

**Syllabus:**

# Unit – IV

* **Queues**: Introduction to Queues, Representation of Queues-using Arrays and using Linked list, Implementation of Queues-using Arrays and using Linked list, Application of Queues-Circular Queues, Deques, Priority Queues, Multiple Queues.

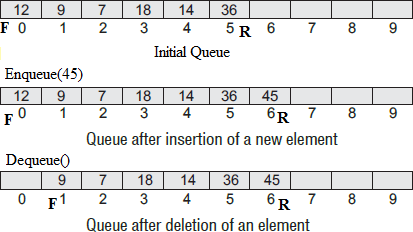
**QUEUE:**

* + Queue is a linear data structure in which elements can be inserted from one end called **rear** and deleted from other end called **front.**
  + The deletion or insertion of elements can take place only at the front or rear end called dequeue and enqueue respectively. The first element that gets added into the queue is the first one to get removed from the queue. Hence the queue is referred to as First-In- First-Out list (FIFO).

### Operations performed on Queue:

There are two possible operations performed on a queue. They are

✔ enqueue: Allows inserting an element at the rear of the queue.

✔ dequeue: Allows removing an element from the front of the queue.

**REPRESENTATION OF QUEUEs:**

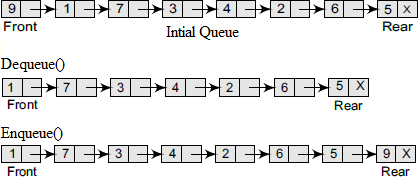
**ARRAYs**: Queues can be easily represented using linear arrays. Every queue has front and rear variables that point to the position from where deletions and insertions can be done, respectively. The array representation of a queue is shown

**Drawback**: The array must be declared to have some fixed size. If we allocate space

for 50 elements in the queue and it hardly uses 20–25 locations, then half of the space will be wasted.

**LINKED LISTs:**

* 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.



**IMPLEMENTATION OF QUEUEs:**

# Using Arrays:

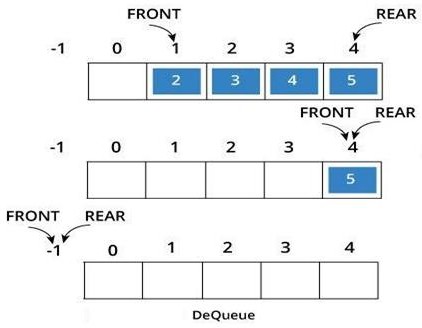
Algorithm for ENQUEUE operation

1. Check whether queue is FULL. **(rear >= SIZE-1)**
2. If it is **FULL**, then display an error message "Queue is FULL!!! Insertion is not possible!!!" and terminate the function.
3. If it is **NOT FULL**, then increment **rear** value by one (rear++) and set **queue[rear] = value.**

Algorithm for DEQUEUE operation

1. Check whether queue is EMPTY. **(front == -1)**
2. If it is **EMPTY,** then display "Queue is EMPTY!!! Deletion is not possible!!!" and terminate the function.
3. If it is **NOT EMPTY**, then display **queue[front]** as deleted element, increment the front value by one **(front ++).** If we are deleting last element both front and rear are equal (front == rear), then set both front and rear to '-1' **(front = rear = -1).**

### Implementation:

* Let us consider a queue, which can hold maximum of five elements.
* Initially the queue is empty. An element can be added to the queue only at the rear end of the queue.
* Before adding an element in the queue, it is checked whether queue is full. If the queue is full, then addition cannot take place. Otherwise, the element is added to the end of the list at therear end. If we are inserting first element into the queue then change front to 0 (Zero).
* Now, delete an element 1. The element deleted is the element at the front of the queue. So the status of the queue is:
* When the last element delete 5. The element deleted at the front of the queue. So the status of the queue is empty. So change the values of front and rear to -1 **(front=rear= -1)**
* The dequeue operation deletes the element from the front of the queue. Before deleting and element, it is checked if the queue is empty. If not the element pointed by front is deleted from the queue and front is now made to point to the next element in the queue.
* **Drawback**: If we implement the queue using an array, we need to specify the array size at the beginning (at compile time). We can't change the size of an array at runtime. So, the queue will only work for a fixed number of elements.

