


Árbol binario ejercicios

Completando el [ejercicio de la clase anterior, completo aquí](#)

js

 Copiar código

```
// Árbol binario
// Creamos la clase principal 'Nodo', con tres propiedades.
class Node {
  // 3 propiedades
  constructor(valor){
    this.valor = valor;
    this.izq = null;
    this.der = null;
  }
}

// Creamos la clase 'Árbol' con una sola propiedad
class Arbol{
  constructor(){
    this.raiz = null;
  }

  // Creamos un método a función de inserción a nuestra clase 'Arbol',
  Agregar(valor){
    const newNode = new Node(valor);
    if(this.raiz === null){
      this.raiz = newNode;
    }else{
      this.AgregarNode(this.raiz, newNode);
    }
  }

  // Creamos la función agregar a los nodos Izq o Der.
  AgregarNode(nodo, newNode){
    if(newNode.valor < nodo.valor){
      if(nodo.izq === null){
        nodo.izq = newNode;
      }else{
        this.AgregarNode(nodo.izq, newNode);
      }
    }else{
      if(nodo.der === null){
        nodo.der = newNode;
      }else{
        this.AgregarNode(nodo.der, newNode);
      }
    }
  }
}
```

```

// Función buscar
Buscar(valor){
    return this.BuscarNodo(this.raiz, valor)
}

// Función Buscar nodo
BuscarNodo(nodo, valor){
    if(nodo === null || nodo.valor === valor ){
        return nodo;
    }else if(valor < nodo.valor){
        return this.BuscarNodo(nodo.izq, valor)
    }else{
        return this.BuscarNodo(nodo.der, valor)
    }
}

// Función eliminar nodo
eliminar(valor){
    this.raiz = this.eliminarNodo(this.raiz, valor);
}

eliminarNodo(nodo, valor) {
    if (nodo === null) {
        return null;
    } else if (valor < nodo.valor) {
        nodo.izq = this.eliminarNodo(nodo.izq, valor);
        return nodo;
    } else if (valor > nodo.valor) {
        nodo.der = this.eliminarNodo(nodo.der, valor);
        return nodo;
    } else {
        if (nodo.izq === null && nodo.der === null) {
            return null;
        } else if (nodo.izq === null) {
            return nodo.der;
        } else if (nodo.der === null) {
            return nodo.izq;
        } else {
            const sucesor = this.encontrarSucesor(nodo.der);
            nodo.valor = sucesor.valor;
            nodo.der = this.eliminarNodo(nodo.der, sucesor.valor);
            return nodo;
        }
    }
}

// Función para encontrar el lugar donde ubicar a el valor eliminado
encontrarSucesor(nodo) {
    let sucesor = nodo;
    while (sucesor.izq !== null) {
        sucesor = sucesor.izq;
    }
    return sucesor;
}

// Recorrido en orden de un árbol binario
recorridoEnOrden() {
    this.recorrerEnOrden(this.raiz);
}

recorrerEnOrden(nodo) {
    if (nodo !== null) {
        this.recorrerEnOrden(nodo.izq);
        console.log(nodo.valor);
        this.recorrerEnOrden(nodo.der);
    }
}

```

```
// Recorrido en pre-orden
recorridoPreOrden() {
  this.recorrerPreOrden(this.raiz);
}
recorrerPreOrden(nodo) {
  if (nodo !== null) {
    console.log(nodo.valor);
    this.recorrerPreOrden(nodo.izq);
    this.recorrerPreOrden(nodo.der);
  }
}

// Recorrido en post-orden
recorridoPostOrden() {
  this.recorrerPostOrden(this.raiz);
}

recorrerPostOrden(nodo) {
  if (nodo !== null) {
    this.recorrerPostOrden(nodo.izq);
    this.recorrerPostOrden(nodo.der);
    console.log(nodo.valor);
  }
}
}

// comprobando la función Agregar()
const newArbol = new Arbol();
newArbol.Agregar("D");
console.log(newArbol);
newArbol.Agregar("B");
console.log(newArbol);
newArbol.Agregar("C");
console.log(newArbol);
newArbol.Agregar("a");
console.log(newArbol);
newArbol.Agregar("A");
console.log(newArbol);
newArbol.Agregar("c");
console.log(newArbol);
newArbol.Agregar("b");
console.log(newArbol);

// Comprobante de la función EliminarNodo()
console.log(newArbol.eliminar("C"));

// Comprobando recorrido en orden
console.log("Comprobando recorrido en orden");
console.log(newArbol.recorridoEnOrden());


// Comprobando recorrido en pre-orden
console.log("Comprobando recorrido en pre-orden");
console.log(newArbol.recorridoPreOrden());

// Comprobando recorrido en post-orden
console.log("Comprobando recorrido en post-orden");
console.log(newArbol.recorridoPostOrden());
```

