DSA Implementation in Javascript(Basic)

Algorithms - Set of rules that need to be followed

Time complexity -

- Big O (Worst case)
- Theta (Best case)
- Omega(Average case)

Big O is expressed in terms of input size and it is the most important notation, some of the common complexities are O(1) constant input size, O(n) Linear input size, O(n^2) Quadratic input size, O(n^3) Cubic input size, O(log n) Logarithmic input size, O(n log n), O(2^n) Exponential, O(n!) Factorial

Space complexity - eg O(1) constant no extra array or space is used

Array and Objects Operations —

Objects (Collection of key-value pairs)

- Insert - O(1)
- Remove - O(1)
- Access - O(1)
- Search -O(n)
- Object.keys() -O(n)
- Object.values() -O(n)
- Object.entries() -O(n)

Arrays (Ordered collection of values)

- Insert/remove at the end - O(1)
- Insert/remove at the beginning - O(n)
- Access - O(1)
- Searching -O(n)
- Push /Pop - O(1)
- Shift /unshift /concat /slice/splice - O(n)

Math Algorithms

Fibonacci sequence - O(n)

```
function fibonacci(n)
{
let t1 = 0;
let t2 = 1;

for(let i=2;i<=n;i++)
{
    let t3 = t1+t2;
    t1=t2;
    t2=t3;
}
return t1

}

console.log(fibonacci(2))
console.log(fibonacci(3))
console.log(fibonacci(7))
```

```
output :-
1
1
8
```

```
function fibonacci(n)
{
const fib =[0,1];
for(let i=2;i<n;i++)
{
    fib[i]=fib[i-1]+fib[i-2];

}
return fib

}

console.log(fibonacci(2))
console.log(fibonacci(3))
console.log(fibonacci(7))

output:-
[ 0, 1 ]
[ 0, 1, 1 ]
[
  0, 1, 1, 2,
  3, 5, 8
]
```

Factorial of a number - O(n)

```
function factorial(n)
{
let result =1
for(let i=2;i<=n;i++)
{
    result = result*i;

}
return result

}

console.log(factorial(2))
console.log(factorial(5))
console.log(factorial(7))

output:-
2
120
5040
```

Prime Number -O(n)

```
function isPrime(n)
{
    if(n<2)
    {
        return false
    }
    for(let i=2;i<n;i++)
    {
        if(n%i==0)
        {
            return false
        }
    }
    return true
}

console.log(isPrime(5))
```

Optimized solution O(sqrt(n))

Integers larger than square root do not need to be checked because, whenever n=a*b , one of the two factors ‘a’ and ‘b’ is less than or equal to the square root of ‘n’

```
function isPrime(n)
{
    if(n<2)
    {
        return false
    }
    for(let i=2;i<=Math.sqrt(n);i++)
    {
        if(n%i==0)
        {
            return false
        }
    }
    return true
}

console.log(isPrime(5))
```

Power of Two - O(logn)

```
function isPowerOFTwo(n)
{
    if(n<1)
    {
        return false
    }
    while(n>1)
    {
     if(n%2!=0)
     {
         return false
     }
     n=Math.floor(n/2)
    }
    return true;
}

console.log(isPowerOFTwo(8))
```

```
function isPowerOFTwoBitWise(n)
{
    if(n<1)
    {
        return false
    }
  return (n & (n-1)===0)
}

console.log(isPowerOFTwoBitWise(8))
```

Recursion

Recursion is a problem-solving technique where the solution depends on solutions to smaller instances of the same problem

when a function calls itself

Fibonacci Sequence Recursive (2^n)

```
function recursiveFibonacci(n){
    if(n<2)
    {
        return n
    }

    return recursiveFibonacci(n-1)+ recursiveFibonacci(n-2);
}

console.log(recursiveFibonacci(0))
console.log(recursiveFibonacci(1))
console.log(recursiveFibonacci(6))

output:-
0
1
8
```

Recursive factorial of a number (o(n))

```
function recursiveFactorial(n){
if(n==0)
{
    return 1;
}
    return n*recursiveFactorial(n-1);
}
```

```
console.log(recursiveFactorial(0))
console.log(recursiveFactorial(1))
console.log(recursiveFactorial(6))

output:-
1
1
720
```

Search Algorithms

Linear Search (o(n))

```
function linearSearch(arr,target)
{
    for(let i=0;i<arr.length;i++)
    {
        if(arr[i]===target)
        {
            return i
        }
    }
    return -1
}

console.log(linearSearch([-5,2,10,4,6],10))
console.log(linearSearch([-5,2,10,4,6],6))
console.log(linearSearch([-5,2,10,4,6],20))

output:-
2
```

```
4
-1
```

Binary search o(log n)

Binary search only works on sorted array

If the array is empty , return -1 as the element cannot be found

If the array has elements, find the middle element in the array. If target is equal to the middle element, return the middle element index.

