**DÍA 3**

**JAVASCRIPT AVANZADO II**

**HOMEWORK**

"use strict";

// **Closures** => Funciones q retornan funciones.

Un ***closure*** es una función que retorna otra función, que hace uso de los argumentos de la función creadora (padre). Lo que crea, genera un nuevo contexto de ejecución que está enlazado al contexto de ejecución del padre. Mas generalmente, una Closure es una función que devuelve funciones, sea una función, un objeto con funciones o un array de funciones que mantiene un contexto de ejecución con algunas variables que son heredadas.

function counter() {

/\*

Ejercicio 1

La función counter debe retornar otra función. Esta función retornada debe actuar como un contador, retornando un valor numérico que empieza en 1 e incrementa con cada invocación.

Ejemplo:

const nuevoContador = counter()

nuevoContador() // 1

nuevoContador() // 2

nuevoContador() // 3

const otroContador = counter()

otroContador() // 1

otroContador() // 2

otroContador() // 3

\*/

let count = 0;

return function () {

count++;

return count;

}

}

function cacheFunction(cb) {

/\*

Ejercicio 2

Tu tarea aquí es lograr, mediante un closure, que cacheFunction actúe como una memoria caché para el callback que recibe por parámetro (cb); es decir, que "recuerde" el resultado de cada operación que hace, de manera que, al realizar una operación por segunda vez, se pueda obtener el resultado de esa "memoria" sin tener que efectuar otra vez cálculos que ya se hicieron anteriormente.

cacheFunction debe retornar una función. Esta función debe aceptar un argumento (arg) e invocar a cb con ese argumento; hecho eso, debe guardar el argumento junto con el resultado de la invocación (tip: usá un objeto donde cada propiedad sea el argumento, y su valor el resultado de la correspondiente invocación a cb) de manera que, la próxima vez que reciba el mismo argumento, no sea necesario volver a invocar a cb, porque el resultado estará guardado en la "memoria caché".

Ejemplo:

function square(n){

return n \* n

}

const squareCache = cacheFunction(square)

squareCache(5) // invocará a square(5), almacenará el resultado y lo retornará

squareCache(5) // no volverá a invocar a square, simplemente buscará en la caché cuál es el resultado de square(5) y lo retornará (tip: si usaste un objeto, podés usar hasOwnProperty)

\*/

var cacheObj = {};

return function(arg) {

if(cacheObj.hasOwnProperty(arg)) return cacheObj[arg];

cacheObj[arg]=cb(arg);

return cacheObj[arg];

}

}

// Bind

var instructor = {

nombre: "Franco",

edad: 25,

};

var alumno = {

nombre: "Juan",

curso: "FullStack",

};

function getNombre() {

return this.nombre;

}

/\*

Ejercicio 3

IMPORTANTE: no modificar el código de arriba (variables instructor y alumno, y función getNombre)

Usando el método bind() guardar, en las dos variables declaradas a continuación, dos funciones que actúen como getNombre pero retornen el nombre del instructor y del alumno, respectivamente.

\*/

let getNombreInstructor = getNombre.bind(instructor);

let getNombreAlumno = getNombre.bind(alumno);

/\*

Ejercicio 4

Sin modificar la función crearCadena, usar bind para guardar, en las tres variables declaradas a continuación, tres funciones que retornen una cadena (string) y el delimitador especificado (asteriscos, guiones, guiones bajos, respectivamente). Las funciones obtenidas deberían recibir solamente un argumento - la cadena de texto - ya que los otros argumentos habrán sido "bindeados".

\*/

function crearCadena(delimitadorIzquierda, delimitadorDerecha, cadena) {

return delimitadorIzquierda + cadena + delimitadorDerecha;

}

let textoAsteriscos = crearCadena.bind(null,"\*","\*");

let textoGuiones = crearCadena.bind(null,"-","-");

let textoUnderscore = crearCadena.bind(null,"\_","\_");

// No modifiquen nada debajo de esta linea

// --------------------------------

module.exports = {

counter,

cacheFunction,

getNombreInstructor,

getNombreAlumno,

textoAsteriscos,

textoGuiones,

textoUnderscore,

};