Grafos

Los grafos en JavaScript son estructuras de datos que consisten en un conjunto de nodos (vértices) conectados entre sí por aristas (aristas). Pueden ser dirigidos o no dirigidos y se utilizan para representar una amplia variedad de relaciones, desde redes sociales hasta mapas de carreteras. En JavaScript, los grafos se pueden implementar utilizando objetos y arrays, donde los nodos son objetos y las aristas son arrays de nodos o parejas de nodos. Las operaciones comunes en grafos incluyen añadir y eliminar nodos y aristas, buscar caminos entre nodos, y realizar recorridos para explorar el grafo en diferentes maneras.

js


 Copiar código

```
// We create a class for the graph
class Graph{
  // The graph has only one property which is the adjacency list
  constructor() {
    this.adjacencyList = {}
  }
  // The addNode method takes a node value as parameter and adds it as a key to the adjacencyList
  addNode(node) {
    if (!this.adjacencyList[node]) this.adjacencyList[node] = []
  }
  // The addConnection takes two nodes as parameters, and it adds each node to the other's array
  addConnection(node1,node2) {
    this.adjacencyList[node1].push(node2)
    this.adjacencyList[node2].push(node1)
  }
  // The removeConnection takes two nodes as parameters, and it removes each node from the other's array
  removeConnection(node1,node2) {
    this.adjacencyList[node1] = this.adjacencyList[node1].filter(v => v !== node2)
    this.adjacencyList[node2] = this.adjacencyList[node2].filter(v => v !== node1)
  }
  // The removeNode method takes a node value as parameter. It removes all connections to that node
  removeNode(node){
    while(this.adjacencyList[node].length) {
      const adjacentNode = this.adjacencyList[node].pop()
      this.removeConnection(node, adjacentNode)
    }
    delete this.adjacencyList[node]
  }
}

const spain = new Graph()
spain.addNode("Cordoba")
spain.addNode("Sevilla")
spain.addNode("Toledo")
spain.addNode("Madrid")
spain.addConnection("Sevilla", "Cordoba")
spain.addConnection("Cordoba", "Toledo")
spain.addConnection("Toledo", "Madrid")
console.log(spain)
// Graph {
//   adjacencyList: {
```


```
//   Cordoba: [ 'Sevilla', 'Toledo' ],
//   Sevilla: [ 'Cordoba' ],
//   Toledo: [ 'Cordoba', 'Madrid' ],
//   Madrid: [ 'Toledo' ]
// }
// }
```

js

 Copiar código

```
// Ejemplo web
// We create a class for the graph
class Graph{
  // The graph has only one property which is the adjacency list
  constructor() {
    this.adjacencyList = {}
  }
  // The addNode method takes a node value as parameter and adds it as a key to the adjacencyList
  addNode(node) {
    if (!this.adjacencyList[node]) this.adjacencyList[node] = []
  }
  // The addConnection takes two nodes as parameters, and it adds each node to the other's array
  addConnection(node1,node2) {
    this.adjacencyList[node1].push(node2)
    this.adjacencyList[node2].push(node1)
  }
  // The removeConnection takes two nodes as parameters, and it removes each node from the other's
  removeConnection(node1,node2) {
    this.adjacencyList[node1] = this.adjacencyList[node1].filter(v => v !== node2)
    this.adjacencyList[node2] = this.adjacencyList[node2].filter(v => v !== node1)
  }
  // The removeNode method takes a node value as parameter. It removes all connections to that node
  removeNode(node){
    while(this.adjacencyList[node].length) {
      const adjacentNode = this.adjacencyList[node].pop()
      this.removeConnection(node, adjacentNode)
    }
    delete this.adjacencyList[node]
  }
}
const spain = new Graph()
spain.addNode("Cordoba")
spain.addNode("Sevilla")
spain.addNode("Toledo")
spain.addNode("Madrid")
spain.addConnection("Sevilla", "Cordoba")
spain.addConnection("Cordoba", "Toledo")
spain.addConnection("Toledo", "Madrid")
console.log(spain)
// Graph {
//   adjacencyList: {
//     Cordoba: [ 'Sevilla', 'Toledo' ],
//     Sevilla: [ 'Cordoba' ],
//     Toledo: [ 'Cordoba', 'Madrid' ],
//     Madrid: [ 'Toledo' ]
//   }
// }
```