If target is less than the middle element, binary search left half of the array.

if target is greater than middle element, binary search right half of the array.

```
function binarySearch(arr,target)
{
let leftIndex=0;
let rightIndex=arr.length-1;
while(leftIndex<=rightIndex)
{
    let midIndex = Math.floor((leftIndex+rightIndex)/2);
    if(target==arr[midIndex])
    {
    return midIndex;
    }

    if(target<arr[midIndex])
    {
        rightIndex = midIndex-1;
    }else
    {
        leftIndex=midIndex+1;
    }

}
return -1
}

console.log(binarySearch([-5,2,4,6,10],10))
output:-
4
```

Recursive binary search o(log n)

```

function recursiveBinarySearch(arr,target)
{
    return search(arr,target,0,arr.length-1)

}

function search(arr,target,leftIndex,rightIndex)
{
    if(leftIndex>rightIndex)
    {
        return -1
    }

    let midIndex = Math.floor((leftIndex+rightIndex)/2);
    if(target===arr[midIndex])
    {
        return midIndex;
    }
    if(target<arr[midIndex])
    {
        return search(arr,target,leftIndex,midIndex-1)
    }else
    {
        return search(arr,target,midIndex+1,rightIndex)
    }
}

console.log(recursiveBinarySearch([-5,2,4,6,10],10))

output:-
4
```

Sorting

Bubble Sort o(n^2)

```
function bubbleSort(arr)
{
    let swapped

do{
    swapped = false

 for(let i=0;i<arr.length-1;i++)
 {
     if(arr[i]>arr[i+1])
     {
         let temp = arr[i];
         arr[i]=arr[i+1];
         arr[i+1]=temp;
         swapped=true;
     }
 }
}while(swapped)
}
const arr=[8,20,-2,4,-6]
bubbleSort(arr)
console.log(arr);
output:-
[ -6, -2, 4, 8, 20 ]
```

Insertion Sort (o(n^2))

In Insertion sort, we virtually split the array into sorted and unsorted part

Assume that first element is already sorted and remaining elements are unsorted

Select an unsorted element and compare with all elements in the sorted part

If the elements in the sorted part is smaller than the selected elements, proceed to the next element in the unsorted part else shift larger elements in the sorted part towards right.

Insert the selected element at the right index

Repeat till all the unsorted elements are placed in the right order

```
function insertionSort(arr)
{
    for(let i=1;i<arr.length;i++)
    {
        let numberToInsert=arr[i];
        let j=i-1;
        while(j>=0 && arr[j]>numberToInsert)
        {
            arr[j+1]=arr[j]
            j=j-1
        }
        arr[j+1]=numberToInsert
    }
}
```

```
const arr=[8,20,-2,4,-6]
insertionSort(arr)
console.log(arr)
output:-
[ -6, -2, 4, 8, 20 ]
```

Selection Sort o(n^2)

In Selection sort we divide the array in sorted and unsorted part

we select a number and then look for a number smaller than selected number

once we find it we swap them with each other and repeat the process

```
function selectionSort(arr)
{

   for(let i=0;i<arr.length;i++)
   {
       let min=i;
       for(let j=i+1;j<arr.length;j++)
       {
           if(arr[min]>arr[j])
           {
               min=j;
           }
       }
       let temp = arr[i];
       arr[i]=arr[min];
       arr[min]=temp

   }
}

const arr=[8,20,-2,4,-6]
```

```
selectionSort(arr)
console.log(arr)
output:-
[ -6, -2, 4, 8, 20 ]
```

Quick Sort

worst case -O(n^2) when array is already sorted

average case - O(nlogn)

Identify the pivot element in the array it can be any element

Put everything that's smaller than the pivot into left array and everything that's greater than pivot into right array

Repeat the process for the individual left and right arrays till you have an array of length 1 which is sorted by definition

Repeatedly concatenate the left array , pivot and right array till one sorted array remains