**Code to implement Queue using Arrays**

#include<stdio.h>

#include<stdlib.h>

#define MAX 10

int queue[MAX];

int front=-1,rear=-1;

void enque(int ele);

void deque();

void display();

void Front();

void Rear();

int is\_full();

int is\_empty();

void main(){

int c;

while(c!=8){

printf("----QUEUE DATA STRUCTURES----\n");

printf("\n1.inserting element in queue\n2.deleting element\n3.display queue\n4.display front element\n5.display last element\n6.queue full\n7.queue empty\n8.exit\n");

printf("enter your choice:");

scanf("%d",&c);

switch(c){

case 1:

int ele;

printf("\nenter the elemnt to insert\n");

scanf("%d",&ele);

enque(ele);

break;

case 2:

deque();

break;

case 3:

display();

break;

case 4:

Front();

break;

case 5:

Rear();

break;

case 6:

int l =is\_full();

if(l==1){

printf("\nqueue is full\n");

}else{

printf("\nqueue is not full\n");

}

break;

case 7:

int k = is\_empty();

if(k==1){

printf("\nqueue is empty\n");

}else{

printf("\nqueue has elemnts\n");

}

break;

case 8:

exit(0);

}

}

}

void enque(int ele){

if(rear==MAX-1){

printf("\n queue is overflown\n");

}

if(front==-1){

front=0;

}

rear+=1;

queue[rear]=ele;

}

void deque(){

if(front==-1&&rear==-1){

printf("\nqueue is underflowing\n");

}

else if(front>rear){

printf("\nno elements to delete\n");

}

else{

queue[front++];

}

}

void display(){

if(front==-1&&rear==-1){

printf("\nno elemntd to delete\n");

return;

}

for(int i=front;i<=rear;i++){

printf(" %d ",queue[i]);

}

}

void Front(){

if(front==-1&&rear==-1){

printf("no elemntd to delete");

return;

}

printf(" front element in the queue is %d \n",queue[front]);

}

void Rear(){

if(front==-1&&rear==-1){

printf("no elemntd to delete");

return;

}

printf("rear element is %d \n",queue[rear]);

}

int is\_empty(){

if(front==-1&&rear==-1||front>rear){

return 1;

}

return 0;

}

int is\_full(){

if(rear==MAX-1){

return 1;

}

return 0;

}

**Using Linked List:**

* In a linked queue, each node of the queue consists of two parts i.e. data part and the next part. Each element of the queue points to its immediate next element in the memory.
* In the linked queue, there are two pointers maintained in the memory i.e. front pointer and rear pointer.
* The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.
* Insertion and deletions are performed at rear and front end respectively. If front and rear both are NULL, it indicates that the queue is empty. Initially

**struct** node \*front = NULL, \*rear = NULL;

**Operation on Linked Queue:** There are two basic operations which can be implemented on the linked queues. The operations are Enqueue and Dequeue.

**Enqueue function:** Enqueue function will add the element at the end of the linked list.

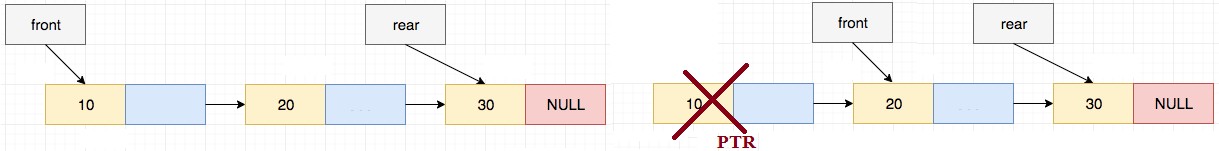
* 1. Declare a new node and allocate memory for it.
  2. If front == NULL, make both front and rear points to the new node.
  3. Otherwise, add the new node in rear->next (end of the list) and make the new node as the rear node. i.e. rear = new node

**Dequeue function:** Dequeue function will remove the first element from the queue.

1. Check whether the queue is empty or not
2. If it is the empty queue (front == NULL), We can't dequeue the element. 3.Otherwise, Make the front node points to the next node. i.e front = front->next;

if front pointer becomes NULL, set the rear pointer also NULL. Free the front node's memory.

