**. It does not follow the FIFO rule. It is also known as **double-ended queue**

Representation of deque

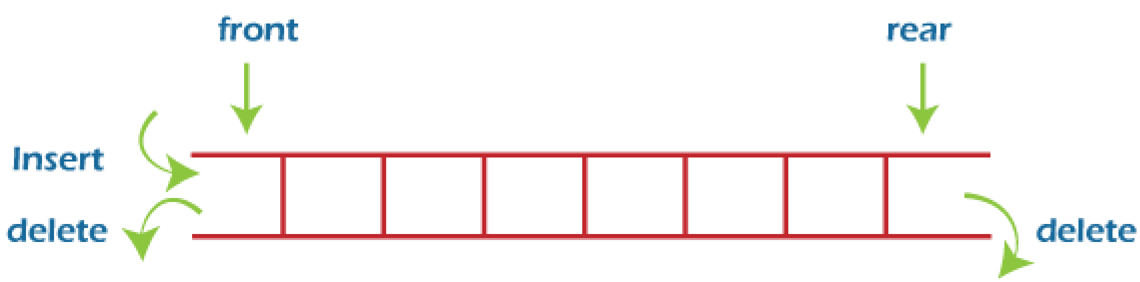
Types of deque

There are two types of deque -

- Input restricted queue
- Output restricted queue

Input restricted Queue

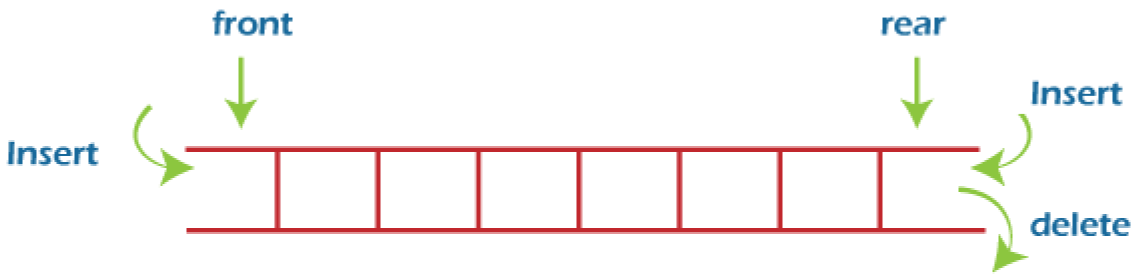
In input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.



input restricted double ended queue

Output restricted Queue

In output restricted queue, deletion operation can be performed at only one end, while insertion can be performed from both ends.



Output restricted double ended queue

Operations on Deque:

Deque consists of mainly the following operations:

- Insert Front
- Insert Rear
- Delete Front
- Delete Rear

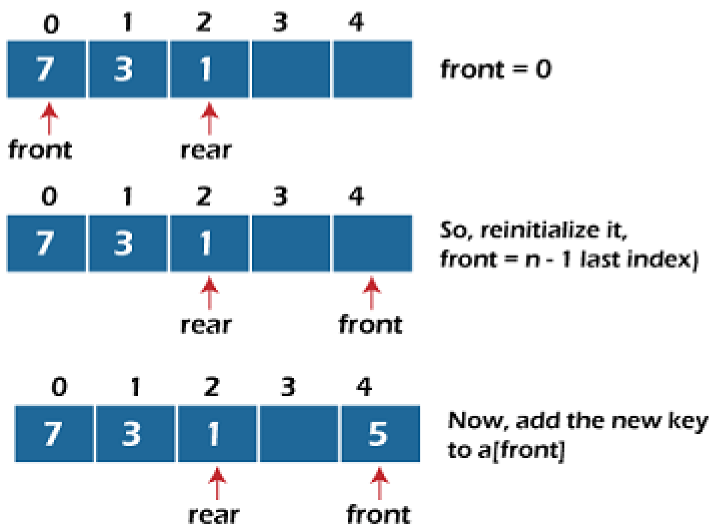
in addition to the above operations, following operations are also supported in deque -

- Get the front item from the deque
- Get the rear item from the deque
- Check whether the deque is full or not
- Checks whether the deque is empty or not

