Name	Big O Notation
Constant	O(1)
Logarithmic	O(log(n))
Linear	O(n)
Linear Logarithmic	O(n*log(n))
Polynomial	$O(n^m)$
Exponential	$O(m^n)$
Factorial	O(n!)

```
//? #1.)-------------------------------------------------------------
function myFunction( n ) {
    return n * 2 + 1;
}
/*
*Answer:
!Constant: O( 1 )
*/
//? #2.)-------------------------------------------------------------
function myFunction( n ) {
    for ( let i = 1; i <= 100; i++ ) {
        console.log( i );
    }
}
/*
*Answer:
!O( 1 ): Constant;
!Why ? Because there are a fixed number of loops in the for loop.
!The for loop is not dependent upon n.
*/
```

```
//? #3.)-----------------------------------------------------------------
function myFunction( n ) {
    if ( n <= 1 ) return;
    myFunction( n / 2 );
}
/*
*Answer:
!Logarithmic: O( log( n ) ): because the recursion will half the size of n each time
*/
//? #4.)-----------------------------------------------------------------
function myFunction( n ) {
    for ( let i = 1; i <= n; i++ ) {
        console.log( i );
    }
}
/*
*Answer:
!Linear : O(n) : Because the for loop will iterate n times
*/
```

```
*/
//? #5.)-------------------------------------------------------------
function myFunction( n ) {
    if ( n === 1 ) return;
    myFunction( n - 1 )
}
/*
*Answer:
!Linear: O( n ): Because the recursive call will be made n times.
*/
//? #6.)-------------------------------------------------------------
function myFunction( n ) {
    if ( n <= 1 ) return;
    for ( let i = 1; i <= n; i++ ) {
        console.log( i );
    }
    myFunction( n / 2 );
    myFunction( n / 2 );
}
/*
*Answer:
!O(n * log(n)) : LogLinear because the for loop will run n times;
! then the myFunction will be called recursively 2(log(n)) so we can drop the constant and consider log(n).
!Combining the for loop time complexity of O(n) and the recursive calls of log(n) we have O(n*log(n));
*/
```

```
//? #7.)-------------------------------------------------------------------
function myFunction( n ) {
    for ( let i = 1; i <= n; i++ ) {
        for ( let j = 1; j <= n; j++ ) {
            for ( let k = 1; k <= n; k++ ) {}
        }
    }
}
/*
*Answer:
! Polynomial O(n^3) : Because the outer for (i) will run n times,
! and for each time the outer loop runs the inner (j) will run n times
!- which is n * n - and there's another inner loop (k) that will run n times
!- so we have n*n*n which is n3 so we have O(n3)
*/
```

```
//? #8.)-------------------------------------------------------------
function myFunction( n ) {
    if ( n === 1 ) return;
    myFunction( n - 1 );
    myFunction( n - 1 );
}
/*
*Answer:
!Exponential: O( 2^n ):
! Because each call will make two more recursive calls so they will run a total of 2n times, or O( 2n )
*/
//? #9.)-------------------------------------------------------------
function myFunction( n ) {
    if ( n === 0 ) return;
    myFunction( n - 1 );
    myFunction( n - 1 );
    myFunction( n - 1 );
}
/*
*Answer:
!Exponential: O( 3^n ): Because each call will make 3 more recursive calls, n times.
*/
```

```
//? #10.)-------------------------------------------------------------
function myFunction( n ) {
    if ( n === 1 ) return;
    for ( let i = 1; i <= n; i++ ) {
        myFunction( n - 1 );
    }
}
/*
*Answer:
! Factorial : O(n!) : The code is recursive,
!but the number of recursive calls made in a a single stack frame depends on the input.
! So in this case, the for loop executes n times;
! but within the for loop the recursive call is made n-1 times.
!So you have n * n-1 * n-2 * n-3 ... or O(n!);
*/
```

```
//*Identify the type of sort, the time complexity, and the space complexity of the following code:
function swap( array, idx1, idx2 ) {
    [ array[ idx1 ], array[ idx2 ] ] = [ array[ idx2 ], array[ idx1 ] ]
}

function whichSort( array ) {
    let swapped = true;

    while ( swapped ) {
        swapped = false;

        for ( let i = 0; i < array.length; i++ ) {
            if ( array[ i ] > array[ i + 1 ] ) {
                swap( array, i, i + 1 );
                swapped = true;
            }
        }
    }
}
/*
!Bubble Sort.Time complexity O( n^2 );
!Space Complexity of O( 1 )
*/
```

```