### Example: Enqueue()

**Dequeue()**

### Code to implement Queue Using Linked List

### #include<stdio.h>

### #include<stdlib.h>

### #define MAX 10

### struct queue{

### int data;

### struct queue \*next;

### };

### struct queue \*front=NULL,\*rear=NULL;

### static int count=0;

### void enqueue(int ele);

### void display();

### void dequeue();

### int is\_full();

### int is\_empty();

### void print\_front();

### void print\_rear();

### void main(){

### int ele,choice=0;

### while(choice!=8){

### printf("----QUEUE DATA STRUCTURE ----");

### printf("\n1.insert the element\n2.delete the element\n3.display the queue\n4.is queue full\n5.is queue empty\n6.front element\n7.rear element \n8.exit\n");

### printf("enter your choice: ");

### scanf("%d",&choice);

### switch(choice){

### case 1:

### printf("element to be inserted in queue is :");

### scanf("%d",&ele);

### enqueue(ele);

### break;

### case 2:

### dequeue();

### break;

### case 3:

### printf("elementd in the queue are:");

### display();

### printf("\n");

### break;

### case 4:

### int x=is\_full();

### if(x==1){

### printf("queue is full\n");

### }

### else{

### printf("queue is not full\n");

### }

### break;

### case 5:

### int y=is\_empty();

### if(y==1){

### printf("queue is empty\n");

### }

### else{

### printf("queue has elements\n");

### }

### break;

### case 6:

### print\_front();

### break;

### case 7:

### print\_rear();

### break;

### case 8:

### exit(0);

### default:

### printf("enter valid choice");

### }

### }

### }

### void enqueue(int ele){

### struct queue new =(struct queue)malloc(sizeof (struct queue));

### if(count==MAX-1){

### printf("queue overflow\n");

### return;

### }

### else if(front==NULL && rear==NULL){

### new->data=ele;

### new->next=NULL;

### front=new;

### rear=new;

### count++;

### }

### else{

### new->data=ele;

### new->next=NULL;

### rear->next=new;

### rear = new;

### count++;

### }

### }

### void display(){

### if(front==NULL&&rear==NULL&&count==0){

### printf("there are no elements in queue\n");

### return;

### }

### else{

### struct queue \*temp=front;

### while(temp!=NULL){

### printf(" %d ",temp->data);

### temp=temp->next;

### }

### }

### return;

### }

### void dequeue(){

### if(count==0){

### printf("queue underflow\n");

### return;

### }

### else{

### struct queue \*temp = front;

### front=front->next;

### free (temp);

### count--;

### }

### if(front== NULL){

### rear=NULL;

### }

### }

### int is\_full(){

### if(count==MAX-1){

### return 1;

### }

### else{

### return 0;

### }

### }

### int is\_empty(){

### if(count==0&&front==NULL&&rear==NULL){

### return 1;

### }

### else{

### return 0;

### }

### }

### void print\_front(){

### if(count==0&& front==NULL&&rear==NULL){

### printf("queue is empty");

### }

### else{

### printf(" front element in queue is %d\n",front->data);

### }

### }

### void print\_rear(){

### if(count==0&& front==NULL&&rear==NULL){

### printf("queue is empty\n");

### }

### else{

### printf(" rear element in queue is %d\n",rear->data);

### }

### }

**Applications of Queue:**

1. Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
2. Queues are used to transfer data asynchronously between two processes
3. Queues are used as buffers on MP3 players and portable CD players, iPod playlist.
4. Queues are used in Playlist for jukebox to add songs to the end, play from the front.
5. Queues are used in operating system for handling interrupts. The interrupts are handled in the same order as they arrive i.e First come first served.

### TYPES OF QUEUES:

A queue data structure can be classified into the following types:

1. Circular Queue 2. Deque 3. Priority Queue 4. Multiple Queue

**CIRCULAR QUEUEs:**

* In a Linear queue, once the queue is completely full, it's not possible to insert any more elements. When we **dequeue** any element to remove it from the queue, we are actually moving the **front** of the queue forward, but **rear** is still pointing to the last element of the queue, we cannot insert new elements.
* **Circular Queue** is also a linear data structure, which follows the principle of **FIFO**(First In First Out), but instead of ending the queue at the last position, it again starts from the first position after the last, hence making the queue behave like a circular data structure.

**Operations on Circular Queue:** The following are the operations that can be performed

* **enQueue(value):** This function is used to insert the new value in the Queue. The new element is always inserted from the **rear end**.
* **deQueue():** This function deletes an element from the Queue. The deletion in a Queue always takes place from the **front end**.