**DÍA 4**

**ESTRUCTURACIÓN DE DATOS I**

**RECURSIÓN**

**Ejercicio 1:**

Funcion que cuenta los elementos dentro de una Array, incluyendo las sub arrays.

var miArr=[1,2,3,4,[5,6]];

function n(arr) {

let count=0;

for(const element of arr) {

if (Array.isArray(element)) {

**count += n(element);**

} else {

count++;

}

}

return count

}

console.log(n(miArr));

**Ejercicio 2:**

Función que divide un número por 2 las veces que pueda hasta llegar a cero. (Caso base, 0 o 1 son 0)

function div(val) {

let count=0;

if (val < 2) return 0;

const newVal = Math.floor(val/2);

count++;

**count += div(newVal);**

return count

}

console.log(div(10));

**ESTRUCTURAS DE DATOS PARTE I**

Estructura de datos nos referimos a cómo se organizan los datos cuando programamos. En un cache quiero almacenar datos de tipo clave – valor.

Tipos de estructuras: Arreglos, Sets (Arreglo que no repite valores)

Stacks (Pilas) => LIFO (Last in, First Out) (push-pop)

Queue (Colas) => FIFO (First in, First Out) (push-shift)

Colas con el métodos de array

Function Queue() {

this.data = []; //Crea una clase

}

Queue.prototype.enqueue = function(val) { //Metodos

this.data.unshift(val)

}

Queue.prototype.dequeue = function(val) { //Metodos

this.data.pop(val)

}

Queue.prototype.size = function(val) { //Metodos

this.data.length

}

**HOMEWORK**

/\*

Definir las funciones recursivas nFactorial y nFibonacci.

nFactorial(n) debe retornar el factorial de n sabiendo que, siendo n un número natural, su factorial (representado como n!) es el producto de n por todos los números naturales menores que él y mayores a 0. Ejemplo: 5! = 5 \* 4 \* 3 \* 2 \* 1

nFibonacci(n) debe retornar el enésimo número de la secuencia de Fibonacci, tomando al 0 y al 1, respectivamente, como primer y segundo elementos de la misma, y sabiendo que cualquier elemento que se agregue a esta secuencia será el resultado de la suma del último elemento y el anterior.

Ejemplo: nFibonacci(7) retornará 13, ya que 13 es el dígito que está en la posición 7 de la secuencia.

Secuencia:  0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Como ejercicio adicional y completamente opcional, al terminar de resolver este problema pueden intentar definir funciones que logren los mismos resultados pero de manera iterativa.

\*/

function nFactorial(val) {

  if (val <= 1) return 1;

  return val \* nFactorial(val - 1);

}

function nFibonacci(n) {

  if (n <= 1) return n;

  return  nFibonacci(n - 1) + nFibonacci(n - 2);

}

/\*

Implementar la clase Queue, sabiendo que es una estructura de tipo FIFO, donde el primer elemento que ingresa es el primero que se quita. Definir los siguientes métodos:

  - enqueue: agrega un valor respetando el orden.

  - dequeue: remueve un valor respetando el orden. Retorna undefined cuando la queue está vacía.

  - size: retorna el tamaño (cantidad de elementos) de la queue.

Pueden utilizar class o función constructora.

\*/

function Queue() {

  this.datos = [];

}

Queue.prototype.enqueue = function (val) {

  return this.datos.unshift(val)

}

Queue.prototype.dequeue = function (val) {

  return this.datos.pop(val)

}

Queue.prototype.size = function () {

  return this.datos.length

}

**ESTRUCTURAS DE DATOS PARTE II**

Estructura de datos nos referimos a cómo se organizan los datos cuando programamos. En un cache quiero almacenar datos de tipo clave – valor.

**LINKED LIST**

// function List() {

//     this.\_length = 0;

//     this.head = null;//aca se creo el primer vagon.

