**LINKED LISTS**

/\* LISTAS ENLAZDAS \*/

function Lista(){

    this.point = null;

    this.len = 0;

}

function Nodo(data) {

    this.data = data;

    this.next = null;

}

Lista.prototype.push = function(data) { // Insert Last

    var newNodo = new Nodo(data);

    if (this.point == null) {

        this.point = newNodo;

    } else {

        pointer = this.point;

        while (pointer.next!=null) {

            pointer = pointer.next;

        }

        pointer.next = newNodo;

    }

    this.len++;

}

Lista.prototype.insertFirst = function(data) {

    var newNodo = new Nodo(data);

    if (this.point == null) {

        this.point = newNodo;

    } else {

        var pointer = this.point;

        this.point = newNodo;

        newNodo.next = pointer;

    }

    this.len++;

}

Lista.prototype.insert = function(data,pos) {

    if (this.len < pos) return this.push(data);

    var newNodo = new Nodo(data);

    if (pos == 1) {

        newNodo.next = this.point;

        this.point = newNodo;

    }

    var pointer = this.point;

    while (pos > 1) {

        pos--;

        pointer = pointer.next;

    }

    newNodo.next = pointer.next;

    pointer.next = newNodo;

}

Lista.prototype.print = function() {

    pointer = this.point;

    while (pointer.next!=null) {

        console.log(pointer.data);

        pointer = pointer.next;

    }

    console.log(pointer.data);

}

Lista.prototype.printOne = function(pos) {

    if (pos > this.len) return console.log('Posicion fuera de rango.');

    pointer = this.point;

    var aux = pos;

    while (aux > 1) {

        aux--;

        pointer = pointer.next;

    }

    console.log('Data ['+pos+'] = '+pointer.data)

}

Lista.prototype.deleteFirst = function() {

    if (this.len == 0) return console.log('Es una lista vacia');

    this.point = this.point.next;

    this.len--;

}

Lista.prototype.deleteLast = function() {

    pointer = this.point;

    while (pointer.next.next!=null) {

        pointer = pointer.next;

    }

    pointer.next = null;

    this.len--;

}

Lista.prototype.find = function(val) {

    if (this.len == 0) return console.log('Es una lista vacia');

    var pointer = this.point;

    var check = false;

    if (pointer.data == val) check = true;

    while (!check && pointer.next != null) {

        pointer = pointer.next;

        if (pointer.data == val) check = true;

    }

    if (check) {

        return pointer;

    } else {

        return undefined;

    }

}

// ----- LinkedList -----

// Deben completar la siguiente implementacion 'OrderedLinkedList'(OLL)

// que es muy similar a las LinkedList vistas en clase solo que

// los metodos son distintos y deben de estar pensados para conservar la lista

// ordenada de mayor a menor.

// ejemplos:

// head --> 5 --> 3 --> 2 --> null

// head --> 4 --> 3 --> 1 --> null

// head --> 9 --> 3 --> -1 --> null

// Las dos clases principales ya van a estar implementadas a continuacion:

function OrderedLinkedList() {

    this.head = null;

}

// notar que Node esta implementado en el archivo DS

// Y el metodo print que permite visualizar la lista:

OrderedLinkedList.prototype.print = function () {

    let print = 'head'

    let pointer = this.head

    while (pointer) {

        print += ' --> ' + pointer.value

        pointer = pointer.next;

    }

    print += ' --> null'

    return print

}

// EJERCICIO 4

// Crea el metodo 'add' que debe agregar nodos a la OLL de forma que la misma se conserve ordenada:

// Ejemplo:

// > LL.print()

// < 'head --> null'

// > LL.add(1)

// > LL.print()

// < 'head --> 1 --> null'

//    2       c

// > LL.add(5)

// > LL.print()

// < 'head --> 5 --> 1 --> null'

// > LL.add(4)

// > LL.print()

// < 'head --> 5 --> 3 --> 1 --> null'

//               4

OrderedLinkedList.prototype.add = function (val) {

  const node = new Node(val);//creo mi vagon nuevo

  if (!this.head) {//si el primer vagon no existe

    this.head = node;//lo creo, es decir le agrego valor y next

  } else {//si hay primer vagon

    let current = this.head;//me paro en el primer vagon

    if (val > current.value) {//si nuevo vagon es mayor al primero

      node.next = current;//el next de mi nuevo vagon es el actual head

      this.head = node;//ahora el primero es el nuevo

    }

    while (current.next) {//mientras este vagon este conectado a otro

      if (node.value > current.next.value) {//si nuevo vagon es mayor al siguiente

        node.next = current.next;//el next de mi nuevo vagon es el siguiente

        current.next = node;//ahora el siguiente del actual es el nuevo

      }

      current = current.next;//salto al siguiente vagon

    }

    current.next = node;//cuando estoy en el pointer vagon le agrego otro vagon a next