**Enqueue operation:** The steps of enqueue operation are given below:

* First, we will check whether the Queue is full or not.
* Initially the front and rear are set to -1. When we insert the first element in a Queue, front and rear both are set to 0.
* From 2nd element onwards, When we insert a new element, the rear gets incremented, i.e., ***rear=rear+1***.

### Queue is not full:

* **If rear != max - 1**, then rear will be incremented and the new value will be inserted at the rear end of the queue.
* **If front != 0 and rear = max - 1**, it means that queue is not full, then set the value of rear to 0 and insert the new element there.

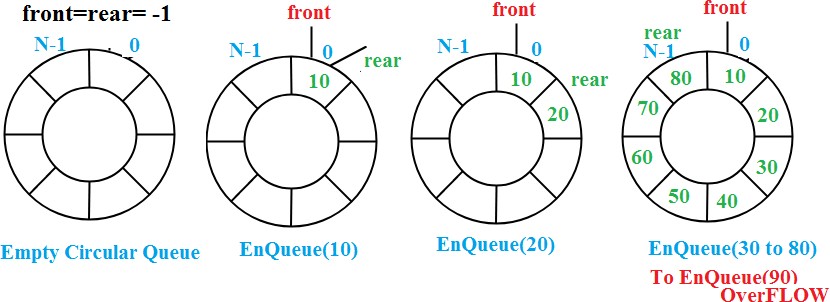
### Queue is full:

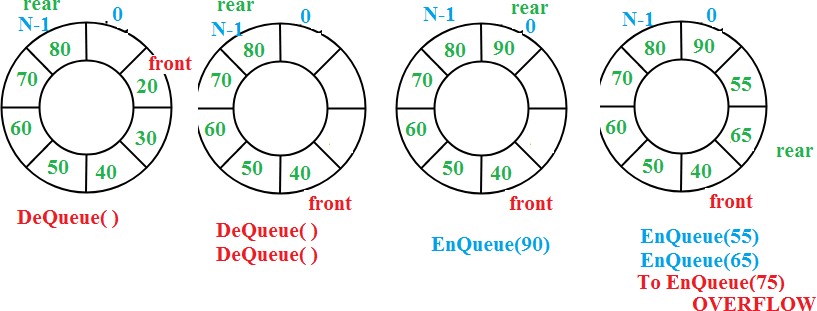
* When **front ==0** && **rear = max-1**, which means that front is at the first position of the Queue and rear is at the last position of the Queue.
* front== rear + 1;

**Dequeue Operation:** The steps of dequeue operation are given below:

* First, we check whether the Queue is empty or not. If the queue is empty, we cannot perform the dequeue operation.
* When the element is deleted, the value of front gets decremented by 1.
* If there is only one element left which is to be deleted, then the front and rear are reset -1.

### Let's understand the enqueue and dequeue operation through the diagrammatic representation.





**Code to implement Circular Queue Using Arrays**

#include<stdio.h>

#define capacity 6

int queue[capacity];

int front = -1, rear = -1;

// Here we check if the Circular queue is full or not

int checkFull ()

{

if ((front == rear + 1) || (front == 0 && rear == capacity - 1))

{

return 1;

}

return 0;

}

// Here we check if the Circular queue is empty or not

int checkEmpty ()

{

if (front == -1)

{

return 1;

}

return 0;

}

// Addtion in the Circular Queue

void enqueue (int value)

{

if (checkFull ())

printf ("Overflow condition\n");

else

{

if (front == -1)

front = 0;

rear = (rear + 1) % capacity;

queue[rear] = value;

printf ("%d was enqueued to circular queue\n", value);

}

}

// Removal from the Circular Queue

int dequeue ()

{

int variable;

if (checkEmpty ())

{

printf ("Underflow condition\n");

return -1;

}

else

{

variable = queue[front];

if (front == rear)

{

front = rear = -1;

}

else

{

front = (front + 1) % capacity;

}

printf ("%d was dequeued from circular queue\n", variable);

return 1;

}

}

// Display the queue

void print ()

{

int i;

if (checkEmpty ())

printf ("Nothing to dequeue\n");

else

{

printf ("\nThe queue looks like: \n");

for (i = front; i != rear; i = (i + 1) % capacity)

{

printf ("%d ", queue[i]);

}

printf ("%d \n\n", queue[i]);

}

}

int main ()

{

// Not possible as the Circular queue is empty

dequeue ();

enqueue (15);

enqueue (20);

enqueue (25);

enqueue (30);

enqueue (35);

print ();

dequeue ();

dequeue ();

print ();

enqueue (40);

enqueue (45);

enqueue (50);

enqueue (55); //Overflow condition

print ();

return 0;

}

**DEQUE:**

Deque or Double Ended Queue is a type of queue in which insertion and removal of elements can be performed from either from the front or rear. Thus, it does not follow FIFO rule (First In First Out).

