## Queue

```
#include<stdio.h>
#include<stdlib.h>
# define SIZE 3
void enqueue();
void dequeue();
void show();
int inp_arr[SIZE];
int Rear = - 1;
int Front = - 1;
int main()
{
        int ch;
        while (1)
        {
        printf("1.Enqueue Operation\n");
        printf("2.Dequeue Operation\n");
        printf("3.Display the Queue\n");
        printf("4.Exit\n");
        printf("Enter your choice of operations : ");
        scanf("%d", &ch);
        switch (ch)
        {
        case 1:
        enqueue();
        break;
        case 2:
        dequeue();
        break;
        case 3:
        show();
        break;
        case 4:
        exit(0);
        default:
        printf("Incorrect choice \n");
        }
        }
}

void enqueue()
{
        int insert_item;
```

```
        if (Rear == SIZE - 1)
        printf("Overflow \n");
        else
        {
        if (Front == - 1)

        Front = 0;
        printf("Element to be inserted in the Queue\n : ");
        scanf("%d", &insert_item);
        Rear = Rear + 1;
        inp_arr[Rear] = insert_item;
        }
}

void dequeue()
{
        if (Front == - 1 || Front > Rear)
        {
        printf("Underflow \n");
        return ;
        }
        else
        {
        printf("Element deleted from the Queue: %d\n", inp_arr[Front]);
        Front = Front + 1;
        }
}

void show()
{

        if (Front == - 1)
        printf("Empty Queue \n");
        else
        {
        printf("Queue: \n");
        for (int i = Front; i <= Rear; i++)
        printf("%d ", inp_arr[i]);
        printf("\n");
        }
}
```

## Queue Program 2

```
#include<stdio.h>
#include<stdlib.h>

struct queue
{
	int size;
	int f;
	int r;
	int* arr;
};

int isEmpty(struct queue *q){
	if(q->r==q->f){
	return 1;
	}
	return 0;
}

int isFull(struct queue *q){
	if(q->r==q->size-1){
	return 1;
	}
	return 0;
}

void enqueue(struct queue *q, int val){
	if(isFull(q)){
	printf("This Queue is full\n");
	}
	else{
	q->r++;
	q->arr[q->r] = val;
	printf("Enqued element: %d\n", val);
	}
}

int dequeue(struct queue *q){
	int a = -1;
	if(isEmpty(q)){
	printf("This Queue is empty\n");
	}
	else{
```

```
        q->f++;
        a = q->arr[q->f];
        }
        return a;
}

int main(){
        struct queue q;
        q.size = 4;
        q.f = q.r = -1;
        q.arr = (int*) malloc(q.size*sizeof(int));

        // Enqueue few elements
        enqueue(&q, 14);
        enqueue(&q, 5);
        enqueue(&q, 1);
        enqueue(&q, 17);
        enqueue(&q, 19);
        printf("Dequeuing element %d\n", dequeue(&q));
        printf("Dequeuing element %d\n", dequeue(&q));
        printf("Dequeuing element %d\n", dequeue(&q));
        enqueue(&q, 2);
        enqueue(&q, 4);
        enqueue(&q, 7);

        if(isEmpty(&q)){
        printf("Queue is empty\n");
        }
        if(isFull(&q)){
        printf("Queue is full\n");
        }

        return 0;
}
```

## Circular Queue

```
#include<stdio.h>
#include<stdlib.h>

struct circularQueue
{
```

```
        int size;
        int f;
        int r;
        int* arr;
};


int isEmpty(struct circularQueue *q){
        if(q->r==q->f){
        return 1;
        }
        return 0;
}

int isFull(struct circularQueue *q){
        if((q->r+1)%q->size == q->f){
        return 1;
        }
        return 0;
}

void enqueue(struct circularQueue *q, int val){
        if(isFull(q)){
        printf("This Queue is full");
        }
        else{
        q->r = (q->r +1)%q->size;
        q->arr[q->r] = val;
        printf("Enqued element: %d\n", val);
        }
}

int dequeue(struct circularQueue *q){
        int a = -1;
        if(isEmpty(q)){
        printf("This Queue is empty");
        }
        else{
        q->f = (q->f +1)%q->size;
        a = q->arr[q->f];
        }
        return a;
}
```

```
int main(){
	struct circularQueue q;
	q.size = 4;
	q.f = q.r = 0;
	q.arr = (int*) malloc(q.size*sizeof(int));

	// Enqueue few elements
	enqueue(&q, 5);
	enqueue(&q, 15);
	enqueue(&q, 47);
	enqueue(&q, 4);
	enqueue(&q, 7);
	printf("Dequeuing element %d\n", dequeue(&q));
	printf("Dequeuing element %d\n", dequeue(&q));
	printf("Dequeuing element %d\n", dequeue(&q));
	enqueue(&q, 10);
	enqueue(&q, 11);
	enqueue(&q, 12);

	if(isEmpty(&q)){
	printf("Queue is empty\n");
	}
	if(isFull(&q)){
	printf("Queue is full\n");
	}

	return 0;
}
```