  }

}

// EJERCICIO 5

// Crea el metodo 'removeHigher' que debe devolver el valor mas alto de la linked list

// removiendo su nodo corresponidente:

// Ejemplo:

// > LL.print()

// < 'head --> 5 --> 4 --> 1 --> null'

// > LL.removeHigher()

// < 5

// > LL.removeHigher()

// < 4

// > LL.removeHigher()

// < 1

// > LL.removeHigher()

// < null

OrderedLinkedList.prototype.higher = function () {

  let pointer = this.head

  let min = pointer.value

  while (pointer.next) {

    pointer = pointer.next;

    if (pointer.value > min) min = pointer.value;

  }

  return min;

}

OrderedLinkedList.prototype.removeHigher = function () {

  if (!this.head) return null;//si no hay primer vagon paila

  let value = this.higher();//traigo el numero mayor

  let pointer = this.head;//me paro en el primer vagon

  if (pointer.value === value) {//si el mayor es el primero

    this.head = this.head.next

    return value

  }

  while (pointer.next.value != value) {//si el valor del sig vagon es dif a value

    pointer = pointer.next;//salto al siguiente vagon

  }

  pointer.next.next ? pointer.next = pointer.next.next : pointer.next = null; //reemplazo next

  return value;

}

// EJERCICIO 6

// Crea el metodo 'removeLower' que deve devolver el valor mas bajo de la linked list

// removiendo su nodo corresponidente:

// Ejemplo:

// > LL.print()

// < 'head --> 5 --> 4 --> 1 --> null'

// > LL.removeHigher()

// < 1

// > LL.removeHigher()

// < 4

// > LL.removeHigher()

// < 5

// > LL.removeHigher()

// < null

OrderedLinkedList.prototype.lower = function () {

  let pointer = this.head

  let min = pointer.value

  while (pointer.next) {

    pointer = pointer.next;

    if (pointer.value < min) min = pointer.value;

  }

  return min;

}

OrderedLinkedList.prototype.removeLower = function(){

  if (!this.head) return null;//si no hay primer vagon paila

  let value = this.lower();//traigo el numero mayor

  let pointer = this.head;//me paro en el primer vagon

  if (pointer.value === value) {//si el mayor es el primero

    this.head = this.head.next

    return value

  }

  while (pointer.next.value != value) {//si el valor del sig vagon es dif a value

    pointer = pointer.next;//salto al siguiente vagon

  }

  pointer.next.next ? pointer.next = pointer.next.next : pointer.next = null; //reemplazo next

  return value;

}

function Pila() {

  this.head = null;

  this.len = 0;

}

function Nodo(data) {

  this.data = data;

  this.next = null;

}

Pila.prototype.push = function(data) {

  var newNodo = new Nodo(data);

  newNodo.next = this.head;

  this.head = newNodo;

  this.len++;

}

Pila.prototype.pop = function() {

  if (this.head == null) return null;

  var popNodo = this.head;

  this.head = popNodo.next;

  this.len--;

  return popNodo;

}

Pila.prototype.print = function() {

  if (this.head == null) return null;

  var pointer = this.head;

  while (pointer != null) {

    console.log(pointer.data);

    pointer = pointer.next;

  }

}

// INVERTIR ARRAYS CON PILAS //

function switchArray(arr) {

  var pila = new Pila();

  while (arr.length > 0) {

    pila.push(arr.shift());

  }

  return function vaciar(pila,array) {

    if (pila.head == null) return array;

    array.push(pila.pop().data);

    return vaciar(pila,array);

  }(pila,arr);

}

/\*

Implementar la clase LinkedList, definiendo los siguientes métodos:

  - add: agrega un nuevo nodo al final de la lista;

  - remove: elimina el último nodo de la lista y retorna su valor (tener en cuenta el caso particular de una lista de un solo nodo y de una lista vacía);

  - search: recibe un parámetro y lo busca dentro de la lista, con una particularidad: el parámetro puede ser un valor o un callback. En el primer caso, buscamos un nodo cuyo valor coincida con lo buscado; en el segundo, buscamos un nodo cuyo valor, al ser pasado como parámetro del callback, retorne true.

  Ejemplo:

  search(3) busca un nodo cuyo valor sea 3;

  search(isEven), donde isEven es una función que retorna true cuando recibe por parámetro un número par, busca un nodo cuyo valor sea un número par.

  En caso de que la búsqueda no arroje resultados, search debe retornar null.