### Types of Deque:

1. **Input Restricted Deque**: In this deque, input is restricted at a single end but allows deletion at both the ends.
2. **Output Restricted Deque**: In this deque, output is restricted at a single end but allows insertion at both the ends.

### Operations on a Deque

* + Initially take an array (deque) of size **n**. and Set two pointers at the first position and set **front = -1** and **rear = -1**.

1. **Insert at the Front:** This operation adds an element at the front.
   * Check the position of front, If front < 1, we can’t add elements in the front end. Otherwise decrement the front and at front location we can insert the element.
2. **Insert at the Rear:** This operation adds an element to the rear.
   * Check if the array is full. Then the queue is overflow. Otherwise, reinitialize rear = 0 & front=0 for the first insertion, Else, increase rear by 1.and at rear location we can insert the element.
3. **Delete from the Front:** The operation deletes an element from the front.
   * Check 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, front = front + 1.
4. **Delete from the Rear:** This operation deletes an element from the rear.
   * 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, rear = rear - 1.

**Code to Implement Deque Using Arrays**

#include <stdio.h>

#include<stdlib.h>

#define size 5

int deque[size];

int f = -1, r = -1;

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;

}

}

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;

}

}

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]);

}

void getfront()

{

if((f==-1) && (r==-1))

{

printf("Deque is empty");

}

else

{

printf("\nThe value of the element at front is: %d", deque[f]);

}

}

void getrear()

{

if((f==-1) && (r==-1))

{

printf("Deque is empty");

}

else

{

printf("\nThe value of the element at rear is %d", deque[r]);

}

}

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;

}

}

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;

}

}

void is\_Full(){

if((f==0 && r==size-1) || (f==r+1))

{

printf("Overflow");

return;

}

else{

printf("deque containts elements. Elements are:\n");

display();

}

}

void is\_Empty(){

if((f==-1) && (r==-1))

{

printf("Deque is empty");

return;

}

else{

printf("deque containts elements. Elements are:\n");

display();

}

}

int main()

{

int ele;

int userChoice;

while (1) {

printf("\n------ DEQUE DATA STRUCTURE ------\n\n");

printf("\n1. Insert at front");

printf("\n2. Insert at rear");

printf("\n3. Delete at front");

printf("\n4. Delete at rear");

printf("\n5. Get Front");

printf("\n6. Get Rear");

printf("\n7. Deque Full");

printf("\n8. Deque Empty");

printf("\n9. Display Deque");

printf("\n10. Exit");

printf("\n\nEnter Your Choice: ");

scanf("%d", &userChoice);

switch (userChoice)

{

case 1:

printf("\n\nEnter Data: ");

scanf("%d", &ele);

insert\_front(ele);

break;

case 2:

printf("\n\nEnter Data: ");

scanf("%d", &ele);

insert\_rear(ele);

break;

case 3:

delete\_front();

break;

case 4:

delete\_rear();

break;

case 5:

getfront();

break;

case 6:

getrear();

break;

case 7:

is\_Full();

break;

case 8:

is\_Empty();

break;

case 9:

display();

break;

case 10:

exit(0);

break;

default:

printf("\n\tInvalid Choice!");

}

}

return 0;

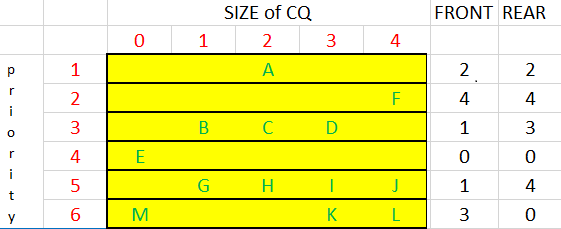
}

**Priority Queue:-**

* 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.