1. Insert at the Front: TIn 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 -

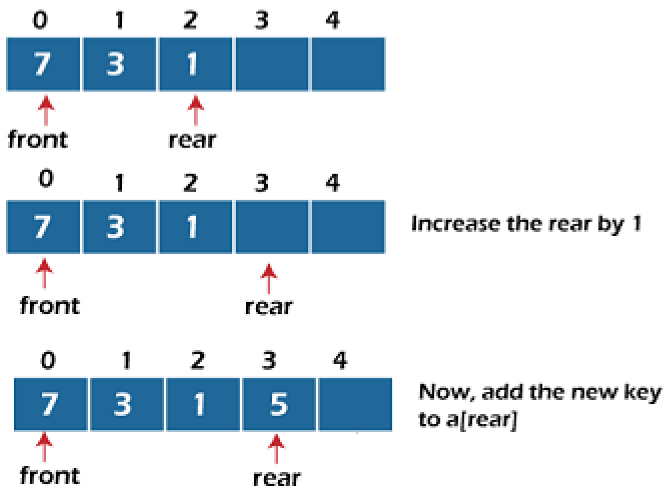
- If the queue is empty, both rear and front are initialized with 0. Now, both will point to the first element.
- Otherwise, check the position of the front if the front is less than 1 (front < 1),

then reinitialize it by **front = n - 1**, i.e., the last index of the array.



2. Insert at the Rear: 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.



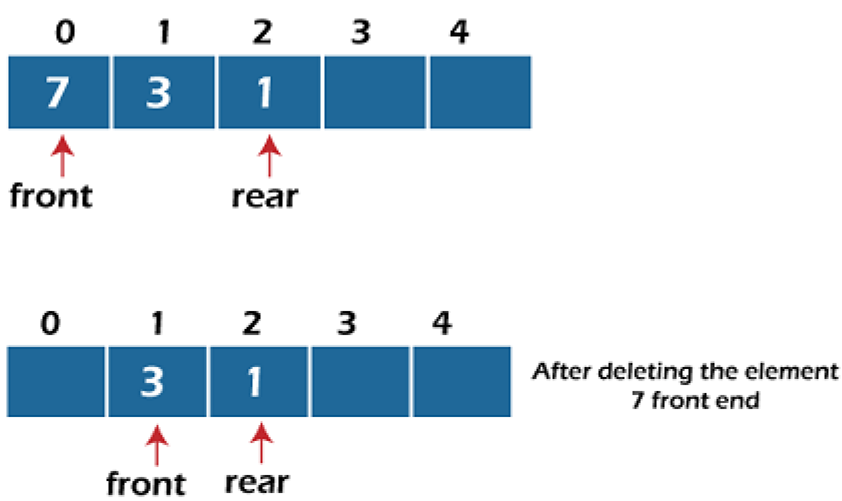
3. Delete from the Front: In this operation, the element is deleted from the front end of the queue. Before implementing the operation, we first have to check whether the queue is empty or not.

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).



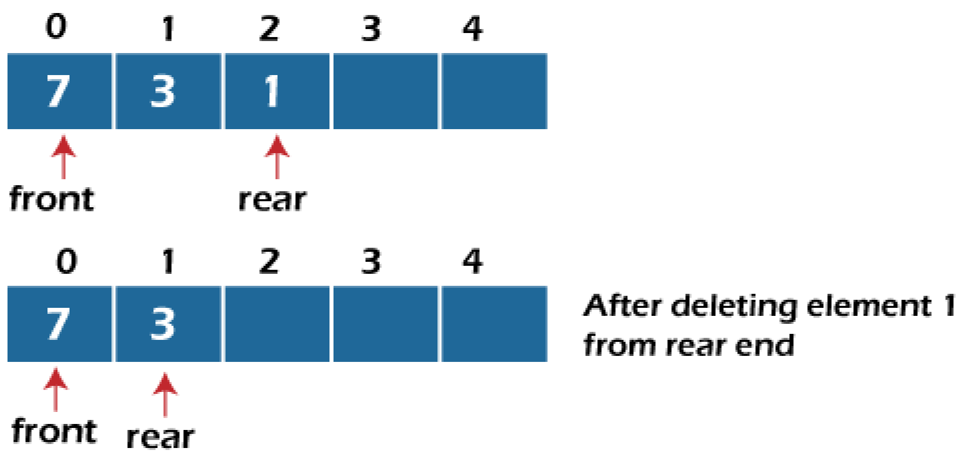
4. Delete from the Rear: In this operation, the element is deleted from the rear end of the queue. Before implementing the operation, we first have to check whether the queue is empty or not.

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.

Properties of Deque:

- Deque is a generalized version of the queue which allows us to insert and delete the element at both ends.

- It does not follow FIFO (first in first out) rule.

Applications of Deque:

- It is used in job scheduling algorithms.
- It supports both stack and queue operations.
- The clockwise and anti-clockwise rotation operations in deque are performed in O(1) time which is helpful in many problems.

Real-time Application of Deque:

- In a web browser's 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.
- Storing a software application's list of undo operations.
- 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.
- 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.
- 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.
- 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.