js

 Copiar código

```

//Ejemplo 2
class Grafo{
  // Solo maneja una sola propiedad
  constructor(){
    this.adjacencyList = {};
  }

  // Creamos la función que tomará el valor del nodo como parámetro y lo agrega como
  // parámetro y lo agrega como clave a adjacencyList
  addVertices(node){
    if(!this.adjacencyList[node]) this.adjacencyList[node] = [];
  }
  // Toma dos nodos como parámetros y lo agrega cada nodo a la matrix de conexiones
  addArista(node1, node2){
    this.adjacencyList[node1].push(node2);
    this.adjacencyList[node2].push(node1); // Trabajando con nodos no dirigidos
  }
  // Va a tomar 2 nodos como parámetros y agrega cada nodo a la matrix de conexión
  addArista2(node1, node2){
    this.adjacencyList[node1].push(node2); // Trabajando con nodos dirigidos
  }
}

const lista = new Grafo();
lista.addVertices("A");
lista.addVertices("B");
lista.addVertices("C");
lista.addVertices("D");
lista.addVertices("E");
console.log("Grafo no dirigido");
console.log(lista);
// Grafo no dirigido
lista.addArista("A", "B");
lista.addArista("A", "C");
lista.addArista("C", "D");
lista.addArista("D", "E");
console.log(lista);

const lista2 = new Grafo();
lista2.addVertices("A");
lista2.addVertices("B");
lista2.addVertices("C");
lista2.addVertices("D");
lista2.addVertices("E");
console.log("Grafo dirigido");

// Grafo dirigido
lista2.addArista2("A", "B");
lista2.addArista2("B", "C");
lista2.addArista2("C", "E");
lista2.addArista2("E", "F");
lista2.addArista2("E", "D");
lista2.addArista2("D", "B");
console.log(lista2);


```

Afianza conocimientos sobre grafos

Grafos / Departamentos y Capitales

Realizar un grafo donde tengan departamentos de Colombia y deben crear un enlace con su respectiva ciudad, es dirigido:

js - Solución ejercicio:

 Copiar código

```
// Solución ejercicio:
class Grafo{
  // Solo maneja una sola propiedad
  constructor(){
    this.adjacencyList = {};
  }

  // Creamos la función que tomará el valor del nodo como parámetro y lo agrega como
  // parametro y lo agrega como clave a adjacencyList
  addVertices(node){
    if(!this.adjacencyList[node]) this.adjacencyList[node] = [];
  }
  // Toma dos nodos como parámetros y lo agrega cada nodo a la matrix de conexiones
  addArista(node1, node2){
    this.adjacencyList[node1].push(node2);
    this.adjacencyList[node2].push(node1); // Trabajando con nodos no dirigidos
  }
  // Va a tomar 2 nodos como parámetros y agrega cada nodo a la matrix de conexión
  addArista2(node1, node2){
    this.adjacencyList[node1].push(node2); // Trabajando con nodos dirigidos
  }
}

const departamento = new Grafo();

departamento.addVertices("Cundinamarca");
departamento.addVertices("Bogotá");
departamento.addVertices("Antioquia");
departamento.addVertices("Medellin");
departamento.addVertices("Valle del Cauca");
departamento.addVertices("Cali");
departamento.addVertices("Atlántico");
departamento.addVertices("Barranquilla");
```



```
departamento.addArista2("Cundinamarca", "Bogotá");
departamento.addArista2("Antioquia", "Medellín");
departamento.addArista2("Valle del Cauca", "Cali");
departamento.addArista2("Atlántico", "Barranquilla");
```

09.04.2024 MOD-2


```
console.log(departamento);
```

[Afianza conocimientos sobre grafos](#)

Grafos / BFS

El BFS (Breadth-First Search) o Búsqueda en Anchura es un algoritmo de recorrido de grafos que explora todos los nodos vecinos de un nodo dado antes de pasar a los nodos vecinos de esos nodos. Utiliza una estructura de datos cola para mantener un registro de los nodos por visitar. Es útil para encontrar el camino más corto en un grafo no ponderado y para descubrir la estructura de un grafo. En JavaScript, se puede implementar BFS utilizando una cola y un conjunto para registrar nodos visitados, y se puede aplicar en diversos escenarios como navegación web, recomendaciones de amigos en redes sociales, y resolución de problemas en inteligencia artificial, entre otros.