//*Identify the type of sort as well as the time and space complexity of the following code:
function swap( arr, index1, index2 ) {
    [ arr[ index1 ], arr[ index2 ] ] = arr[ index2 ], arr[ index1 ] ];
}

function whichSort( list ) {
    for ( let i = 0; i < list.length; i++ ) {
        let min = i;

        for ( let j = i + 1; j < list.length; j++ ) {
            if ( list[ j ] < list[ min ] ) {
                min = j;
            }
        }
        if ( min !== i ) {
            swap( list, i, min );
        }
    }
}
/*
!Selection Sort: Time complexity O( n^2 ) and O( 1 ) space complexity
```

```
//*Identify the type of sort, as well as the time and space complexity of the following code:
function mySort( list ) {
    for ( let i = 1; i < list.length; i++ ) {
        value = list[ i ];
        hole = i;

        while ( hole > 0 && list[ hole - 1 ] > value ) {
            list[ hole ] = list[ hole - 1 ];
            hole--;
        }
        list[ hole ] = value;
    }
}
/*
!Insertion Sort.Time Complexity: O( n^2 ), Space complexity: O( 1 )
*/
```

```
function doSomething( array1, array2 ) {
    let result = []
    while ( array1.length && array2.length ) {
        if ( array1[ 0 ] < array2[ 0 ] ) {
            result.push( array1.shift() );
        } else {
            result.push( array2.shift() );
        }
    }
    return [ ...result, ...array1, ...array2 ];
}

function sortSomething( array ) {
    if ( array.length <= 1 ) return array;

    const mid = Math.floor( array.length / 2 );
    const left = sortSomething( array.slice( 0, mid ) );
    const right = sortSomething( array.slice( mid ) );

    return doSomething( left, right );
}
/*
!Merge Sort: Time Complexity O( n log( n ) )
!since we split the array in half each time, the number of calls is O( log( n ).
!The while loop inside the doSomething( aka merge ) method will go through all of the array - n times.
!So it 's O(n * log(n));
!Space complexity is O( n ),
!because we are copying the array n times. [ Yes, there are two copies, but each is half of the array. ]
*/
```

```
function mySort( array ) {
    if ( array.length <= 1 ) return array;

    let pivot = array.shift();

    let left = array.filter( x => x < pivot );
    let right = array.filter( x => x >= pivot );

    let sortedLeft = mySort( left );
    let sortedRight = mySort( right );

    return [ ...sortedLeft, pivot, ...sortedRight ];
}
//!QuickSort: Time complexity of O( n2 ) because sort on the left is O( n ) and sort on the right is O( n ).
//!Space complexity of Quick Sort is O( n );
```

```
function anotherSearch( list, target ) {
    if ( list.length === 0 ) return false;

    let mid = Math.floor( list.length / 2 );

    if ( list[ mid ] === target ) {
        return true;
    } else if ( list[ mid ] > target ) {
        return anotherSearch( list.slice( 0, mid ), target );
    } else {
        return anotherSearch( list.slice( mid + 1 ), target );
    }
}

//!Binary Search - Time complexity O( log( n ) );

//!Space complexity O( n )
```

```
//*Transform the following fibonacci function to use memoization to re
const fibonacci = ( n ) => {
    if ( ( n === 1 ) || ( n === 2 ) ) {
        return 1;
    }
    return fibonacci( n - 1 ) + fibonacci( n - 2 );
}
//*-------------------------------------------------------------
const fibonacci = ( n, memo = {
    0: 0,
    1: 1
} ) => {
    if ( n in memo ) return memo[ n ];
    memo[ n ] = fibonacci( n - 1, memo ) + fibonacci( n - 2, memo );
    return memo[ n ];
};
//!--------------End of problem-----------------------------------
```

```
//*Apply tabulation to the iterative version of fibonacci
function fib( n ) {
    //! memoization is usually recursive whereas tabulati
    //! Tabulation
    let table = new Array( n + 1 ); //! this line is in m
    //* and array is stored in concurrent memory location
    //table[0]=0;
    table[ 1 ] = 1; //fib(0)=0
    table[ 2 ] = 1; //fib(1)=1
    for ( let i = 2; i <= n; i++ ) { // one loop that goe
        //table[2]=table[0]+table[1];
        table[ i ] = table[ i - 1 ] + table[ i - 2 ];
        /*
        if(n===1 || n===2)return 1;
        return fib(n-1)+fib(n-2)1;
        */
    }

    return table[ n ]; //fib(n)

}
//!--------------End of problem--------------------------
```