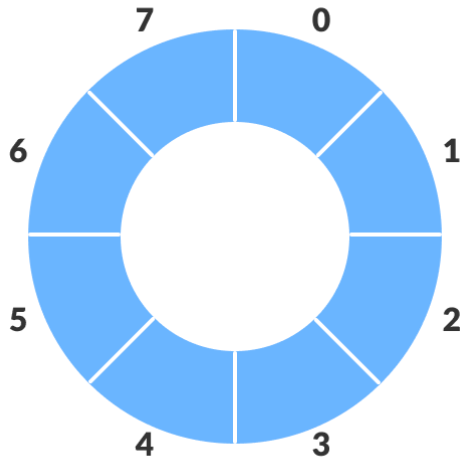
# How Circular Queue Works

Circular Queue works by the process of circular increment i.e. when we try to increment the pointer and we reach the end of the queue, we start from the beginning of the queue.

Here, the circular increment is performed by modulo division with the queue size. That is,

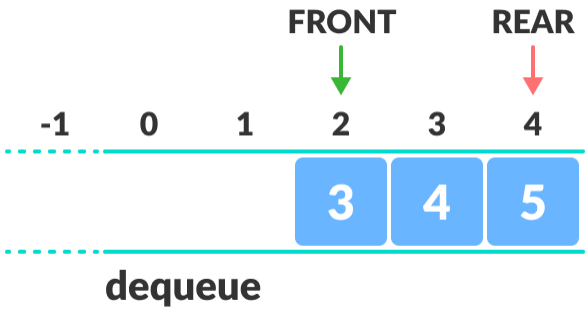
if REAR + 1 == 5 (overflow!), REAR = (REAR + 1)%5 = 0 (start of queue)

A circular queue is the extended version of a regular queue where the last element is connected to the first element. Thus forming a circle-like structure.




***Circular queue representation***

The circular queue solves the major limitation of the normal queue. In a normal queue, after a bit of insertion and deletion, there will be non-usable empty space.




***Limitation of the regular Queue***

Here, indexes **0** and **1** can only be used after resetting the queue (deletion of all elements). This reduces the actual size of the queue.

# Circular Queue Operations

The circular queue work as follows:

- two pointers *FRONT* and *REAR*
- *FRONT* track the first element of the queue

- *REAR* track the last elements of the queue
- initially, set value of *FRONT* and *REAR* to -1

## 1. Enqueue Operation

- check if the queue is full
- for the first element, set value of *FRONT* to 0
- circularly increase the *REAR* index by 1 (i.e. if the rear reaches the end, next it would be at the start of the queue)
- add the new element in the position pointed to by *REAR*

## 2. Dequeue Operation

- check if the queue is empty
- return the value pointed by *FRONT*
- circularly increase the *FRONT* index by 1
- for the last element, reset the values of *FRONT* and *REAR* to -1

However, the check for full queue has a new additional case:

- Case 1: *FRONT* = 0 && `REAR == SIZE - 1`
- Case 2: `FRONT = REAR + 1`

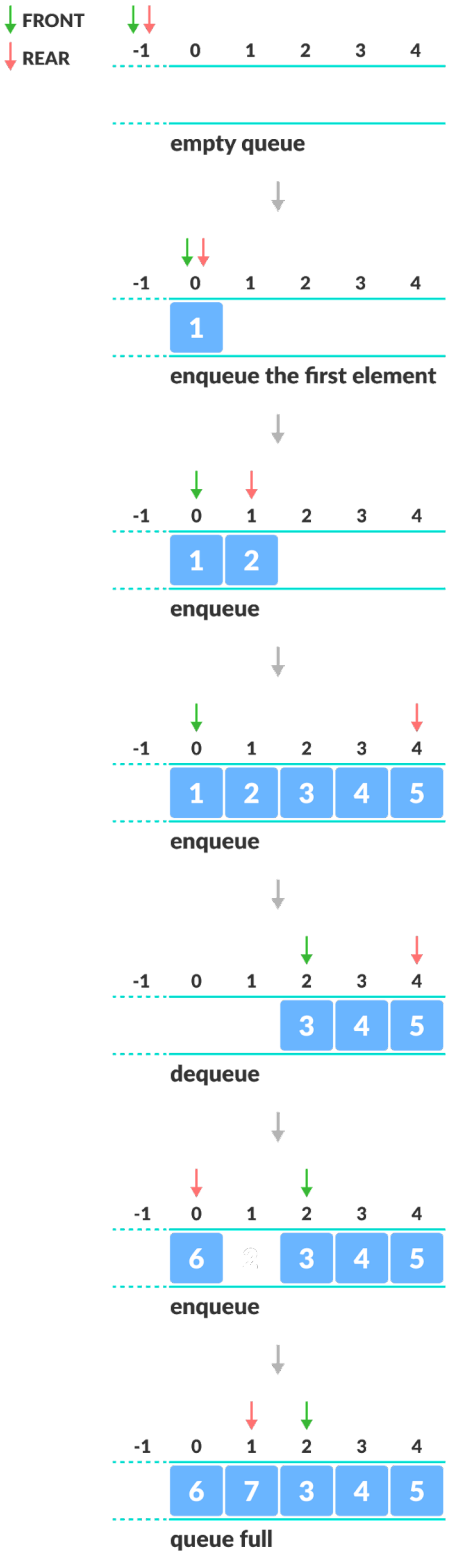
The second case happens when *REAR* starts from 0 due to circular increment and when its value is just 1 less than *FRONT*, the queue is full.

## *// Circular Queue implementation in C*

```
#include <stdio.h>
#define SIZE 5
int items[SIZE];
int front = -1, rear = -1;
// Check if the queue is full
int isFull() {
  if ((front == rear + 1) || (front == 0 && rear == SIZE - 1)) return 1;
  return 0;
}
// Check if the queue is empty
int isEmpty() {
  if (front == -1) return 1;
  return 0;
}
// Adding an element
void enQueue(int element) {
  if (isFull())
```

```
        printf("\n Queue is full!! \n");
  else {
        if (front == -1) front = 0;
        rear = (rear + 1) % SIZE;
        items[rear] = element;
        printf("\n Inserted -> %d", element);
  }
}
// Removing an element
int deQueue() {
  int element;
  if (isEmpty()) {
        printf("\n Queue is empty !! \n");
        return (-1);
  } else {
        element = items[front];
        if (front == rear) {
        front = -1;
        rear = -1;
        }
        // Q has only one element, so we reset the
        // queue after dequeing it. ?
        else {
        front = (front + 1) % SIZE;
        }
        printf("\n Deleted element -> %d \n", element);
        return (element);
  }
}
// Display the queue
void display() {
  int i;
  if (isEmpty())
        printf(" \n Empty Queue\n");
  else {
        printf("\n Front -> %d ", front);
        printf("\n Items -> ");
        for (i = front; i != rear; i = (i + 1) % SIZE) {
        printf("%d ", items[i]);
        }
        printf("%d ", items[i]);
        printf("\n Rear -> %d \n", rear);
  }
}
```

```
int main() {
  // Fails because front = -1
  deQueue();
  enQueue(1);
  enQueue(2);
  enQueue(3);
  enQueue(4);
  enQueue(5);
  // Fails to enqueue because front == 0 && rear == SIZE - 1
  enQueue(6);
  display();
  deQueue();
  display();
  enQueue(7);
  display();
  // Fails to enqueue because front == rear + 1
  enQueue(8);
  return 0;
}
```




