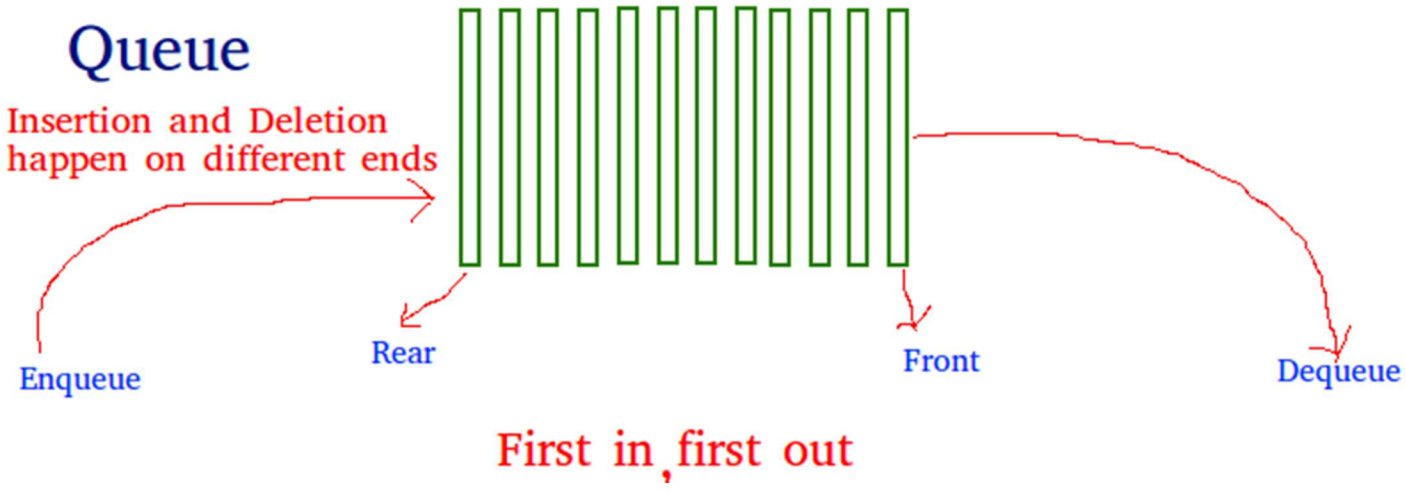
Queue Data Structure

- A Queue is a linear structure which follows a particular order in which the operations are performed.
- The order is First in First out (FIFO).
- A good example of a 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 item the most recently added; in a queue, we remove the item the least recently added.



Definition

Queue is also an abstract data type or a linear data structure, just like stack data structure, in which the first element is inserted from one end called the **REAR**(also called **tail**), and the removal of existing element takes place from the other end called as **FRONT**(also called **head**).

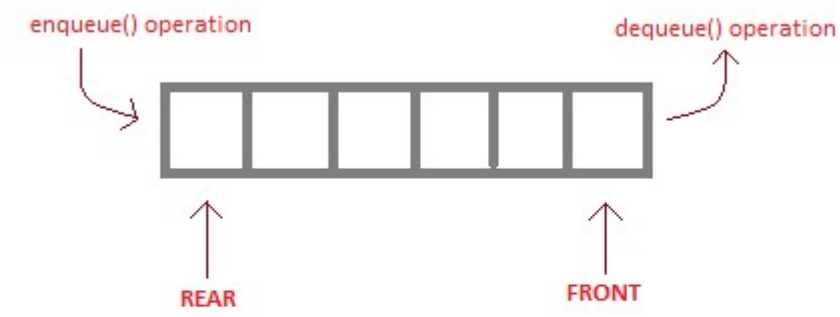
This makes queue as **FIFO** (First in First Out) data structure, which means that element inserted first will be removed first.

Which is exactly how queue system works in real world.

If you go to a ticket counter to buy movie tickets, and are first in the queue, then you will be the first one to get the tickets. Right?

Same is the case with Queue data structure. Data inserted first, will leave the queue first.

The process to add an element into queue is called **Enqueue** and the process of removal of an element from queue is called **Dequeue**.



enqueue( ) is the operation for adding an element into Queue.

dequeue( ) is the operation for removing an element from Queue .