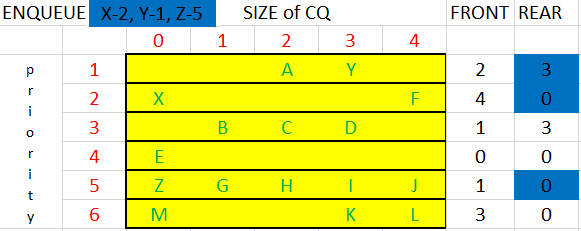
#### Array Representation of a Priority Queue:

* When arrays are used to implement a priority queue, then a separate queue for each priority number is maintained. Each of these queues will be implemented using circular arrays or circular queues. Every individual queue will have its own FRONT and REAR pointers.
* We use a two-dimensional array for this purpose where each queue will be allocated the same amount of space.
* FRONT[P] and REAR[P] contain the front and rear values of row P, where P is the priority number.



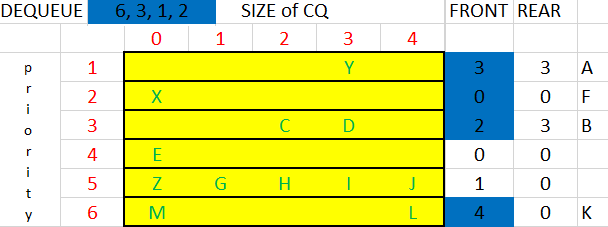
#### Insertion:

* To insert a new element with priority P in the priority queue, add the element at the rear end of row P, where P is the row number as well as the priority number of that element.
* For example, if we have to insert an element X with priority number 2, then the priority queue will be given as shown in Fig.



***Deletion:***

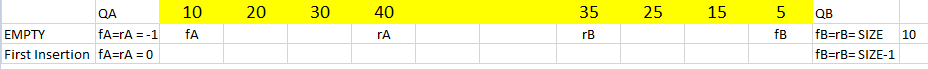
* To delete an element, we find the first nonempty queue and then process the front element of the first non-empty queue.
* In our priority queue, the first non-empty queue is the one with priority number 6 and the front element is K, so K will be deleted and processed first.



**Multiple Queues:-**

* When we implement a queue using an array, the size of the array must be known in advance. If the queue is allocated less space, then frequent overflow conditions will be encountered. To deal with this problem, the code will have to be modified to reallocate more space for the array.
* In case we allocate a large amount of space for the queue, it will result in sheer wastage of the memory. So a better solution to deal with this problem is to have multiple queues or to have more than one queue in the same array of sufficient size.
* An array Queue[n] is used to represent two queues, Queue A and Queue B. The value of n is such that the combined size of both the queues will never exceed n. While operating on these queues, it is important to note one thing—queue A will grow from left to right, whereas queue B will grow from right to left at the same time.

### Example:



* In the above example the array consists two queues like QA and QB. For QA there are pointers like fA(front of QA) and rA(rear of A). similarly for QB are fB & rB.
* Initially for QA, the pointer values of fA=rA= -1. For QB, the pointer values are fB=rB=SIZE. Because initially QA and QB are empty.
* For the first insertion in QA, the values of fA=rA=0. Similarly for QB, the values are fB=rB=SIZE-1.
* From the second insertion onwards we can increment only the rear pointer rA for QA and decrement the rear rB for QB.
* Delete the elements from queue only at front end. In QA, the elements can delete from fA, if you delete the element then increment fA. In QB, the elements can delete from fB, if you delete the element then decrement fB.
* When the condition rA=rB-1 or rA+1=rB meets then the entire queue is full. If you try to insert the element in either of queues it says that QUEUE is OVERFLOW.

**BFS:**

**Breadth First Search (BFS)** is a graph traversal algorithm that explores all the vertices in a graph at the current depth before moving on to the vertices at the next depth level. It starts at a specified vertex and visits all its neighbors before moving on to the next level of neighbors. **BFS** is commonly used in algorithms for pathfinding, connected components, and shortest path problems in graphs.

## How Does the BFS Algorithm Work?

Starting from the root, all the nodes at a particular level are visited first and then the nodes of the next level are traversed till all the nodes are visited.

To do this a queue is used. All the adjacent unvisited nodes of the current level are pushed into the queue and the nodes of the current level are marked visited and popped from the queue.

