<https://www.tutorialspoint.com/data_structures_algorithms/data_structures_algorithms_interview_questions.htm>

<https://www.tutorialspoint.com/data_structures_algorithms/dsa_quick_guide.htm>

<https://www.geeksforgeeks.org/commonly-asked-data-structure-interview-questions-set-1/>

**STACK**

## Basic Operations

* **Push**: Add an element to the top of a stack
* **Pop**: Remove an element from the top of a stack
* **IsEmpty**: Check if the stack is empty
* **IsFull**: Check if the stack is full
* **Peek**: Get the value of the top element without removing it



## Implementation

// Stack implementation in C++

#include <stdlib.h>

#include <iostream>

using namespace std;

#define MAX 10

int size = 0;

// Creating a stack

struct stack {

int items[MAX];

int top;

};

typedef struct stack st;

void createEmptyStack(st \*s) {

s->top = -1;

}

// Check if the stack is full

int isfull(st \*s) {

if (s->top == MAX - 1)

return 1;

else

return 0;

}

// Check if the stack is empty

int isempty(st \*s) {

if (s->top == -1)

return 1;

else

return 0;

}

// Add elements into stack

void push(st \*s, int newitem) {

if (isfull(s)) {

printf("STACK FULL");

} else {

s->top++;

s->items[s->top] = newitem;

}

size++;

}

// Remove element from stack

void pop(st \*s) {

if (isempty(s)) {

printf("\n STACK EMPTY \n");

} else {

printf("Item popped= %d", s->items[s->top]);

s->top--;

}

size--;

cout << endl;

}

// Print elements of stack

void printStack(st \*s) {

printf("Stack: ");

for (int i = 0; i < size; i++) {

cout << s->items[i] << " ";

}

cout << endl;

}

// Driver code

int main() {

int ch;

st \*s = (st \*)malloc(sizeof(st));

createEmptyStack(s);

push(s, 1);

push(s, 2);

push(s, 3);

push(s, 4);

printStack(s);

pop(s);

cout << "\nAfter popping out\n";

printStack(s);

}

**Time Complexity**

For the array-based implementation of a stack, the push and pop operations take constant time, i.e. O(1)

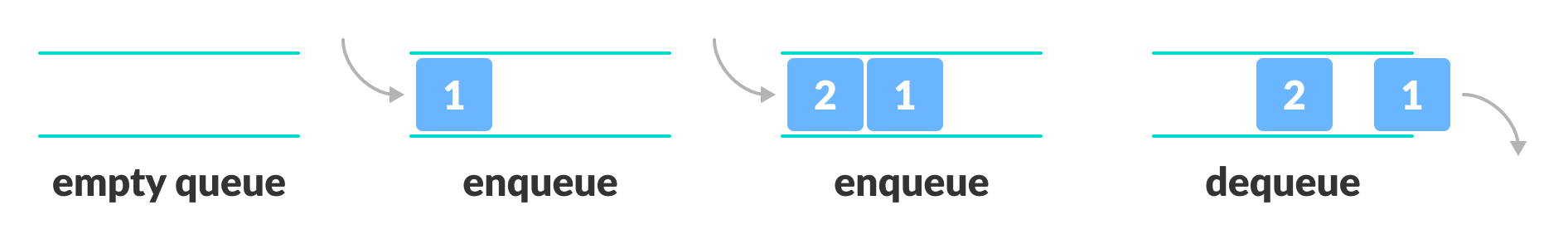
## Applications

* **To reverse a word** - Put all the letters in a stack and pop them out. Because of the LIFO order of stack, you will get the letters in reverse order.
* **In compilers** - Compilers use the stack to calculate the value of expressions like 2 + 4 / 5 \* (7 - 9) by converting the expression to prefix or postfix form.
* **In browsers** - The back button in a browser saves all the URLs you have visited previously in a stack. Each time you visit a new page, it is added on top of the stack. When you press the back button, the current URL is removed from the stack, and the previous URL is accessed.