// }

function LinkedList() {

    this.head = null;

}

function Node(val) {

    this.data = val;

    this.next = null;

}

LinkedList.prototype.add = function (val) {

    const node = new Node(val); //creo un nuevo nodo

    if(!this.head) {//Si no hay vagon inicial lo crea

      this.head = node;

    } else {

      let current = this.head;// aca almaceno la referencia del nodo

      while(current.next) { //para ir recorriendo

        current = current.next;//cambio la referencia

      }

      current.next = node;//Le asigno el valor al ultimo vagon

    }

}

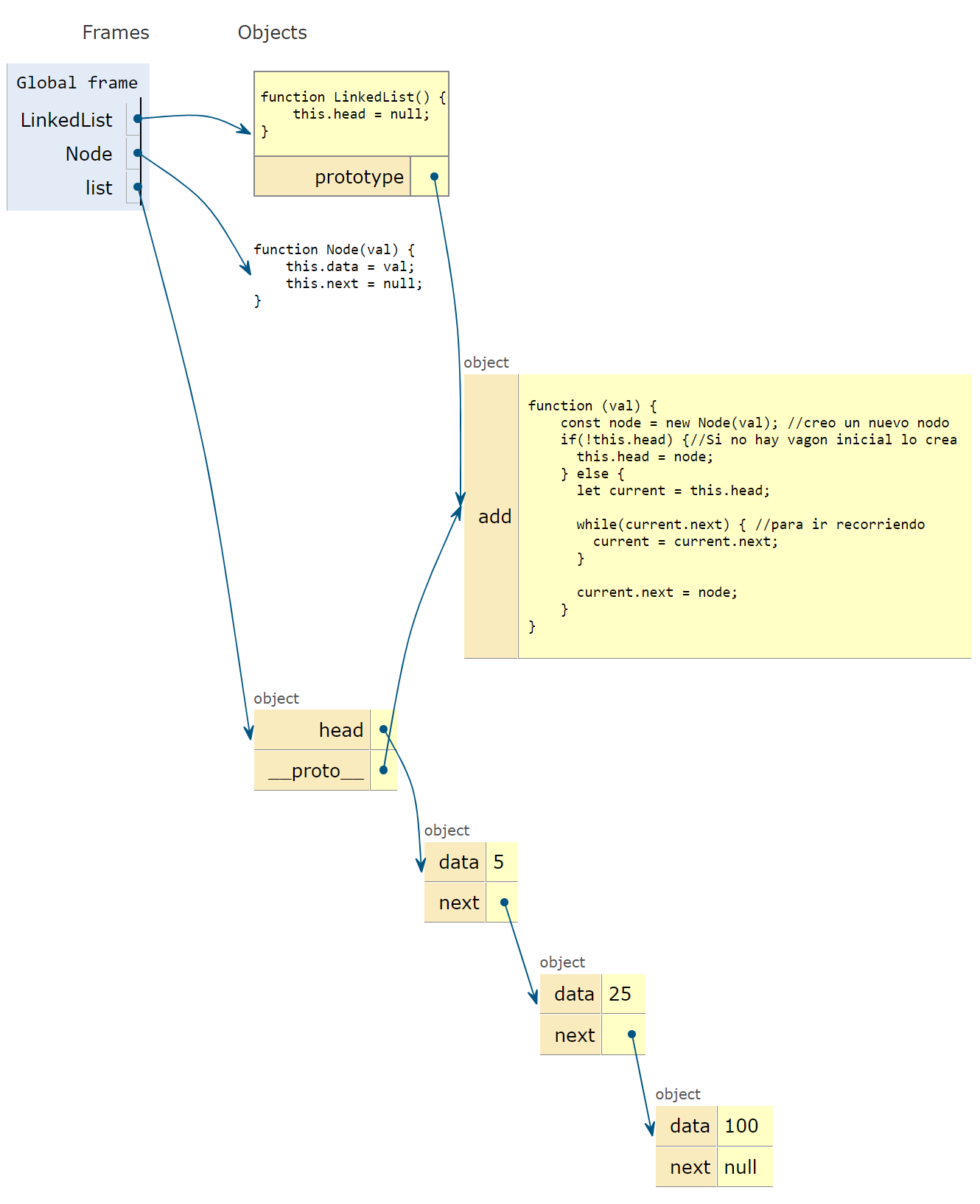
const list = new LinkedList()

list.add(5);

list.add(25);

list.add(100);

console.log(list);



**DOUBLE LINKED LIST**

function LinkedList() {

    this.head = null;

