CLASE 05 | ESTRUCTURA DE DATOS II

POR ALEJO BENGOECHEA

ESTRUCTURA DE DATOS II

▼ LISTAS ENLAZADAS

Las ***listas enlazadas*** son otro tipo de estructura de datos. Son una secuencia de *nodos* enlazados que contienen información. Cada *nodo*, además de tener datos, también tiene *links* con otros *nodos*. Según la cantidad de restricciones que tengan los links, estas pueden ser ***simplemente enlazadas***, ***dobles*** o ***múltiples***. Hay que destacar que estas listas no son estructuras nativas de JavaScript.

A diferencia de los *arrays*, en lo que podemos acceder a cualquiera de los valores por separado, en las listas enlazadas tendremos que recorrer toda la lista para llegar a la información que queremos.

```
function Node(data) {
  this.data = data;
  this.next = null;
}

function List() {
  this._length = 0;
  this.head = null;
}
```

▼ UNDERSCORE (dato curioso)

> **._length:** se utiliza el underscore como convensión social para señalar como privado el atributo de una clase. Es decir, que no se pueda acceder desde afuera de la clase.

Como base crearemos dos clases. Una clase **Node**, con los atributos **data** y **next**, y otra clase **List**, con los atributos **length** y **head**.

En las listas podremos iterar, y agregar y eliminar elementos al comienzo, al medio y al final.

```
/*function Node(data) {
    this.data = data;
    this.next = null;
  }

  function List() {
    this._length = 0;
    this.head = null;
  }*/

//MÉTODO PARA AGREGAR ELEMENTOS AL FINAL DE LA LISTA
List.prototype.add = function(data) {
  var node = new Node(data);     //node = {data: data, next: null}
  var current = this.head;
  if (!current) {
    this.head = node;
    this._length++;
    return node;
  }
  while (current.next) {
    current = current.next;
  }
  current.next = node;
  this._length++;
  return node;
};
```

Luego de crear las clases **Node** y **List**, haremos un método para la clase ***List*** a partir de su *prototype*. En este caso, como buscamos agregar elementos al final de la lista, el método se llamará **add**.

Al comienzo establecemos dos variables; **node**, que creará un nuevo nodo con datos, y **current**, que será una referencia al primer nodo de la lista.

Luego hacemos un condicional que, en caso de que no exista ningún nodo en nuestra lista, se encargará de crear el primero.

Después del condicional hay un ciclo *while*. En su argumento encontramos la variable **current**, que apunta al primer nodo. Utilizamos su propiedad **.next** para saber si existe un nodo luego del primero. De esta forma podremos recorrer toda la lista. Cuando el ciclo *while* termine, será porque estamos parados en un nodo que no tiene otro posterior.

Es aquí cuando se crea un nodo posterior al que estamos parados.

```
//MÉTODO PARA IMPRIMIR LA DATA DE CADA NODO
List.prototype.getAll = function(){
  var current = this.head;
  if(!current){
    console.log('La lista esta vacia!');
    return;
  }

  while(current){
    console.log(current.data);
    current = current.next;
  }

  return;
};
```

Aquí le creamos un nuevo método a nuestra lista; **getAll**.

En primera instancia, en el método se declara la variable **current = this.head**. Aquí entramos a un condicional que, en caso de no existir ningún elemento en nuestra lista, nos imprimirá "*La lista está vacía!*".

En caso de que no esté vacía, hará un recorrido con el ciclo *while*. Cada vez que pase por un nodo, imprimirá la data que este tenga adentro. Cuando lleguemos al último nodo, el ciclo *while* se terminará.

▼ HASH TABLE

Las **Hash Tables** son estructuras de datos que te permiten crear una lista de valores en par-clave. De este modo podremos retornar un valor a partir de su "contraseña" que tendremos de antemano. En otras palabras, una **Hash Function** le asignará un valor único a una variable, para que podamos obtenerla más tarde. (Esto está asociado a la encriptación).