**QUEUE**

## Basic 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



## Working

Queue operations work as follows:

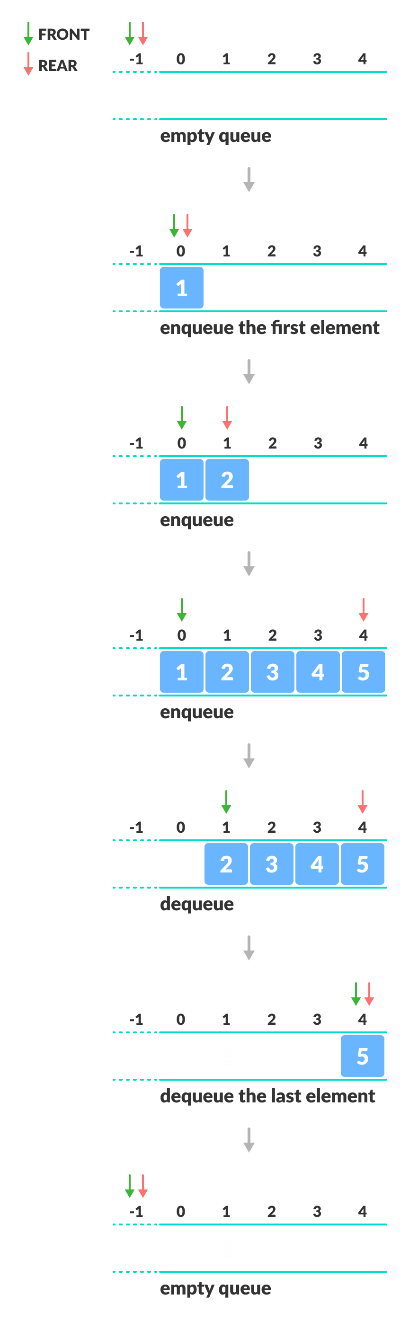
* two pointers FRONT and REAR
* FRONT track the first element of the queue
* REAR track the last element of the queue
* initially, set value of FRONT and REAR to -1

### Enqueue Operation

* check if the queue is full
* for the first element, set the value of FRONT to 0
* increase the REAR index by 1
* add the new element in the position pointed to by REAR

### Dequeue Operation

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



## Implementation

// Queue implementation in C++

#include <iostream>

#define SIZE 5

using namespace std;

class Queue {

private:

int items[SIZE], front, rear;

public:

Queue() {

front = -1;

rear = -1;

}

bool isFull() {

if (front == 0 && rear == SIZE - 1) {

return true;

}

return false;

}

bool isEmpty() {

if (front == -1)

return true;

else

return false;

}

void enQueue(int element) {

if (isFull()) {

cout << "Queue is full";

} else {

if (front == -1) front = 0;

rear++;

items[rear] = element;

cout << endl

<< "Inserted " << element << endl;

}

}

int deQueue() {

int element;

if (isEmpty()) {

cout << "Queue is empty" << endl;

return (-1);

} else {

element = items[front];

if (front >= rear) {

front = -1;

rear = -1;

} /\* Q has only one element, so we reset the queue after deleting it. \*/

else {

front++;

}

cout << endl

<< "Deleted -> " << element << endl;

return (element);

}

}

void display() {

/\* Function to display elements of Queue \*/

int i;

if (isEmpty()) {

cout << endl

<< "Empty Queue" << endl;

} else {

cout << endl

<< "Front index-> " << front;

cout << endl

<< "Items -> ";

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

cout << items[i] << " ";

cout << endl

<< "Rear index-> " << rear << endl;

}

}

};

int main() {

Queue q;

//deQueue is not possible on empty queue

q.deQueue();

//enQueue 5 elements

q.enQueue(1);

q.enQueue(2);

q.enQueue(3);

q.enQueue(4);

q.enQueue(5);

// 6th element can't be added to because the queue is full

q.enQueue(6);

q.display();

//deQueue removes element entered first i.e. 1

q.deQueue();

//Now we have just 4 elements

q.display();

return 0;

}

## Complexity Analysis

The complexity of enqueue and dequeue operations in a queue using an array is O(1).

## Applications of Queue

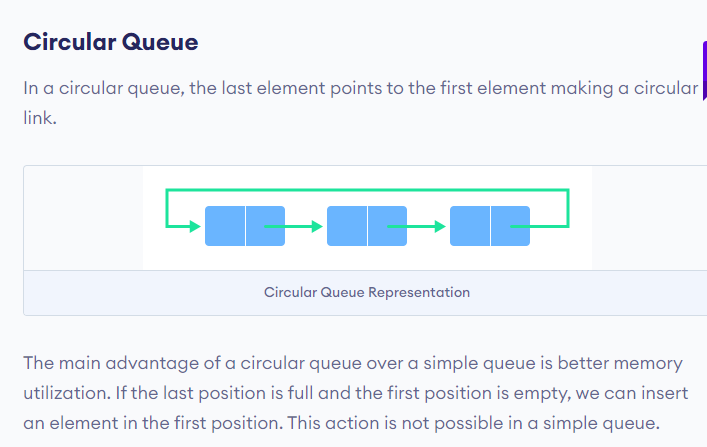
* CPU scheduling, Disk Scheduling
* When data is transferred asynchronously between two processes.The queue is used for synchronization. For example: IO Buffers, pipes, file IO, etc
* Handling of interrupts in real-time systems.
* Call Center phone systems use Queues to hold people calling them in order.