***Enque and Deque Operations***

# Applications of Circular Queue

- CPU scheduling
- Memory management
- Traffic Management

# Priority Queue Applications

Some of the applications of a priority queue are:

- Dijkstra's algorithm
- for implementing stack
- for load balancing and interrupt handling in an operating system
- for data compression in Huffman code

The priority queue in a data structure is used in Google Maps for searching the optimal path to reach any destination. Dijkstra’s Shortest Path algorithm utilizes a min priority queue to store all possible paths to reach the destination, considering distance as a parameter for priority assignment. In the end, it will return the element with the highest priority as the optimal route.

# Introduction to Priority Queue in Data Structure

Priority Queue is an abstract data type that performs operations on data elements per their priority. To understand it better, first analyze the real-life scenario of a priority queue.

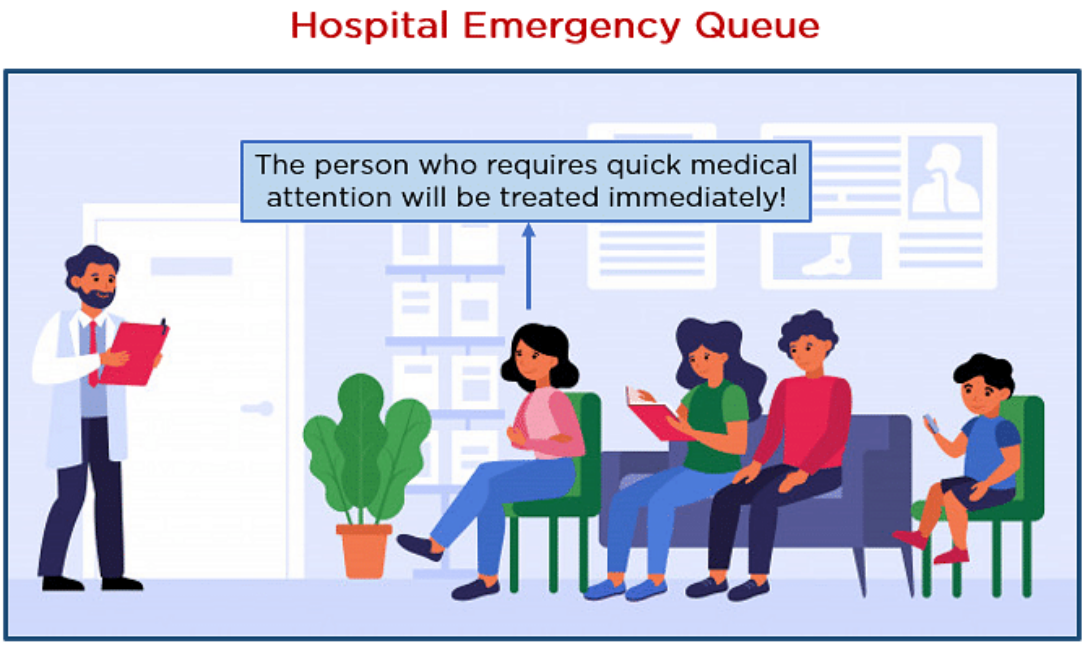
The hospital emergency queue is an ideal real-life example of a priority queue. In this queue of patients, the patient with the most critical situation is the first in a queue, and the patient who doesn’t need immediate medical attention will be the last. In this queue, the priority depends on the medical condition of the patients.

# Characteristics of Priority Queue

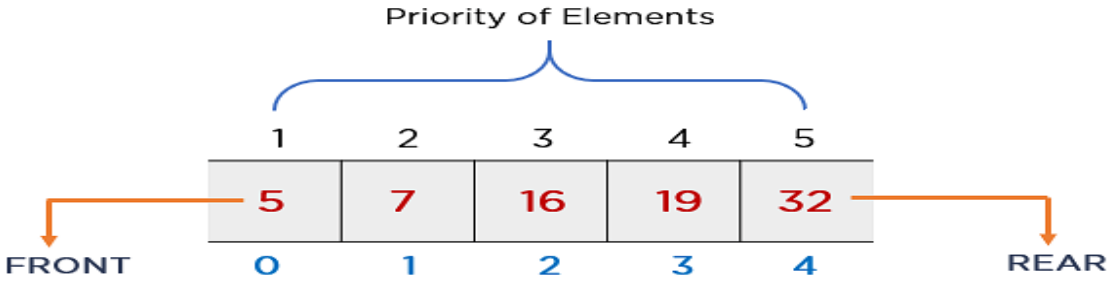
Priority queue in a data structure is an extension of a linear queue that possesses the following properties:

- Every element has a certain priority assigned to it.
- Every element of this queue must be comparable.
- It will delete the element with higher priority before the element with lower priority.