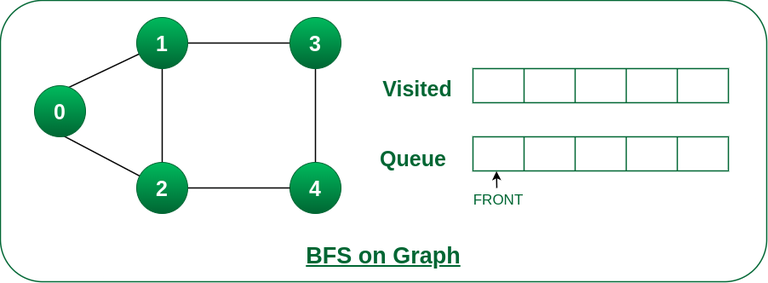
## Breadth First Search (BFS) for a Graph Algorithm:

Let’s discuss the algorithm for the BFS:

1. **Initialization:**Enqueue the starting node into a queue and mark it as visited.
2. **Exploration:**While the queue is not empty:
   * Dequeue a node from the queue and visit it (e.g., print its value).
   * For each unvisited neighbor of the dequeued node:
     + Enqueue the neighbor into the queue.
     + Mark the neighbor as visited.
3. **Termination:** Repeat step 2 until the queue is empty.

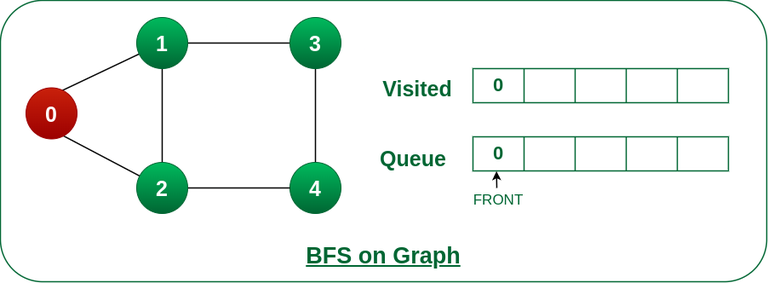
Let us understand the working of the algorithm with the help of the following example.

***Step1:****Initially queue and visited arrays are empty.*

**

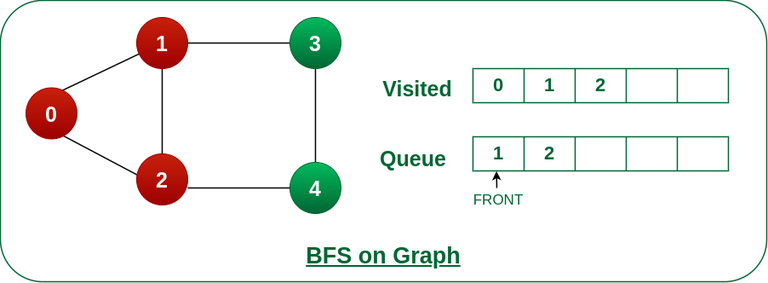
*Queue and visited arrays are empty initially.*

***Step2:****Push node 0 into queue and mark it visited.*

**

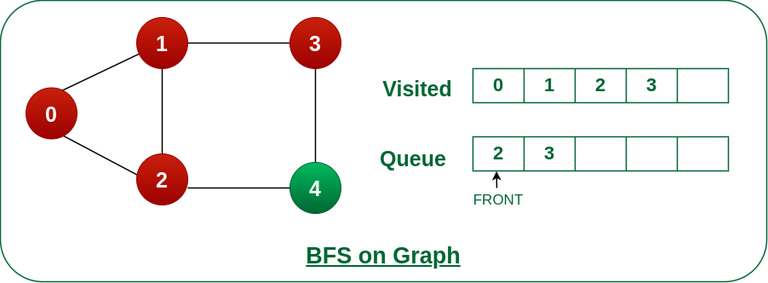
*Push node 0 into queue and mark it visited.*