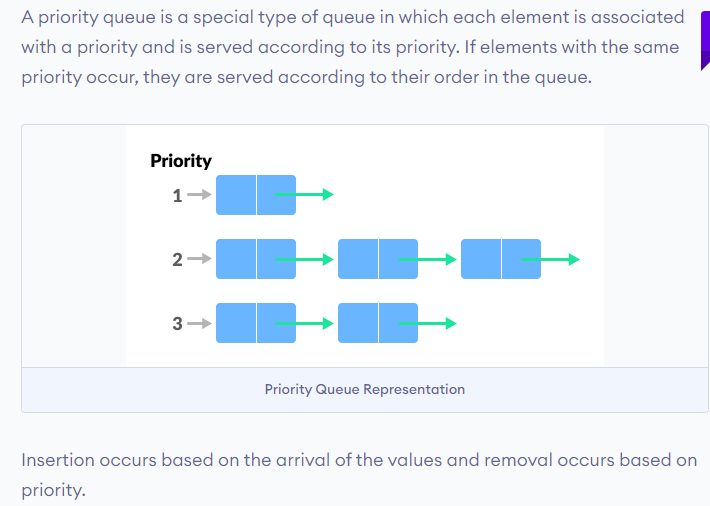
# Types of Queues

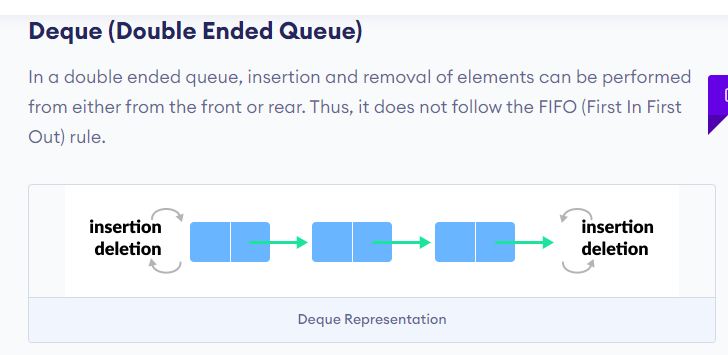
A [queue](https://www.programiz.com/dsa/queue) is a useful data structure in programming. It is similar to the ticket queue outside a cinema hall, where the first person entering the queue is the first person who gets the ticket.

There are four different types of queues:

* Simple Queue
* Circular Queue
* Priority Queue
* Double Ended Queue







A **binary tree** can have a **minimum** of zero **nodes**, which occurs when the **nodes** have NULL values. Furthermore, a **binary tree** can also have 1 or 2 **nodes**.

The **maximum number of nodes** at level 'l' of a **binary tree** is 2l. Here level is the **number of nodes** on the path from the root to the **node** (including root and **node**).

An **in-place** sorting **algorithm** sorts the elements **in place**: that is, it needs only O(1) extra space. An **out**-of-**place** sorting **algorithm** needs extra space to put the elements in as it's sorting them. Usually this means O(n) extra space.

[**https://www.geeksforgeeks.org/real-time-application-of-data-structures/**](https://www.geeksforgeeks.org/real-time-application-of-data-structures/)

[**https://www.programiz.com/dsa/divide-and-conquer**](https://www.programiz.com/dsa/divide-and-conquer)

**Quicksort** is one of the most efficient sorting algorithms, and this makes of it one of the most used as well. The first thing to do is to select a pivot number, this number will separate the data, on its left are the numbers smaller than it and the greater numbers on the right

The time complexity of **Quicksort** is O(n log n) in the best **case**, O(n log n) in the average **case**, and O(n^2) in the worst **case**. But because it has the best performance in the average **case** for most inputs, **Quicksort** is generally considered the “fastest” sorting algorithm