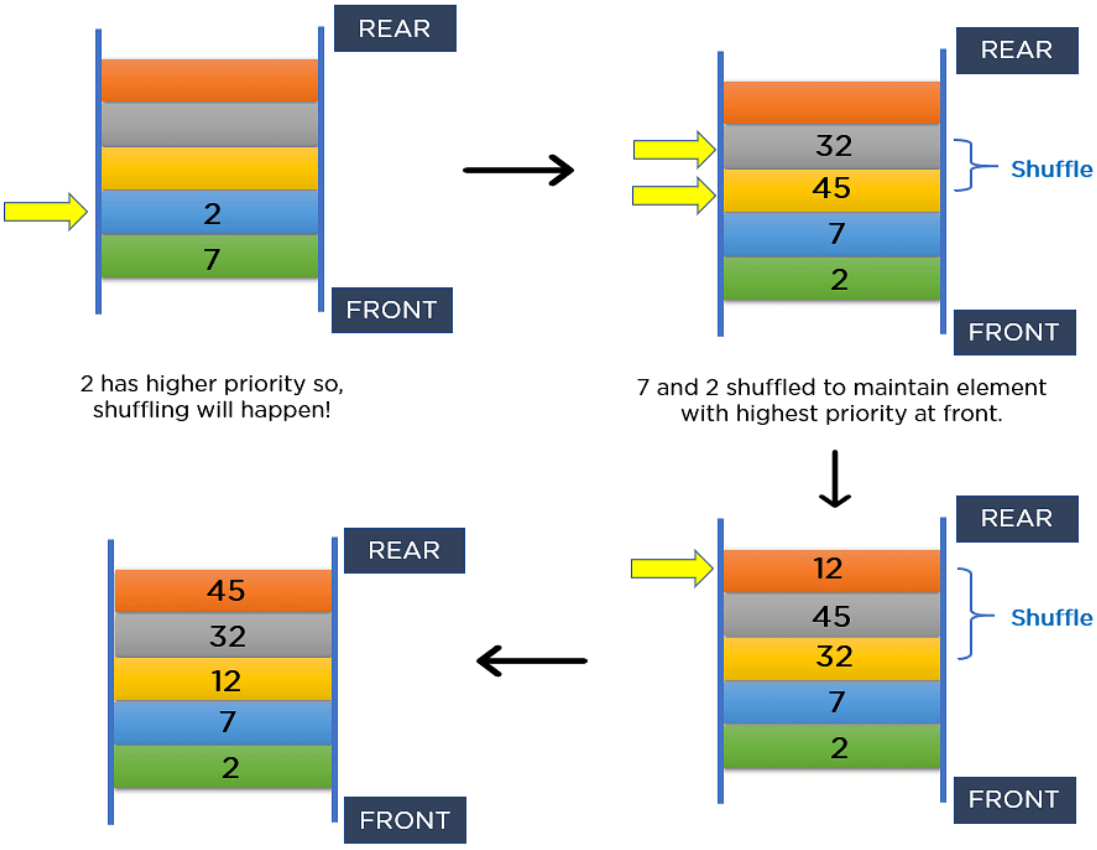
- If multiple elements have the same priority, it does their removal from the queue according to the FCFS principle.




The priority queue in data structure resembles the properties of the hospital emergency queue. Thus, it is highly used in sorting algorithms. It behaves similar to a linear queue except for the fact that each element has some priority assigned to it. The priority of elements determines the order of removal in a queue, i.e., the element with higher priority will leave the queue first, whereas the element with the lowest priority at last.