js - Solución ejercicio:

 Copiar código

```
// Ejemplo 2 - usando prototipos y ALGORITMO 'BFS' https://www.youtube.com/watch?v=_Yf8tneauJ8&t=
class Graph {
  constructor() {
    this.adjList = new Map();
  }

  addNode(node) {
    this.adjList.set(node, []);
  }

  addEdge(node, neighbor) {
    this.adjList.get(node).push(neighbor);
    this.adjList.get(neighbor).push(node);
  }

  bfs(startNode) {
    const visited = new Set();
```



```

const queue = [startNode];

visited.add(startNode);

while (queue.length !== 0) {
  const current = queue.shift();
  console.log(current);

  const neighbors = this.adjList.get(current);

  for (const neighbor of neighbors) {
    if (!visited.has(neighbor)) {
      visited.add(neighbor);
      queue.push(neighbor);
    }
  }
}

// Example Usage
const graph = new Graph();

graph.addNode('A');
graph.addNode('B');
graph.addNode('C');
graph.addNode('D');
graph.addEdge('A', 'B');
graph.addEdge('A', 'C');
graph.addEdge('B', 'D');

graph.bfs('A');
console.log(graph);
console.log(graph.bfs('A'));

```

[Afianza conocimientos sobre algoritmos BFS](#)


Grafos / DFS

El DFS (Depth-First Search) o Búsqueda en Profundidad es un algoritmo de recorrido de grafos que sigue una rama del árbol de expansión hasta llegar a un nodo hoja antes de retroceder y explorar las ramas no exploradas. Utiliza una estructura de datos pila o recursión para mantener un registro de los nodos por visitar. Es útil para encontrar ciclos en grafos, recorrer y buscar en estructuras de datos como árboles y grafos, y para la topología y ordenamiento topológico. En JavaScript, se puede implementar DFS de manera recursiva o iterativa, y se aplica en algoritmos

de búsqueda en bases de datos, resolución de laberintos, y sistemas de recomendación, entre otros.

CS-439-F-2021-F-MOD-2

js - Solución ejercicio:

 Copiar código

```
// Ejemplo 3 - usando prototipos y ALGORITMO 'DFS' https://www.youtube.com/watch?v=\_Yf8tneauJ8&t=
class Graph {
  constructor() {
    this.adjList = new Map();
  }

  addNode(node) {
    this.adjList.set(node, []);
  }

  addEdge(node, neighbor) {
    this.adjList.get(node).push(neighbor);
    this.adjList.get(neighbor).push(node);
  }

  dfs(node, visited = new Set()) {
    console.log(node);
    visited.add(node);

    const neighbors = this.adjList.get(node);

    for (const neighbor of neighbors) {
      if (!visited.has(neighbor)) {
        this.dfs(neighbor, visited);
      }
    }
  }
}

// Example Usage
const graph = new Graph();

graph.addNode('A');
graph.addNode('B');
graph.addNode('C');
graph.addNode('D');
graph.addEdge('A', 'B');
graph.addEdge('A', 'C');
graph.addEdge('B', 'D');


graph.dfs('A');
```

[Afianza conocimientos sobre algoritmos DFS](#)

Algoritmos / búsqueda lineal

La búsqueda lineal, también conocida como búsqueda secuencial, es un método simple y directo para encontrar un elemento en una lista o array. En JavaScript, este algoritmo recorre el array elemento por elemento hasta encontrar el valor buscado o llegar al final del array. Es sencillo de implementar, pero no es el más eficiente para listas grandes. Su complejidad es $O(n)$, donde n es el número de elementos en el array. Aunque no es el algoritmo más rápido, es útil cuando no se conoce la estructura o cuando la lista no está ordenada.

js - Solución ejercicio:


 Copiar código

```
// Algoritmos de búsqueda lineal
let array1 = [1,2,3,4,5,6,7,8,9];

function BusquedaLineal(array1, val){
  for(let i = 0; i < array1.length; i++){
    if(array1[i] == val){
      return i;
    }
  }
  return -1;
}

console.log(BusquedaLineal(array1, 9));
```

js - Solución ejercicio:

 Copiar código

```
// Algoritmos de búsqueda lineal forma 2 -error!

let arregloLetras = ["a", "b","c","d","e","f",]
function buscador(elem, arregloLetras){
  for(let i = in arregloLetras){
    if(arregloLetras[i] == elem) return i;
  }
  return -1;
}

console.log(buscador("d", arregloLetras));
```