HIDE GOT IT!

Explain the complexity of and write a function that performs insertion sort on an array of numbers

Time complexity: $O(n^2)$; The outer loop i contributes O(n) in isolation. The inner while loop will contribute roughly O(n/2) on average. The two loops are nested so our total time complexity is $O(n * n / 2) = O(n^2)$.

Space Complexity: O(1); We use the same amount of memory and create the same amount of variables regardless of the size of our input.

```
function insertionSort(list) {
    for (let i = 1; i < list.length; i++) {
        value = list[i];
        hole = i;

        while (hole > 0 && list[hole - 1] > value) {
            list[hole] = list[hole - 1];
            hole--;
        }
        list[hole] = value;
    }
}
```

Explain the complexity of and write a function that performs merge sort on an array of numbers.

Time Complexity: O(n * log(n)); Since we split the array in half each time, the number of recursive calls is O(log(n)). The while loop within the merge function contributes O(n) in isolation and we call that for every recursive mergeSort call.
Space Complexity: O(n) : We will create a new subarray for each element in the original input.

Explain the complexity of and write a function that performs quick sort on an array of numbers.

Time Complexity: Avg Case: O(n log(n)) : The partition step alone is O(n). We are lucky and always choose the median as the pivot. This will halve the array length at every step of the recursion for O(log(n)).

Worst Case: $O(n^2)$: We are unlucky and always choose the min or max as the pivot. This means one partition will contain everything, and the other partition is empty, yielding O(n).

Space Complexity: O(n) : Our implementation of quickSort uses O(n) space because of the partition arrays we create.

Explain the complexity of and write a function that performs a binary search on a sorted array of numbers.

Time Complexity: O(log(n)) : The number of recursive calls is the number of times we must halve the array until it's length becomes 0.

Space Complexity: O(n) : Our implementation uses n space due to half arrays we create using slice.

```
    function binarySearch(list, target) {
        if (list.length === 0) return false;

        let mid = Math.floor(list.length / 2;

        if (list[mid] === target) {
            return true;
        } else if (list[mid] > target) {
            return binarySearch(list.slice(0, mid, target);
        } else {
            return binarySearch(list.slice(mid+1), target);
        }
    }
```

HIDE GOT IT!

For a linked list comprised of a Node class and a List class, what properties will the list and node require?

List: head, tail; length;
Node: value; next;

HIDE GOT IT! For a linked list class implementation, what methods will the linked list class require?

1. constructor(value)
2. addHead(value)
3. addTail(value)
4. insertAt(idx)
5. removeTail()
6. removeHead()
7. remove(idx)
8. contains(value)
9. set(value, idx)
10. size

HIDE GOT IT! What type of data structure is a stack?

1. FIFO: First In First Out
2. LIFO: Last In Last Out

LIFO : Last in (goes on top of the stack), Last out (removed from top of the stack)

HIDE GOT IT!

When implementing a stack, define the properties needed on both the Node and Stack classes.

Node: value; next;
Stack: top; length;

SHOW GOT IT! When implementing a stack, define the methods you will need for the stack.

HIDE GOT IT! What type of data structure is a queue?

1. LIFO : Last In First Out
2. FIFO : First in First Out

FIFO : First in First Out;

HIDE GOT IT! What properties would you need on the Node and Queue classes?