Now, understand these properties with the help of an example. Consider you have to insert 7, 2, 45, 32, and 12 in a priority queue. The element with the least value has the highest property. Thus, you should maintain the lowest element at the front node.




The image above shows how it maintains the priority during insertion in a queue.

### Difference between Priority Queue and Normal Queue

In a queue, the **first-in-first-out rule** is implemented whereas, in a priority queue, the values are removed **on the basis of priority**. The element with the highest priority is removed first.

## How to Implement Priority Queue?

Priority queue can be implemented using the following data structures:

- Arrays
- Linked list

- Heap data structure
- Binary search tree

# What is a Priority Queue?

It is an abstract data type that provides a way to maintain the dataset. The “normal” queue follows a pattern of first-in-first-out. It dequeues elements in the same order followed at the time of insertion operation. However, the element order in a priority queue depends on the element’s priority in that queue. The priority queue moves the highest priority elements at the beginning of the priority queue and the lowest priority elements at the back of the priority queue.

It supports only those elements that are comparable. Hence, a **priority queue in the data structure** arranges the elements in either ascending or descending order.

You can think of a priority queue as several patients waiting in line at a hospital. Here, the situation of the patient defines the priority order. The patient with the most severe injury would be the first in the queue.

# What are the Characteristics of a Priority Queue?

A queue is termed as a priority queue if it has the following characteristics:

- Each item has some priority associated with it.
- An item with the highest priority is moved at the front and deleted first.
- If two elements share the same priority value, then the priority queue follows the first-in-first-out principle for de queue operation.

# What are the Types of Priority Queue?

A priority queue is of two types:

- Ascending Order Priority Queue
- Descending Order Priority Queue

## Ascending Order Priority Queue

An ascending order priority queue gives the highest priority to the lower number in that queue. For example, you have six numbers in the priority queue that are 4, 8, 12, 45, 35, 20. Firstly, you will arrange these numbers in ascending order. The new list is as follows: 4, 8, 12, 20. 35, 45. In this list, 4 is the smallest number. Hence, the ascending order priority queue treats number 4 as the highest priority.

| 4 | 8 | 12 | 20 | 35 | 45 |
|---|---|---|---|---|---|

In the above table, 4 has the highest priority, and 45 has the lowest priority.

## Descending Order Priority Queue

A descending order priority queue gives the highest priority to the highest number in that queue. For example, you have six numbers in the priority queue that are 4, 8, 12, 45, 35, 20. Firstly, you will arrange these numbers in ascending order. The new list is as follows: 45, 35, 20, 12, 8, 4. In this list, 45 is the highest number. Hence, the descending order priority queue treats number 45 as the highest priority.

| 45 | 35 | 20 | 12 | 8 | 4 |
|---|---|---|---|---|---|

In the above table, 4 has the lowest priority, and 45 has the highest priority.

## 1. What are the applications of a priority queue?

The priority queue is a special queue where the elements are inserted on the basis of their priority. This feature comes to be useful in the implementation of various other data structures. The following are some of the most popular applications of the priority queue:
1. **Dijkstra's Shortest Path algorithm:** Priority queue can be used in Dijkstra's Shortest Path algorithm when the graph is stored in the form of the adjacency list.
2. **Prim's Algorithm:** Prim's algorithm uses the priority queue to the values or keys of nodes and draws out the minimum of these values at every step.
**Data Compression**: Huffman codes use the priority queue to compress the data.
**Operating Systems:** The priority queue is highly useful for operating systems in various processes such as load balancing and interruption handling.

## 2. What approach is used in the implementation of the priority queue using array?

The approach used in the implementation of the priority queue using an array is simple. A structure is created to store the values and priority of the element and then the array of that structure is created to store the elements. The following operations are involved in this implementation:
1. enqueue()-This function inserts the elements at the end of the queue.
2. peek() - This function will traverse the array to return the element with the highest priority. If it finds two elements having the same priority, it returns the highest value element among them.

3. dequeue() - This function will shift all the elements, 1 position to the left of the element returned by the peek() function and decrease the size.