Using last index as pivot

```
function quickSort(arr) {
  if (arr.length <= 1) {
    return arr; // Base case: arrays with 0 or 1 elements are al
  }

  const pivot = arr[arr.length - 1]; // Choose the last element
  const left = []; // Elements less than the pivot
  const right = []; // Elements greater than the pivot

  for (let i = 0; i < arr.length - 1; i++) { // Exclude the piv
    if (arr[i] < pivot) {
      left.push(arr[i]);
    } else {
      right.push(arr[i]);
    }
  }

  // Recursively sort the left and right subarrays and combine w
  return [...quickSort(left), pivot, ...quickSort(right)];
}

// Example usage:
const array = [8, 3, 1, 7, 0, 10, 2];
console.log(quickSort(array)); // Output: [0, 1, 2, 3, 7, 8, 10]
```

Using first index as pivot and in place sorting (without using extra space)

```

function partition(array,low,high)
{
    let pivot = array[low];
    let start=low;
    let end=high;
    while(start<end)
    {
        while(array[start]<=pivot)
        {
            start++;
        }

        while(array[end]>pivot)
        {
            end--;
        }
        if(start<=end)
        {
            [array[start],array[end]]=[array[end],array[start]]
        }
    }

    [array[low],array[end]]=[array[end],array[low]];


    return end;
}

function quickSort(array,start,end)
{
    if(start<end)
    {
        let idx = partition(array,start,end);
        quickSort(array,start,idx-1);
        quickSort(array,idx+1,end);
    }
}

let array=[7,6,10,5,9,2,1,15,7];
quickSort(array,0,array.length-1);

console.log(array);
```

Merge Sort O(nlogn)

Divide the array into sub arrays , each containing only one element

Repeatedly merge the sub arrays to produce new sorted sub arrays until there is only one sub array remaining. That will be the sorted array.

```
function mergeSort(arr)
{
    //base condition
    if(arr.length<2)
    {
        return arr
    }
    const mid = Math.floor(arr.length/2)
    const leftArr = arr.slice(0,mid);
    const rightArr = arr.slice(mid)

    return merge(mergeSort(leftArr),mergeSort(rightArr))

}
function merge(leftArr,rightArr){
    const sortedArr=[];
    while(leftArr.length && rightArr.length)
    {
        if(leftArr[0]<=rightArr[0])
        {
            sortedArr.push(leftArr.shift())
        }else
        {
            sortedArr.push(rightArr.shift())
        }
    }
  return  [...sortedArr,...leftArr,...rightArr]
}


const arr = [8,20,-2,4,-6]
console.log(mergeSort(arr))

output:-
[ -6, -2, 4, 8, 20 ]
```

```
function merge(L,R,A)
{
    let nL = L.length;
    let nR=R.length;
    let i=0;
    let j=0;
    let k=0;

    while(i<nL && j<nR)
    {
        if(L[i]<=R[j])
        {

            A[k]=L[i];
            i++;
        }else
        {
            A[k]=R[j];
            j++;
        }
        k++;
    }

    while(i<nL)
    {
        A[k]=L[i];
        i++;
        k++;
    }
    while(j<nR)
    {
        A[k]=R[j];
        j++;
        k++;
    }


}
```

```
function mergeSort(Arr)
{
let n = Arr.length;
if(n<2)
{
    return;
}
let mid = Math.floor(n / 2);
let left = [];
let right = [];


  for (let i = 0; i < mid; i++) {
        left.push(Arr[i]);
    }

    for (let i = mid; i < n; i++) {
        right.push(Arr[i]);
    }




mergeSort(left);
mergeSort(right);
merge(left,right,Arr);

}

let array = [2,4,1,6,8,5,3,7,10,11,9];
mergeSort(array);
console.log(array);
```

Data Structure - A data structure is a way to store and organize data so that it can be used efficiently

DOM - Tree data structure, Browser back and forward - Stack data structure, OS job scheduling - Queue Data structure

Arrays -

An array is a data structure that can hold a collection of values

Arrays can contain a mix of different data types. You can store strings, boolean, numbers or even objects all in the same array