QUEUE DATA STRUCTURE

Basic features of Queue

1. Like stack, queue is also an ordered list of elements of similar data types.
2. Queue is a FIFO(First in First Out) structure.
3. Once a new element is inserted into the Queue, all the elements inserted before the new element in the queue must be removed, to remove the new element.
4. peek( ) function is oftenly used to return the value of first element without dequeuing it.

Basic Operations of Queue

A queue is an object or more specifically an abstract data structure (ADT) that allows the following operations:

- **Enqueue**: Add an element to the end of the queue
- **Dequeue**: Remove an element from the front of the queue
- **IsEmpty**: Check if the queue is empty
- **IsFull**: Check if the queue is full
- **Peek**: Get the value of the front of the queue without removing it

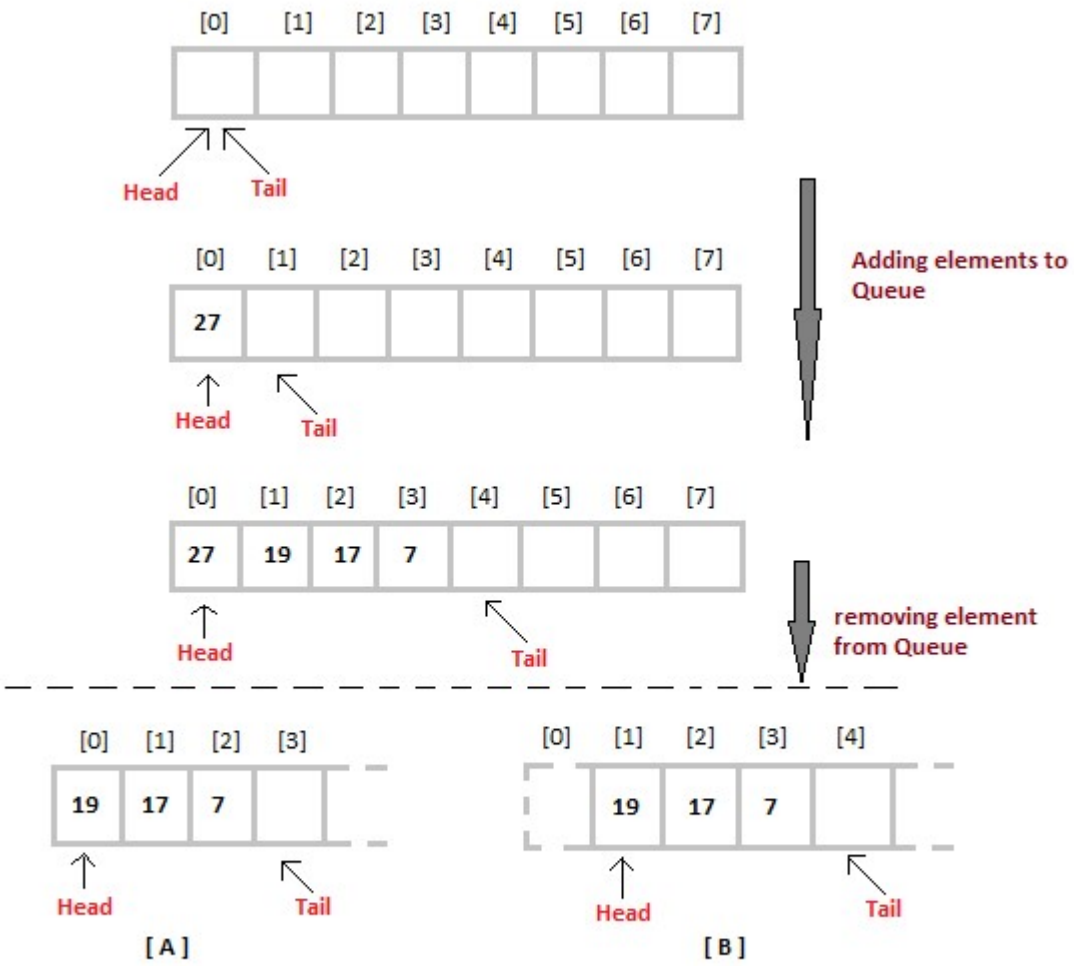
Applications of Queue

Queue, as the name suggests is used whenever we need to manage any group of objects in an order in which the first one coming in, also gets out first while the others wait for their turn, like in the following scenarios:

1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
2. In real life scenario, Call Center phone systems uses Queues to hold people calling them in an order, until a service representative is free.
3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive i.e First come first served.