### 3. What is the difference between max heap and min heap?

he following illustrates the difference between max heap and min-heap.
**Min Heap -** In a min-heap, the key of the root node must be less than or equal to the keys of its children node. It uses ascending priority. The node with the smallest key is the priority. The smallest element is popped before any other element.
**Max Heap -** In a max heap, the key of the root node must be greater than or equal to the key of its children's nodes. It uses descending priority. The node with the largest key is the priority. The largest element is popped before any other element.

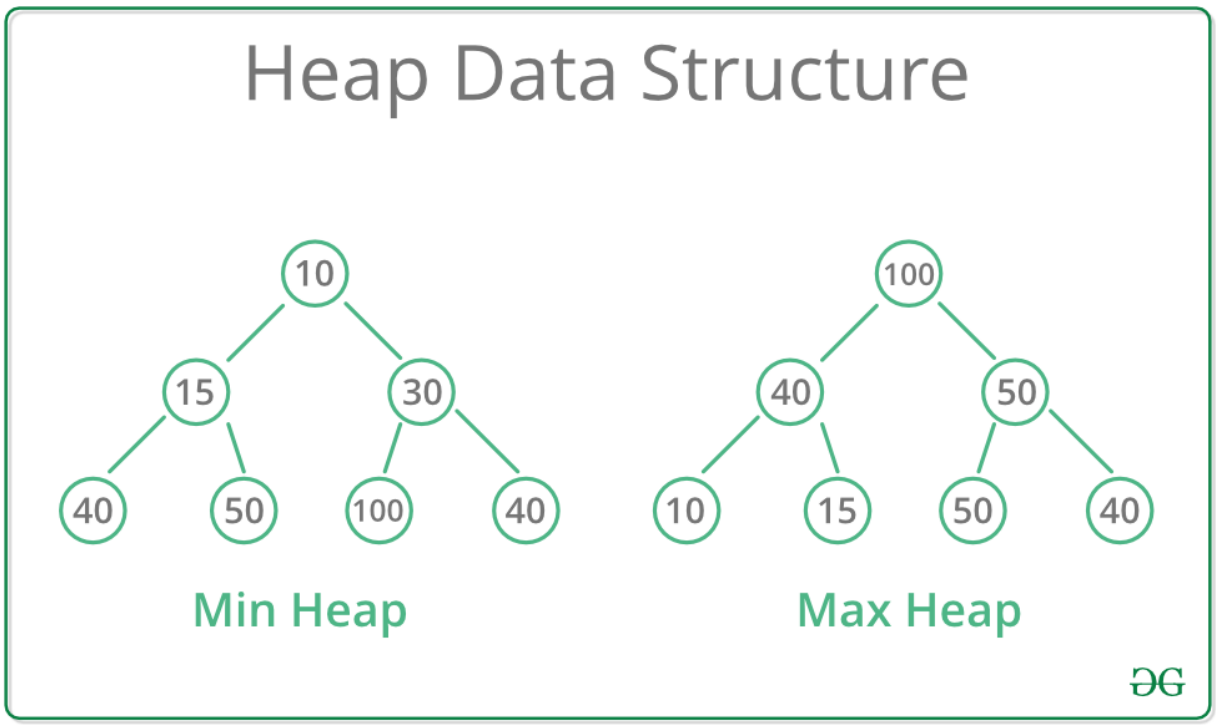
### Applications of Priority queue

**The following are the applications of the priority queue:**

- It is used in the Dijkstra's shortest path algorithm.
- It is used in prim's algorithm
- It is used in data compression techniques like Huffman code.
- It is used in heap sort.
- It is also used in operating system like priority scheduling, load balancing and interrupt handling.

## What is Heap Data Structure?

A Heap is a special Tree-based data structure in which the tree is a complete binary tree.




Heap Data Structure

## Operations of Heap Data Structure:

- **Heapify:** a process of creating a heap from an array.
- **Insertion:** process to insert an element in existing heap time complexity O(log N).
- **Deletion:** deleting the top element of the heap or the highest priority element, and then organizing the heap and returning the element with time complexity O(log N).
- **Peek:** to check or find the first (or can say the top) element of the heap.

## Types of Heap Data Structure

Generally, Heaps can be of two types:

1. **Max-Heap**: In a Max-Heap the key present at the root node must be greatest among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.

2. **Min-Heap**: In a Min-Heap the key present at the root node must be minimum among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.