Arrays are resizable, you don't have to declare the size of an array before creating it in js

```
// Creating an array
let fruits = ["apple", "banana", "cherry"];

// Accessing elements using indices
console.log(fruits[0]); // Output: "apple"
console.log(fruits[1]); // Output: "banana"
console.log(fruits[2]); // Output: "cherry"

// Adding an element to the end of the array
fruits.push("orange");
console.log(fruits); // Output: ["apple", "banana", "cherry", "
// Removing the last element from the array
fruits.pop();
console.log(fruits); // Output: ["apple", "banana", "cherry"]

// Adding an element to the beginning of the array
fruits.unshift("kiwi");
console.log(fruits); // Output: ["kiwi", "apple", "banana", "che
```

```
// Removing the first element from the array
fruits.shift();
console.log(fruits); // Output: ["apple", "banana", "cherry"]

// Changing an element in the array
fruits[1] = "blueberry";
console.log(fruits); // Output: ["apple", "blueberry", "cherry"

// Getting the length of the array
console.log(fruits.length); // Output: 3
```

Inserting Elements into an Array Without Built-in Functions

Insert at the Start:

To insert an element at the start of an array, we need to shift all elements to the right by one position to make space for the new element.

```
function insertAtStart(arr, element) {
    // Shift all elements to the right by one position
    for (let i = arr.length; i > 0; i--) {
        arr[i] = arr[i - 1];
    }
    // Insert the new element at the first position
    arr[0] = element;
}

let numbers = [1, 2, 3, 4];
insertAtStart(numbers, 0);
console.log(numbers); // Output: [0, 1, 2, 3, 4]
```

Insert at the End:

Inserting an element at the end of an array is straightforward, just place the element at the next available index.

```
function insertAtEnd(arr, element) {
    // Insert the new element at the end of the array
    arr[arr.length] = element;
}

let numbers = [1, 2, 3, 4];
insertAtEnd(numbers, 5);
console.log(numbers); // Output: [1, 2, 3, 4, 5]
```

Insert in the Middle:

To insert an element in the middle of an array, you need to shift all elements from the insertion point onwards to the right by one position.

```
function insertAtMiddle(arr, index, element) {
    // Shift all elements from index onwards to the right by one
    for (let i = arr.length; i > index; i--) {
        arr[i] = arr[i - 1];
    }
    // Insert the new element at the specified index
    arr[index] = element;
}

let numbers = [1, 2, 3, 4];
insertAtMiddle(numbers, 2, 99);
console.log(numbers); // Output: [1, 2, 99, 3, 4]
```

CRUD Operations on an Array Without Built-in Functions

Let's now look at the basic CRUD operations: **Create**, **Read**, **Update**, and **Delete**.

Create: (Add a new element at the end of the array)

```
function create(arr, element) {
    // Add the new element at the end of the array
    arr[arr.length] = element;
}

let numbers = [1, 2, 3];
create(numbers, 4);
console.log(numbers); // Output: [1, 2, 3, 4]
```

Read: (Access an element by index)

```
function read(arr, index) {
    // Check if the index is within the bounds of the array
    if (index >= 0 && index < arr.length) {
        return arr[index];
    } else {
        return undefined; // Return undefined if the index is ou
    }
}

let numbers = [1, 2, 3, 4];
console.log(read(numbers, 2)); // Output: 3
console.log(read(numbers, 5)); // Output: undefined
```

Update: (Change an element at a specific index)

```
function update(arr, index, newElement) {
    // Check if the index is within the bounds of the array
    if (index >= 0 && index < arr.length) {
        arr[index] = newElement;
    }
}

let numbers = [1, 2, 3, 4];
update(numbers, 2, 99);
console.log(numbers); // Output: [1, 2, 99, 4]
```

Delete: (Remove an element at a specific index)

To delete an element, you need to shift all elements to the left, starting from the element after the index.

```
function deleteElement(arr, index) {
    // Check if the index is within the bounds of the array
    if (index >= 0 && index < arr.length) {
        // Shift all elements after the index to the left
        for (let i = index; i < arr.length - 1; i++) {
            arr[i] = arr[i + 1];
        }
        // Decrease the length of the array by 1
        arr.length = arr.length - 1;
    }
}

let numbers = [1, 2, 3, 4];
deleteElement(numbers, 2);
console.log(numbers); // Output: [1, 2, 4]
```

Objects

An object is an unordered collection of key-value pairs. The key must either be a string or symbol data type whereas the value can be of any data type

```
const obj ={
    name:"Heisenberg",
    age:25,
    "key-three":true,
    sayMyName:function(){
        console.log(this.name)
    }
}

obj.hobby ='football'
delete obj.hobby

console.log(obj.name);
console.log(obj['age']);
console.log(obj['key-three']);
console.log(obj);
obj.sayMyName()

output:
Heisenberg
25
true
{
  name: 'Heisenberg',
  age: 25,
  'key-three': true,
  sayMyName: [Function: sayMyName]
}
Heisenberg
```