Implementation of Queue Data Structure

- Queue can be implemented using an Array, Stack or Linked List.
- The easiest way of implementing a queue is by using an Array.
- Initially the **head** (FRONT) and the **tail** (REAR) of the queue points at the first index of the array (starting the index of array from 0).
- As we add elements to the queue, the **tail** keeps on moving ahead, always pointing to the position where the next element will be inserted, while the **head** remains at the first index.



- When we remove an element from Queue, we can follow two possible approaches (mentioned [A] and [B] in above diagram).
- In [A] approach, we remove the element at **head** position, and then one by one shift all the other elements in forward position.
- In approach [B] we remove the element from **head** position and then move **head** to the next position.
- In approach [A] there is an **overhead of shifting the elements one position forward** every time we remove the first element.
- In approach [B] there is no such overhead, but whenever we move head one position ahead, after removal of first element, the **size on Queue is reduced by one space** each time.

Algorithm for ENQUEUE operation

1. Check if the queue is full or not.
2. If the queue is full, then print overflow error and exit the program.
3. If the queue is not full, then increment the tail and add the element.

Algorithm for DEQUEUE operation

1. Check if the queue is empty or not.
2. If the queue is empty, then print underflow error and exit the program.
3. If the queue is not empty, then print the element at the head and increment the head.

/* Below program is written in C++ language */

```
#include<iostream>

using namespace std;

#define SIZE 10

class Queue
{
   int a[SIZE];
   int rear;   //same as tail
   int front;  //same as head

   public:
   Queue()
   {
      rear = front = -1;
   }

   //declaring enqueue, dequeue and display functions
   void enqueue(int x);
   int dequeue();
   void display();
};

// function enqueue - to add data to queue
```

```
void Queue :: enqueue(int x)
{
    if(front == -1) {
        front++;
    }
    if( rear == SIZE-1)
    {
        cout << "Queue is full";
    }
    else
    {
        a[++rear] = x;
    }
}

// function dequeue - to remove data from queue
int Queue :: dequeue()
{
    return a[++front];  // following approach [B], explained above
}

// function to display the queue elements
void Queue :: display()
{
    int i;
    for( i = front; i <= rear; i++)
    {
        cout << a[i] << endl;
    }
}

// the main function
int main()
{
    Queue q;
    q.enqueue(10);
    q.enqueue(100);
    q.enqueue(1000);
    q.enqueue(1001);
    q.enqueue(1002);
```

```
    q.dequeue();
    q.enqueue(1003);
    q.dequeue();
    q.dequeue();
    q.enqueue(1004);

    q.display();

    return 0;
}
```

To implement approach [A], you simply need to change the dequeue method, and include a for loop which will shift all the remaining elements by one position.

```
return a[0];    //returning first element
for (i = 0; i < tail-1; i++)    //shifting all other elements
{
    a[i] = a[i+1];
    tail--;
}
```

Complexity Analysis of Queue Operations

Just like Stack, in case of a Queue too, we know exactly, on which position new element will be added and from where an element will be removed, hence both these operations requires a single step.

- Enqueue: **O(1)**
- Dequeue: **O(1)**
- Size: **O(1)**

Remarks on Queue Data Structure:

- Queue is used when things don't have to be processed immediately, but have to be processed in **F**irst **in First out** (**FIFO**) order like Breadth First Search.

- This property of Queue makes it also useful in following kind of scenarios.

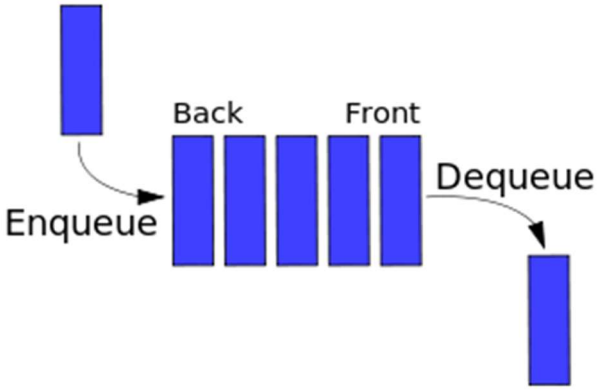
1) When a resource is shared among multiple consumers.
2) Examples include CPU scheduling, Disk Scheduling.
3) When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.