Advantages of Deque:

- You are able to add and remove items from the both front and back of the queue.
- Deques are faster in adding and removing the elements to the end or beginning.
- The clockwise and anti-clockwise rotation operations are faster in a deque.
- **Dynamic Size:** Deques can grow or shrink dynamically.
- **Efficient Operations**: Deques provide efficient O(1) time complexity for inserting and removing elements from both ends.
- **Versatile**: Deques can be used as stacks (LIFO) or queues (FIFO), or as a combination of both.
- **No Reallocation**: Deques do not require reallocation of memory when elements are inserted or removed.
- **Thread Safe**: Deques can be thread-safe if used with proper synchronization.
- **Cache-Friendly**: Deques have a contiguous underlying storage structure which makes them cache-friendly.

Disadvantages of Deque:

- Deque has no fixed limitations for the number of elements they may contain. This interface supports capacity-restricted deques as well as the deques with no fixed size limit.
- They are less memory efficient than a normal queue.
- **Memory Overhead**: Deques have higher memory overhead compared to other data structures due to the extra pointers used to maintain the double-ended structure.
- **Synchronization**: Deques can cause synchronization issues if not used carefully in multi-threaded environments.
- **Complex Implementation**: Implementing a deque can be complex and error-prone, especially if implementing it manually.
- **Not All Platforms**: Deques may not be supported by all platforms, and may need to be implemented manually.
- **Not Suitable for Sorting**: Deques are not designed for sorting or searching, as these operations require linear time.
- **Limited Functionality**: Deques have limited functionality compared to other data structures such as arrays, lists, or trees.

Implementation of deque

```
#include <stdio.h>
#define size 5
int deque[size];
int f = -1, r = -1;
//  insert_front function will insert the value from the front
void insert_front(int x)
{
```

```
    if((f==0 && r==size-1) || (f==r+1))
    {
        printf("Overflow");
    }
    else if((f==-1) && (r==-1))
    {
        f=r=0;
        deque[f]=x;
    }
    else if(f==0)
    {
        f=size-1;
        deque[f]=x;
    }
    else
    {
        f=f-1;
        deque[f]=x;
    }
}

// insert_rear function will insert the value from the rear
void insert_rear(int x)
{
    if((f==0 && r==size-1) || (f==r+1))
    {
        printf("Overflow");
    }
    else if((f==-1) && (r==-1))
    {
        r=0;
        deque[r]=x;
    }
    else if(r==size-1)
    {
        r=0;
        deque[r]=x;
    }
    else
    {
        r++;
        deque[r]=x;
    }
```

```
}

// display function prints all the value of deque.
void display()
{
    int i=f;
    printf("\nElements in a deque are: ");

    while(i!=r)
    {
        printf("%d ",deque[i]);
        i=(i+1)%size;
    }
     printf("%d",deque[r]);
}

// getfront function retrieves the first value of the deque.
void getfront()
{
    if((f==-1) && (r==-1))
    {
        printf("Deque is empty");
    }
    else
    {
        printf("\nThe value of the element at front is: %d", deque[f]);
    }

}

// getrear function retrieves the last value of the deque.
void getrear()
{
    if((f==-1) && (r==-1))
    {
        printf("Deque is empty");
    }
    else
    {
        printf("\nThe value of the element at rear is %d", deque[r]);
    }

}
```

```
// delete_front() function deletes the element from the front
void delete_front()
{
   if((f==-1) && (r==-1))
   {
      printf("Deque is empty");
   }
   else if(f==r)
   {
      printf("\nThe deleted element is %d", deque[f]);
      f=-1;
      r=-1;

   }
    else if(f==(size-1))
    {
       printf("\nThe deleted element is %d", deque[f]);
       f=0;
    }
    else
    {
        printf("\nThe deleted element is %d", deque[f]);
        f=f+1;
    }
}

// delete_rear() function deletes the element from the rear
void delete_rear()
{
   if((f==-1) && (r==-1))
   {
      printf("Deque is empty");
   }
   else if(f==r)
   {
      printf("\nThe deleted element is %d", deque[r]);
      f=-1;
      r=-1;

   }
    else if(r==0)
    {
       printf("\nThe deleted element is %d", deque[r]);
       r=size-1;
```

```
    }
    else
    {
        printf("\nThe deleted element is %d", deque[r]);
        r=r-1;
    }
}

int main()
{
   insert_front(20);
   insert_front(10);
   insert_rear(30);
   insert_rear(50);
   insert_rear(80);
   display();  // Calling the display function to retrieve the values of deque
   getfront();  // Retrieve the value at front-end
   getrear();  // Retrieve the value at rear-end
   delete_front();
   delete_rear();
   display(); // calling display function to retrieve values after deletion
   return 0;
}
```

OUTPUT:

Elements in a deque are: 10 20 30 50 80
The value of the element at front is: 10
The value of the element at rear is 80
The deleted element is 10
The deleted element is 80
Elements in a deque are: 20 30 50