***Step 3:****Remove node 0 from the front of queue and visit the unvisited neighbours and push them into queue.*

**

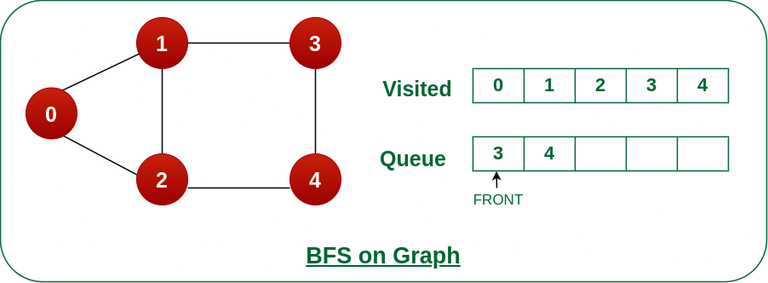
*Remove node 0 from the front of queue and visited the unvisited neighbours and push into queue.*

***Step 4:****Remove node 1 from the front of queue and visit the unvisited neighbours and push them into queue.*

**

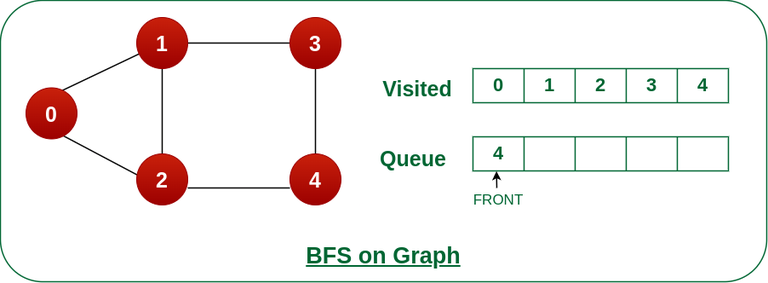
*Remove node 1 from the front of queue and visited the unvisited neighbours and push*

***Step 5:****Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.*

**

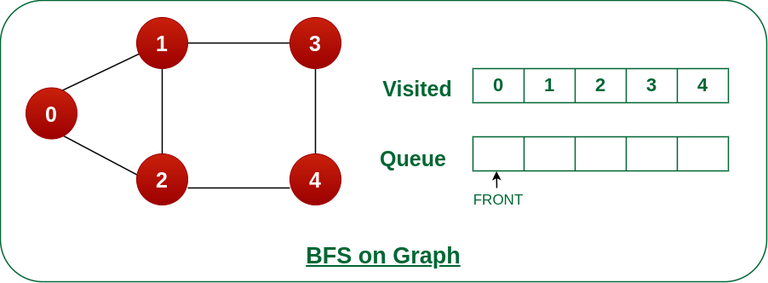
*Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.*

***Step 6:****Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.   
As we can see that every neighbours of node 3 is visited, so move to the next node that are in the front of the queue.*

**

*Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.*

***Steps 7:****Remove node 4 from the front of queue and visit the unvisited neighbours and push them into queue.   
As we can see that every neighbours of node 4 are visited, so move to the next node that is in the front of the queue.*

**

*Remove node 4 from the front of queue and visit the unvisited neighbours and push them into queue.*

*Now, Queue becomes empty, So, terminate these process of iteration.*

**Code to implement Breadth First Search**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_VERTICES 100

struct Node {

int data;

struct Node\* next;

};

void addEdge(struct Node\* adjList[], int u, int v)

{

struct Node\* newNode

= (struct Node\*)malloc(sizeof(struct Node));

newNode->data = v;

newNode->next = adjList[u];

adjList[u] = newNode;

}

void bfs(struct Node\* adjList[], int vertices,

int startNode, int visited[])

{

int queue[MAX\_VERTICES];

int front = 0, rear = 0;

visited[startNode] = 1;

queue[rear++] = startNode;

while (front != rear) {

int currentNode = queue[front++];

printf("%d ", currentNode);

struct Node\* temp = adjList[currentNode];

while (temp != NULL) {

int neighbor = temp->data;

if (!visited[neighbor]) {

visited[neighbor] = 1;

queue[rear++] = neighbor;

}

temp = temp->next;

}

}

}

int main()

{

int vertices = 5;

struct Node\* adjList[vertices];

for (int i = 0; i < vertices; ++i)

adjList[i] = NULL;

addEdge(adjList, 0, 1);

addEdge(adjList, 0, 2);

addEdge(adjList, 1, 3);

addEdge(adjList, 1, 4);

addEdge(adjList, 2, 4);

int visited[vertices];

for (int i = 0; i < vertices; ++i)

visited[i] = 0;

printf(

"Breadth First Traversal starting from vertex 0: ");

bfs(adjList, vertices, 0, visited);

return 0;

}