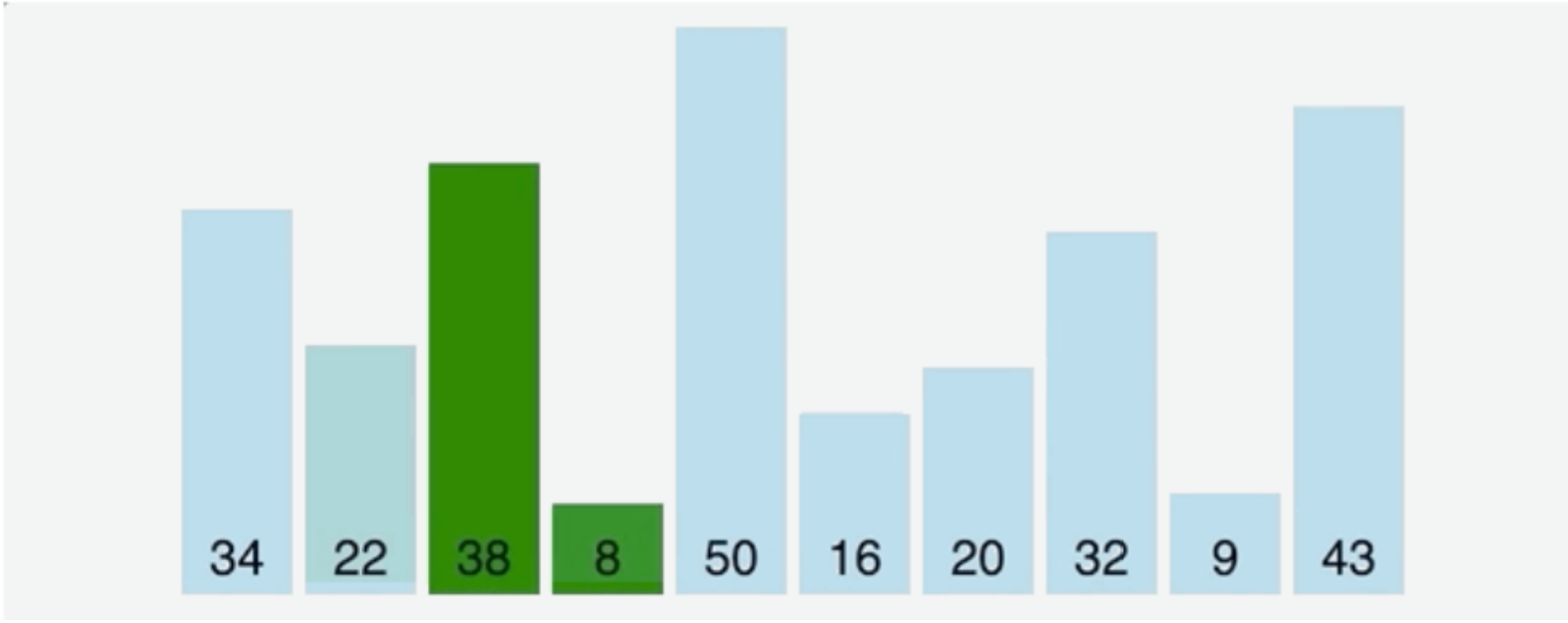
Node: value; next;
Queue: front; back; length;

HIDE GOT IT! What methods would you need on the queue class?

1. enqueue(value) : to add the new node to the back of the queue
2. dequeue: removes a value from the front of the queue
3. size: Returns the size of the queue