}

function Node(val) {

    this.data = val;

    this.next = null;

    this.prev = null;

}

LinkedList.prototype.add = function (val) {

    const node = new Node(val); //creo un nuevo nodo

    if(!this.head) {//Si no hay vagon inicial lo crea

      this.head = node;

    } else {

      let current = this.head;// aca almaceno la referencia del nodo

      while(current.next) { //para ir recorriendo

        current = current.next;//cambio la referencia

      }

      node.prev = current;

      current.next = node;//Le asigno el valor al ultimo vagon

    }

}

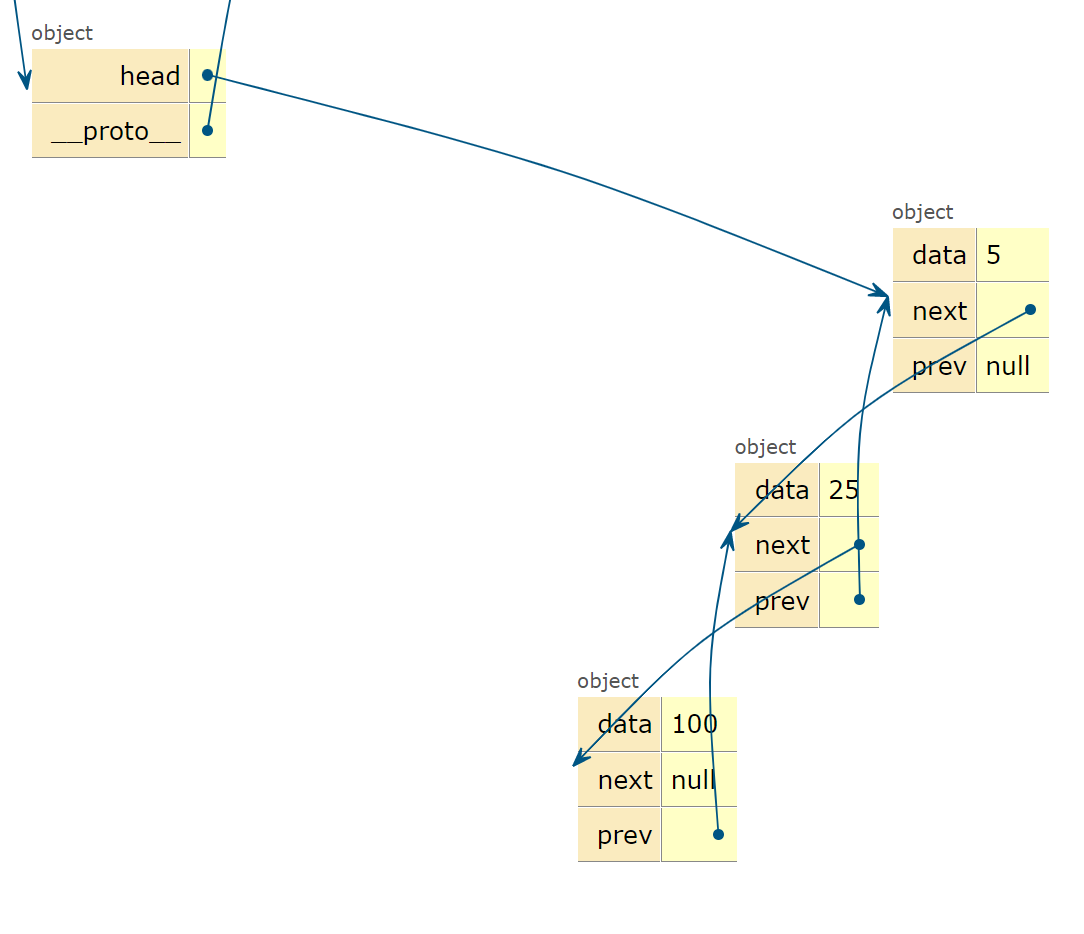
const list = new LinkedList()

list.add(5);

list.add(25);

list.add(100);

console.log(list);



**HASH TABLE**

Dados ciertos datos, buscar algo que lo identifique y guardarlo y almacenarlo en una especie de cajón.