Set

A set is a data structure that can hold a collection of values. The values however must be unique sets are dynamic, does not maintain insertion order

common methods set.add(), set.has(), set.size()

Map

A Map is an unordered collection of key-value pairs. Both keys and values can be of any data types to retrieve a value, you can use the corresponding key

common methods map.set(), map.delete(), map.has(), map.clear()

Object vs Map

- Objects are unordered whereas maps are ordered.
- Keys in objects can only be string or symbol type whereas in maps, they can be of any type
- An object has a prototype and may contain a few default keys which may collide with your own keys if your not careful. A map on the other hand does not contain any keys by default
- Objects are not iterables whereas maps are iterables
- The number of items in an object must be determined manually where as it is readily available with the size property in a map
- Apart from storing data, you can attach functionality to an object whereas maps are restricted to just storing data

Stack

The stack data structure is a sequential collection of elements that follows the principle of Last in First Out (LIFO). The last element inserted into the stack is the first element to be removed .Stack is used in Browser history tracking, Undo operation when typing, Expression conversions, Call stack in Javascript runtime

```
class Stack {
  constructor() {
    this.items = [];
  }

  // Push an item onto the stack
  push(element) {
    this.items.push(element);
  }

  // Pop an item off the stack
  pop() {
    if (this.isEmpty()) {
      return "Stack is empty";
    }
    return this.items.pop();
  }

  // Peek at the top item of the stack
  peek() {
    if (this.isEmpty()) {
      return "Stack is empty";
    }
    return this.items[this.items.length - 1];
  }

  // Check if the stack is empty
  isEmpty() {
    return this.items.length === 0;
  }

  // Get the size of the stack
  size() {
    return this.items.length;
  }
  // Print the stack
  printStack() {
    console.log(this.items.toString());
  }
}

// Example usage
const stack = new Stack();
stack.push(10);
stack.push(20);
stack.push(30);
stack.printStack(); // Output: 10,20,30

console.log(stack.pop()); // Output: 30
console.log(stack.peek()); // Output: 20
console.log(stack.size()); // Output: 2
stack.printStack(); // Output: 10,20
```

without using in-build methods

```
class Stack {
  constructor() {
    this.items = {}; // Object to store stack elements
    this.count = 0; // Counter to keep track of the stack size
  }

  // Push an item onto the stack
  push(element) {
    this.items[this.count] = element; // Add the element at the
    this.count++; // Increment the count
  }

  // Pop an item off the stack
  pop() {
    if (this.isEmpty()) {
      return "Stack is empty"; // Check for underflow
    }
    this.count--; // Decrement the count
    const element = this.items[this.count]; // Get the top eleme
    delete this.items[this.count]; // Remove the element from th
    return element; // Return the popped element
  }

  // Peek at the top item of the stack
  peek() {
    if (this.isEmpty()) {
      return "Stack is empty"; // Check if the stack is empty
    }
    return this.items[this.count - 1]; // Return the top element
  }

  // Check if the stack is empty
  isEmpty() {
    return this.count === 0; // Check if the count is zero
  }

  // Get the size of the stack
  size() {
    return this.count; // Return the count
  }

  // Print the stack
  printStack() {
    if (this.isEmpty()) {
      console.log("Stack is empty");
      return;
    }

    let stackString = "";
    for (let i = 0; i < this.count; i++) {
      stackString += this.items[i] + " ";
    }
    console.log(stackString.trim()); // Print all elements
  }
}

// Example usage
const stack = new Stack();
stack.push(10);
stack.push(20);
stack.push(30);
stack.printStack(); // Output: 10 20 30

console.log(stack.pop()); // Output: 30
console.log(stack.peek()); // Output: 20
console.log(stack.size()); // Output: 2
stack.printStack(); // Output: 10 20
```

Queues

The queue data structure is a sequential collection of elements that follows the principle of First In First Out (FIFO)

