Table

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

A picture containing timeline

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Text, letter

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Chart, bar chart

Description automatically generated

Chart, bar chart

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

*//! 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;

    }

}

*//!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;

    }

}