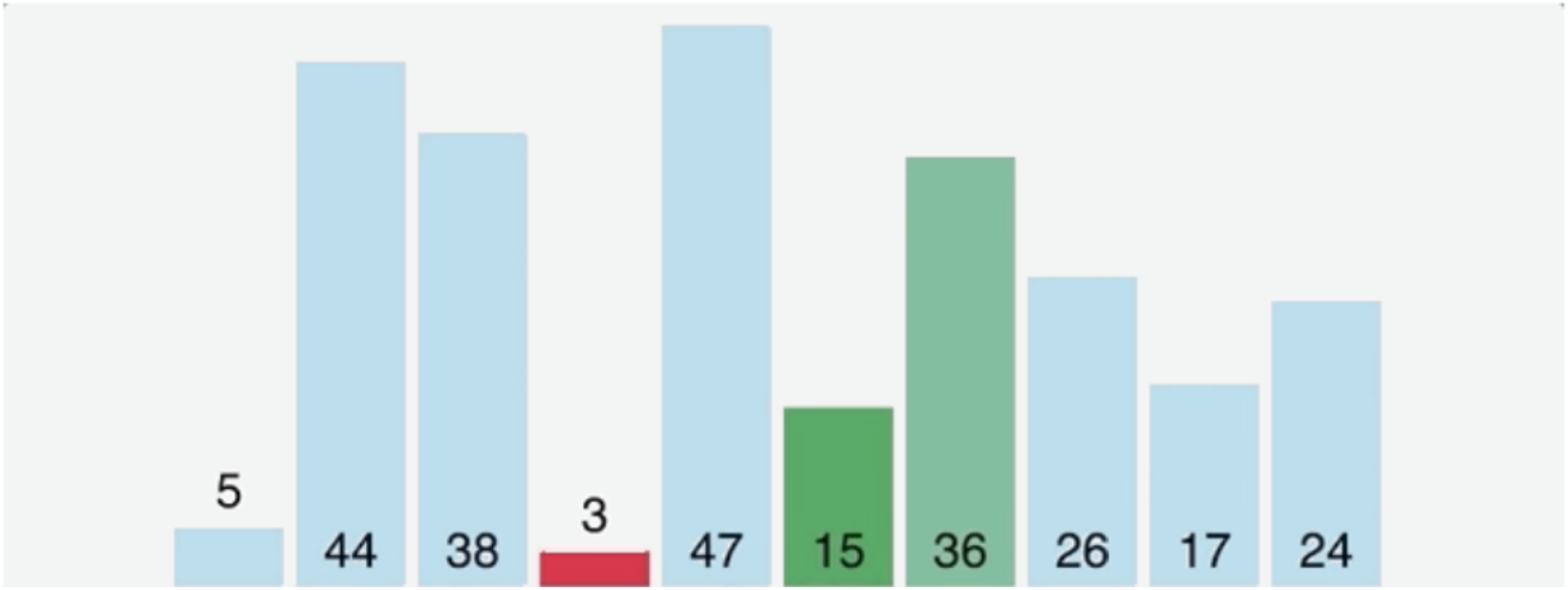
HIDE GOT IT! Explain in words the algorithm for bubbleSort

bubbleSort will look at every element of the array, comparing it to the next element. If the next element is smaller, we will swap that element. We continue that to the end of the array. Each time we swap we set swapped to true. We continue iterating through the array until we've made it all the way through the array without swapping any elements.



HIDE GOT IT! Explain in words how the selection sort works.

Selection Sort is very similar to Bubble Sort. The major difference between the two is that Bubble Sort bubbles the largest elements up to the end of the array, while Selection Sort selects the smallest elements of the array and directly places them at the beginning of the array in sorted position. Selection sort will utilize swapping just as bubble sort did.



HIDE GOT IT! Explain in words what insertion sort does.

Insertion Sort is similar to Selection Sort in that it gradually builds up a larger and larger sorted region at the left-most end of the array. However, Insertion Sort differs from Selection Sort because this algorithm does not focus on searching for the right element to place (the next smallest in our Selection Sort) on each pass through the array. Instead, it focuses on sorting each element in the order they appear from left to right, regardless of their value, and inserting them in the most appropriate position in the sorted region.

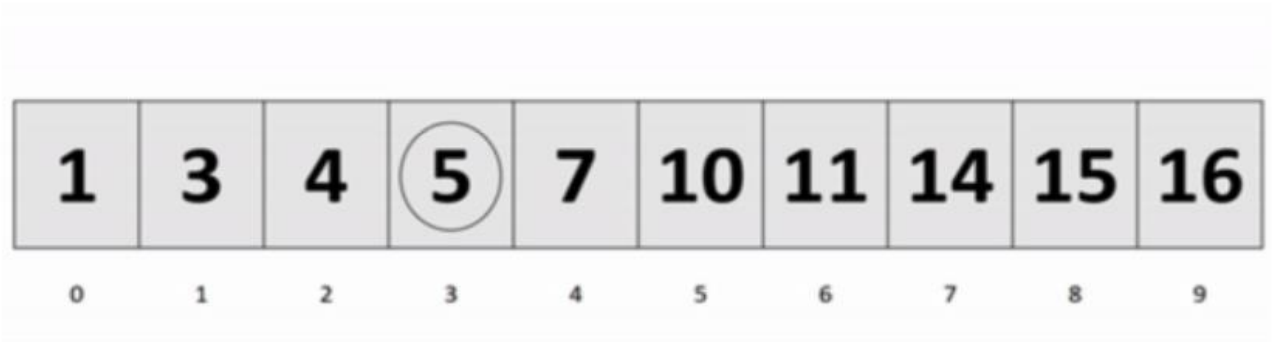
HIDE GOT IT! Describe merge sort in words

