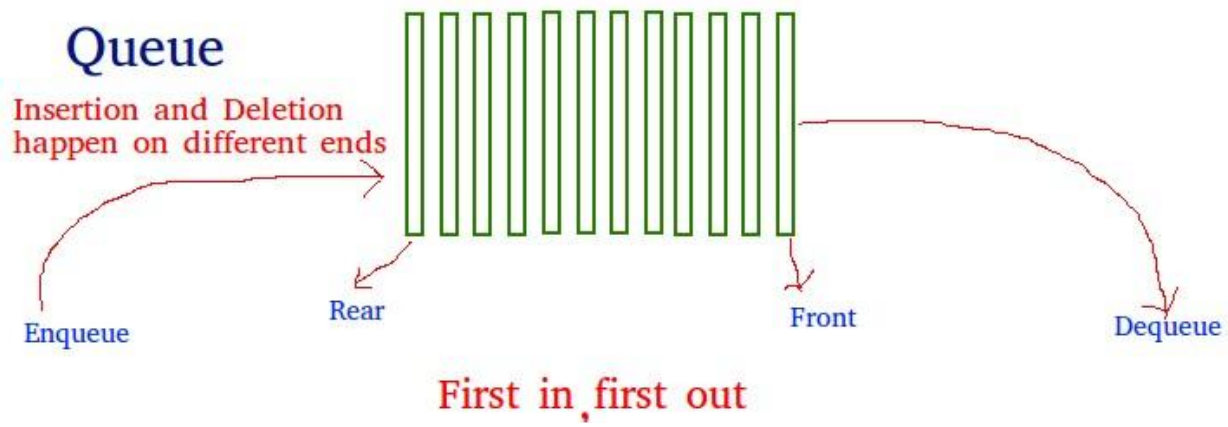
Introduction to Queues

Like *Stack* data structure, ***Queue*** is also a linear data structure which follows a particular order in which the operations are performed. The order is **F**irst **I**n **F**irst **O**ut (FIFO) which means that the element which is inserted first in the queue will be the first one to be removed from the queue. A good example of queue is any queue of consumers for a resource where the consumer that came first is served first.

The difference between stacks and queues is in removing. In a stack we remove the most recently added item; in a queue, we remove the least recently added item.

Operations on Queue: Mainly the following four basic operations are performed on queue:

- **Enqueue:** Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition.
- **Dequeue:** Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition.
- **Front:** Get the front item from queue.
- **Rear:** Get the last item from queue.



Array implementation Of Queue: For implementing a *queue*, we need to keep track of two indices, front and rear. We enqueue an item at the rear and dequeue an item from the front. If we simply increment front and rear indices, then there may be problems, the front may reach the end of the array. The solution to this problem is to increase front and rear in a circular manner.

Consider an Array of size **N** is taken to implement a queue. Initially, the size of the queue will be zero(0). The total capacity of the queue will be the size of the array i.e. N. Now initially, the index *front* will be equal to 0, and *rear* will be equal to N-1. Every time on inserting an item, the index *rear* will increment by one, so increment it as: **rear = (rear + 1)%N** and everytime on removing an item, the front index will shift to right by 1 place so increment it as: **front = (front + 1)%N**.

Example:

```
Array = queue[N].
front = 0, rear = N-1.
N = 5.

Operation 1:
enque(5);
front = 0,
rear = (N-1 + 1)%N = 0.
Queue contains: [5].

Operation 2:
enque(10);
front = 0,
rear = (rear + 1)%N = (0 + 1)%N = 1.
Queue contains: [5, 10].

Operation 3:
enque(15);
front = 0,
rear = (rear + 1)%N = (1 + 1)%N = 2.
Queue contains: [5, 10, 15].

Operation 4:
deque();
print queue[front];
front = (front + 1)%N = (0 + 1)%N = 1.
Queue contains: [10, 15].
```

Below is the Array implementation of queue in C++ and Java:

C++ Java

```cpp
// CPP program for array implementation of queue
#include <bits/stdc++.h>
using namespace std;
// A structure to represent a queue
class Queue
{
    public:
    int front, rear, size;
    unsigned capacity;
    int* array;
};
// function to create a queue of given capacity. It initializes size of queue as 0
Queue* createQueue(unsigned capacity)
{
    Queue* queue = new Queue();
    queue->capacity = capacity;
    queue->front = queue->size = 0;
    queue->rear = capacity - 1; // This is important, see the enqueue
    queue->array = new int[(queue->capacity * sizeof(int))];
    return queue;
}
// Queue is full when size  becomes equal to the capacity
int isFull(Queue* queue)
{ return (queue->size == queue->capacity); }
// Queue is empty when size is 0
int isEmpty(Queue* queue)
{ return (queue->size == 0); }
// Function to add an item to the queue. It changes rear and size
void enqueue(Queue* queue, int item)
```

```
{
    if (isFull(queue))
        return;
    queue->rear = (queue->rear + 1) % queue->capacity;
    queue->array[queue->rear] = item;
    queue->size = queue->size + 1;
    cout << item << " enqueued to queue\n";
}
// Function to remove an item from queue.  It changes front and size
int dequeue(Queue* queue)
{
    if (isEmpty(queue))
        return INT_MIN;
    int item = queue->array[queue->front];
    queue->front = (queue->front + 1) % queue->capacity;
    queue->size = queue->size - 1;
    return item;
}
// Function to get front of queue
int front(Queue* queue)
{
    if (isEmpty(queue))
        return INT_MIN;
    return queue->array[queue->front];
}
// Function to get rear of queue
int rear(Queue* queue)
{
    if (isEmpty(queue))
        return INT_MIN;
    return queue->array[queue->rear];
}
// Driver code
int main()
{
    Queue* queue = createQueue(1000);
    enqueue(queue, 10);
    enqueue(queue, 20);
    enqueue(queue, 30);
    enqueue(queue, 40);
    cout<<dequeue(queue)<<" dequeued from queue\n";
    cout << "Front item is " << front(queue) << endl;
    cout << "Rear item is " << rear(queue) << endl;
    return 0;
}
```

Output:

```
10 enqueued to queue

20 enqueued to queue

30 enqueued to queue

40 enqueued to queue

10 dequeued from queue

Front item is 20

Rear item is 40
```

Time Complexity: Time complexity of all operations like enqueue(), dequeue(), isFull(), isEmpty(), front() and rear() is O(1). There is no loop in any of the operations.

Applications of Queue: Queue is used when things don't have to be processed immediatly, but have to be processed in **F**irst **I**n**F**irst **O**ut order like Breadth First Search. This property of Queue makes it also useful in following kind of scenarios.

1. When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.
2. When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.