\*/

function LinkedList() {//creo una clase de linked list

  this.head = null;//creo el primer vagon vacion

}

function Node(value) {//creo las caracteristicas de cada vagon (objeto)

  this.value = value;//cada vagon debe tener una propiedad valor

  this.next = null;//cada vagon debe tener una propiedad next (vinculo) inicial/ null

}

LinkedList.prototype.add = function (value) {//funcion que agrega valores prototipo de linked list

  const node = new Node(value);//Instancion node como Node (props valor,next)

  if (!this.head) {//si el primer vagon no existe

    this.head = node;//lo creo, es decir le agrego valor y next

  } else {//si hay primer vagon

    let current = this.head;//me paro en el primer vagon

    while (current.next) {//mientras este vagon tenga una propiedad next

      current = current.next;//salto al siguiente vagon

    }

    current.next = node;//cuando estoy en el ultimo vagon le agrego otro vagon a next

  }

}

LinkedList.prototype.remove = function (value) {

  if (!this.head) return null;

  let penultimo = this.head;

  if (!penultimo.next) {//si solo hay un elemento

    let miVal = penultimo.value;//asigo el valor a una variable

    delete penultimo.value;//borro value

    delete penultimo.next;//borro next

    this.head = null;//asigno null a head

    return miVal;//retorno la variable

  }

  while (penultimo.next.next) {//mientras hayan dos vagones adelante

    penultimo = penultimo.next;//salto un vagon

  }//llegue al penultimo vagon

  let miVal = penultimo.next.value;//asigno el valor a una variable

  penultimo.next = null;//asigno null a next

  return miVal;

}

LinkedList.prototype.search = function (value) {

  if (!this.head) return null;//si no hay primer vagon paila

  let ultimo = this.head;//me paro en el primer vagon

  if (typeof (value) === 'function') {//si el argumento es funcion

    while (!value(ultimo.value)) {//si retorna false

      if (!ultimo.next) return null;//si no hay mas vagones, paila

      ultimo = ultimo.next;//salto al siguiente vagon

    }

  } else {

    while (ultimo.value != value) {//si el valor del vagon es dif a value

      if (!ultimo.next) return null;//si no hay mas vagones, paila

      ultimo = ultimo.next;//salto al siguiente vagon

    }

  }

  return ultimo.value;

}

/\*

Implementar la clase HashTable.

Nuetra tabla hash, internamente, consta de un arreglo de buckets (slots, contenedores, o casilleros; es decir, posiciones posibles para almacenar la información), donde guardaremos datos en formato clave-valor (por ejemplo, {instructora: 'Ani'}).

Para este ejercicio, la tabla debe tener 35 buckets (numBuckets = 35). (Luego de haber pasado todos los tests, a modo de ejercicio adicional, pueden modificar un poco la clase para que reciba la cantidad de buckets por parámetro al momento de ser instanciada.)

La clase debe tener los siguientes métodos:

  - hash: función hasheadora que determina en qué bucket se almacenará un dato. Recibe un input alfabético, suma el código numérico de cada caracter del input (investigar el método charCodeAt de los strings) y calcula el módulo de ese número total por la cantidad de buckets; de esta manera determina la posición de la tabla en la que se almacenará el dato.

  - set: recibe el conjunto clave valor (como dos parámetros distintos), hashea la clave invocando al método hash, y almacena todo el conjunto en el bucket correcto.

  - get: recibe una clave por parámetro, y busca el valor que le corresponde en el bucket correcto de la tabla.

  - hasKey: recibe una clave por parámetro y consulta si ya hay algo almacenado en la tabla con esa clave (retorna un booleano).

Ejemplo: supongamos que quiero guardar {instructora: 'Ani'} en la tabla. Primero puedo chequear, con hasKey, si ya hay algo en la tabla con el nombre 'instructora'; luego, invocando set('instructora', 'Ani'), se almacenará el par clave-valor en un bucket específico (determinado al hashear la clave)

\*/

function HashTable() {//creo una clase HashTable con:

  this.numBuckets = 35;//creo 35 cajones

  this.closet = new Array(this.numBuckets);//creo una lista vacia

}

HashTable.prototype.hash = function (miString) {//funcion q asigna la posicion

  let acum = 0;

  for (let i = 0; i < miString.length; i++) {//recorre la cadena

    acum += miString.charCodeAt(i);//acumula el valor

  }

  return acum % this.numBuckets;//retorna el modulo

}

HashTable.prototype.set = function (clave, valor) {

  if (typeof clave !== 'string') throw new TypeError('Keys must be strings');//creo el tipo de error

  let mibucket = this.hash(clave);//asigno a mibucket la posicion de clave

  let micloset = this.closet;//asigno a mi closet una matriz vacia

  if (!micloset[mibucket]) micloset[mibucket] = {};

  micloset[mibucket][clave] = valor;

}

HashTable.prototype.get = function (key) {

  let mibucket = this.hash(key);//Busco donde esta mibucket, no tengo q recorrer

  if (this.closet[mibucket]) return this.closet[mibucket][key];

}

HashTable.prototype.hasKey = function (key) {

  let mibucket = this.hash(key);//Busco donde esta mibucket, no tengo q recorrer

  return this.closet[mibucket].hasOwnProperty(key);

}