Merge sort is based on the premise that it's easy to merge elements of two sorted arrays into a single sorted array. This screams recursion, with your base case being an empty array. Also, if there's only one element in the array you can consider that array as sorted.

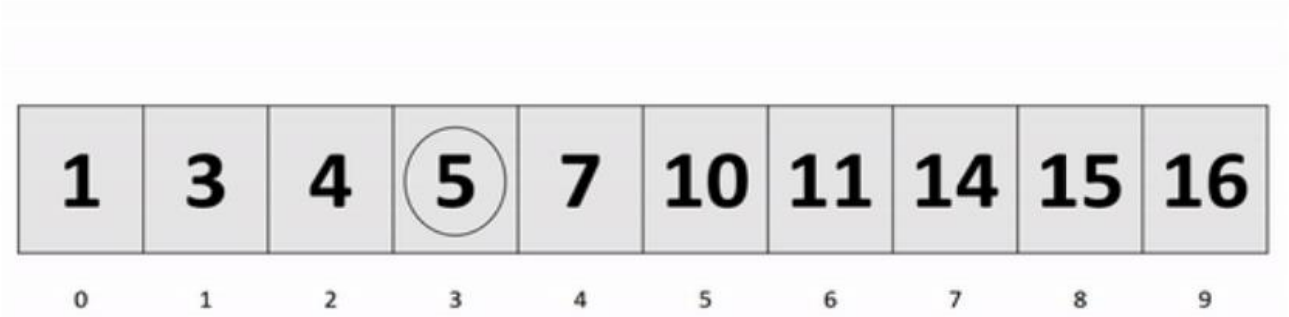
HIDE GOT IT! Describe binary search in words

Binary search only works on sorted trees or lists. Cut the list in half, look at the end element of the left half and if it's greater than what you're looking for, choose your next search for the right element. Otherwise, choose your lefthalf for your next search. Rinse and repeat.

Take for example, using the binary search algorithm on an array of integers:



Looking for the element 5, will cut your search space in 1/2 with each attempt:



```
//! Single Linked List
class Node {
    constructor(val) {
        this.value = val;
        this.next = null;
    }
}
class LinkedList {
    constructor() {
        this.head = null;
        this.tail = null;
        this.length = 0;
    }
```

```
addToTail(val) {
    const newNode = new Node(val);
    if (!this.head) {
        this.head = newNode;
    } else {
        this.tail.next = newNode;
    }
    this.tail = newNode;
    this.length++;
    return this;
}
```

```
removeTail() {
        if (!this.head) return undefined;
        let current = this.head;
        let newTail = current;
        while (current.next) {
            newTail = current;
            current = current.next;
        }
        this.tail = newTail;
        this.tail.next = null;
        this.length--;
        if (this.length === 0) {
            this.head = null;
            this.tail = null;
        }
        return current;
    }
```

```
addToHead(val) {
        let newNode = new Node(val);
        if (!this.head) {
            this.head = newNode;
            this.tail = newNode;
        } else {
            newNode.next = this.head;
            this.head = newNode;
        }
        this.length++;
        return this;
    }
```

```
removeHead() {
        if (!this.head) return undefined;
        const currentHead = this.head;
        this.head = currentHead.next;
        this.length--;
        if (this.length === 0) {
            this.tail = null;
        }
        return currentHead;
    }
    contains(target) {
        let node = this.head;
        while (node) {
            if (node.value === target) return true;
            node = node.next;
        }
        return false;
    }
```

```
get(index) {
        if (index < 0 || index >= this.length) return null;
        let counter = 0;
        let current = this.head;
        while (counter !== index) {
            current = current.next;
            counter++;
        }
        return current;
    }
```

```
set(index, val) {
        const foundNode = this.get(index);
        if (foundNode) {
            foundNode.value = val;
            return true;
        }
        return false;
    }
    insert(index, val) {
        if (index < 0 || index > this.length) return false;
        if (index === this.length) return !!this.addToTail(val);
        if (index === 0) return !!this.addToHead(val);
        const newNode = new Node(val);
        const prev = this.get(index - 1);
        const temp = prev.next;
        prev.next = newNode;
        newNode.next = temp;
        this.length++;
        return true;
    }
```

```
 remove(index) {
        if (index < 0 || index >= this.length) return undefined;
        if (index === 0) return this.removeHead();
        if (index === this.length - 1) return this.removeTail();
        const previousNode = this.get(index - 1);
        const removed = previousNode.next;
        previousNode.next = removed.next;
        this.length--;
        return removed;
    }
    size() {
        return this.length;
    }
}
exports.Node = Node;
exports.LinkedList = LinkedList;
```

```
//!Queue Array:
// Using the first element of the array as the "front" of the queue
class QueueArray {
    constructor() {
        this.queue = [];
    }
    enqueue(value) {
        this.queue.push(value);
    }
    dequeue() {
        return this.queue.shift();
    }
    peek() {
        return this.queue[0];
    }
}
```

```
// Using the last element of the array as the "front" of the queue
class QueueArray {
    constructor() {
        this.queue = [];
    }
    enqueue(value) {
        this.queue.unshift(value);
    }
    dequeue() {
        return this.queue.pop();
    }
    peek() {
        return this.queue[this.queue.length - 1];
    }
}
```

```
//! Queue Node
class Node {
    constructor(val) {
        this.value = val;
        this.next = null;
    }
}

class Queue {
    constructor() {
        this.front = null;
        this.back = null;
        this.length = 0;
    }
```

```
enqueue(val) {
        const newNode = new Node(val);
        if (!this.front) {
            this.front = newNode;
            this.back = newNode;
        } else {
            this.back.next = newNode;
            this.back = newNode;
        }
        return ++this.length;
    }
```

```
dequeue() {
        if (!this.front) {
            return null;
        }
        const temp = this.front;
        if (this.front === this.back) {
            this.back = null;
        }
        this.front = this.front.next;
        this.length--;
        return temp.value;
    }

    size() {
        return this.length;
    }
}
```

```javascript
//!Stack Array
// Using the last element of the array as the "top" of the stack
// This is more efficient than the second implementation because we can push and
// pop from an array in O(1) time since we don't have to reassign any indices.
class StackArray {
    constructor() {
        this.stack = [];
    }

    push(value) {
        this.stack.push(value);
    }

    pop() {
        return this.stack.pop();
    }

    peek() {
        return this.stack[this.stack.length - 1];
    }
}
```

```
// Using the first element of the array as the "top" of the stack
// This is not as efficient as the previous implementation since we have to
// reassign indices for a shift and unshift, but the user will see the same
// functionality.
class StackArray {
    constructor() {
        this.stack = [];
    }

    push(value) {
        this.stack.unshift(value);
    }

    pop() {
        return this.stack.shift();
    }

    peek() {
        return this.stack[0];
    }
}
```

```
class Node {
    constructor(value) {
        this.value = value;
        this.next = null;
    }
}

class StackNode {
    constructor() {
        this.top = null;
        this.length = 0;
    }
```

```
push(val) {
        const newNode = new Node(val);
        if (!this.top) {
            this.top = newNode;
        } else {
            const temp = this.top;
            this.top = newNode;
            this.top.next = temp;
        }
        this.length++;
        return this.length;
    }
```

```
pop() {
        if (!this.top) {
            return null;
        }
        const temp = this.top;
        this.top = this.top.next;
        this.length--;
        return temp.value;
    }

    peek() {
        if (!this.top) {
            return null;
        }
        return this.top.value;
    }
```

```
 size() {
        return this.length;
    }
}
```