The first element inserted into the queue is first element to be removed

common methods - Enqueue, Dequeue

```
class Queue{
    constructor(){
        this.item=[]
    }

    enqueue(element)
    {
        // add the element
        this.item.push(element)
    }

    dequeue()
    {
        if(this.isEmpty())
        {
            console.log("Queue is empty")
        }
        return this.item.shift()
    }

    isEmpty()
    {
        return this.item.length==0
    }

    peek(){
        if(!this.isEmpty())
        {
            return this.item[0]
        }
        return null
    }

    size()
    {
        return this.item.length
    }

    print()
    {
        console.log(this.item.toString())
    }
```

```
}

const queue = new Queue()
console.log(queue.isEmpty())

queue.enqueue(10)
queue.enqueue(20)
queue.enqueue(30)

queue.print()
console.log(queue.size())
console.log(queue.peek())
queue.dequeue()
queue.print()

output:
true
10,20,30
3
10
20,30
```

Optimized version

```
class Queue {
  constructor() {
    this.items = {}; // Object to store queue elements
    this.front = 0;  // Index of the front of the queue
    this.rear = 0;   // Index of the rear of the queue
  }

  // Enqueue an item to the end of the queue
  enqueue(element) {
    this.items[this.rear] = element; // Add the element at the
    this.rear++; // Increment the rear index
  }

  // Dequeue an item from the front of the queue
  dequeue() {
    if (this.isEmpty()) {
      return "Queue is empty"; // Check for underflow
    }
    const element = this.items[this.front]; // Get the front ele
    delete this.items[this.front]; // Remove the front element
    this.front++; // Increment the front index
    return element; // Return the dequeued element
  }

  // Peek at the front item of the queue
  peek() {
    if (this.isEmpty()) {
      return "Queue is empty"; // Check if the queue is empty
    }
    return this.items[this.front]; // Return the front element w
  }

  // Check if the queue is empty
  isEmpty() {
    return this.front === this.rear; // Check if the front index
  }

  // Get the size of the queue
  size() {
    return this.rear - this.front; // Calculate the size based o
  }

  // Print the queue
  printQueue() {
    if (this.isEmpty()) {
      console.log("Queue is empty");
      return;
    }

    let queueString = "";
    for (let i = this.front; i < this.rear; i++) {
      queueString += this.items[i] + " ";
    }
    console.log(queueString.trim()); // Print all elements
  }
}

// Example usage
const queue = new Queue();
queue.enqueue(10);
queue.enqueue(20);
queue.enqueue(30);
queue.printQueue(); // Output: 10 20 30

console.log(queue.dequeue()); // Output: 10
console.log(queue.peek()); // Output: 20
console.log(queue.size()); // Output: 2
queue.printQueue(); // Output: 20 30
```

Circular Queue

The size of the queue is fixed and a single block of memory is used as if the first element is connected to the last element

A circular queue will reuse the empty block created during dequeue operation

```
class CircularQueue {
  constructor(capacity) {
    this.items = new Array(capacity);
    this.rear = -1;
    this.front = -1;
    this.currentLength = 0;
    this.capacity = capacity;
  }

  isFull() {
    return this.currentLength === this.capacity;
  }

  isEmpty() {
    return this.currentLength === 0;
  }

  size() {
    return this.currentLength;
  }

  enqueue(item) {
    if (!this.isFull()) {
      this.rear = (this.rear + 1) % this.capacity;
      this.items[this.rear] = item;
      this.currentLength += 1;
      if (this.front === -1) {
        this.front = this.rear;
      }
    }
  }

  dequeue() {
    if (this.isEmpty()) {
      return null;
    }
    const item = this.items[this.front];
    this.items[this.front] = null;
    this.front = (this.front + 1) % this.capacity;
    this.currentLength -= 1;
    if (this.isEmpty()) {
      this.front = -1;
      this.rear = -1;
    }
    return item;
  }

  peek() {
    if (!this.isEmpty()) {
      return this.items[this.front];
    }
    return null;
  }

  print() {
    if (this.isEmpty()) {
      console.log("Queue is empty");
    } else {
      let i;
      let str = "";
      for (i = this.front; i !== this.rear; i = (i + 1) % this.c
        str += this.items[i] + " ";
      }
      str += this.items[i];
      console.log(str);
    }
  }
}

const queue = new CircularQueue(5);
console.log(queue.isEmpty());
queue.enqueue(10);
queue.enqueue(20);
queue.enqueue(30);
queue.enqueue(40);
queue.enqueue(50);
```

```
console.log(queue.size());
queue.print();
console.log(queue.isFull());
console.log(queue.dequeue());
console.log(queue.peek());
queue.print();
queue.enqueue(60);
queue.print();
```