Queue (abstract data type)

Queue

Time complexity in big O notation

Algorithm	Average	Worst case
Space	O(*n*)	O(*n*)
Search	O(*n*)	O(*n*)
Insert	O(1)	O(1)
Delete	O(1)	O(1)



Representation of a FIFO (first in, first out) queue

- In computer science, a **queue** is a collection of entities that are maintained in a sequence and can be modified by the addition of entities at one end of the sequence and the removal of entities from the other end of the sequence.
- By convention, the end of the sequence at which elements are added is called the **back, tail, or rear** of the queue, and the end at which

elements are removed is called the **head or front** of the queue, analogously to the words used when people line up to wait for goods or services.

- The operation of adding an element to the rear of the queue is known as ***enqueue*****,** and the operation of removing an element from the front is known as ***dequeue*****.**
- Other operations may also be allowed, often including a *peek* or *front* operation that returns the value of the next element to be dequeued without dequeuing it.
- The operations of a queue make it a first-in-first-out (FIFO) data structure.
- In a FIFO data structure, the first element added to the queue will be the first one to be removed.
- This is equivalent to the requirement that once a new element is added, all elements that were added before have to be removed before the new element can be removed.
- A queue is an example of a linear data structure, or more abstractly a sequential collection.
- Queues are common in computer programs, where they are implemented as data structures coupled with access routines, as an abstract data structure or in object-oriented languages as classes. Common implementations are circular buffers and linked lists.
- Queues provide services in computer science, transport, and operations research where various entities such as data, objects, persons, or events are stored and held to be processed later.
- In these contexts, the queue performs the function of a buffer.
- Another usage of queues is in the implementation of breadth-first search.

Queue implementation

- Theoretically, one characteristic of a queue is that it does not have a specific capacity.
- Regardless of how many elements are already contained, a new element can always be added.
- It can also be empty, at which point removing an element will be impossible until a new element has been added again.
- Fixed-length arrays are limited in capacity, but it is not true that items need to be copied towards the head of the queue.

- The simple trick of turning the array into a closed circle and letting the head and tail drift around endlessly in that circle makes it unnecessary to ever move items stored in the array.
- If n is the size of the array, then computing indices modulo n will turn the array into a circle.
- This is still the conceptually simplest way to construct a queue in a high-level language, but it does admittedly slow things down a little, because the array indices must be compared to zero and the array size, which is comparable to the time taken to check whether an array index is out of bounds, which some languages do, but this will certainly be the method of choice for a quick and dirty implementation, or for any high-level language that does not have pointer syntax.
- The array size must be declared ahead of time, but some implementations simply double the declared array size when overflow occurs.
- Most modern languages with objects or pointers can implement or come with libraries for dynamic lists.
- Such data structures may have not specified a fixed capacity limit besides memory constraints.
- Queue *overflow* results from trying to add an element onto a full queue and queue *underflow* happens when trying to remove an element from an empty queue.

A *bounded queue* is a queue limited to a fixed number of items.

- There are several efficient implementations of FIFO queues.
- An efficient implementation is one that can perform the operations—en-queuing and de-queuing—in O(1) time.

- Linked list
  - A doubly linked list has O(1) insertion and deletion at both ends, so it is a natural choice for queues.
  - A regular singly linked list only has efficient insertion and deletion at one end.
  - However, a small modification—keeping a pointer to the *last* node in addition to the first one—will enable it to implement an efficient queue.
- A deque implemented using a modified dynamic array

Queues and programming languages

- Queues may be implemented as a separate data type, or maybe considered a special case of a double-ended queue (dequeue) and not implemented separately.
- For example, Perl and Ruby allow pushing and popping an array from both ends, so one can use **push** and **unshift** functions to enqueue and dequeue a list (or, in reverse, one can use **shift** and **pop**), although in some cases these operations are not efficient.
- C++'s Standard Template Library provides a "queue" templated class which is restricted to only push/pop operations.
- Since J2SE5.0, Java's library contains a Queue interface that specifies queue operations; implementing classes include LinkedList and (since J2SE 1.6) ArrayDeque. PHP has an SplQueue class and third party libraries like beanstalk'd and Gearman.

Examples

A simple queue implemented in JavaScript:

```
class Queue {
   constructor()
   {
      this.items = new Array(0)
   }
   enqueue(element)
   {
      this.items.push(element)
   }
   dequeue()
   {

      this.items.shift()
   }
}
```