Estructuras Implementadas

Árbol Binario

La clase tiene un atributo llamado “raíz” en el que se insertará el primer nodo.

```
constructor() {
    this.raiz = null
    this.dot = ''
    this.id = 0
}
```

Cada nodo tiene apuntadores izquierda y derecha, en los que se insertarán valores mayores o menores a la raíz respectivamente.

```
insert(nuevo) {
    this.raiz = this.add(nuevo,this.raiz)
    this.id ++
}
add(nuevo,actual) {
    if(!actual) {
        return new NodoA(nuevo,this.id)
    }
    if(nuevo.dni > actual.objeto.dni) {
        actual.derecha = this.add(nuevo,actual.derecha)
    }else if(nuevo.dni < actual.objeto.dni) {
        actual.izquierda = this.add(nuevo,actual.izquierda)
    }
    return actual
}
```

La inserción de nodos en el árbol se hace de manera recursiva. El grafo del árbol binario es generado recursivamente para garantizar que se pase por todas las ramas del árbol

```
getBranchesDot(actual) {
    let etiqueta = ''
    if(!actual.izquierda && !actual.derecha) {
        etiqueta = `nodo${actual.id}[label="${actual.objeto.dni}\\n${actual.objeto.nombre_actor}"];`
    }else {
        etiqueta = `nodo${actual.id}[label="<C0> | ${actual.objeto.dni}\\n${actual.objeto.nombre_actor} | <C1>"];`
    }
    if(actual.izquierda) {
        etiqueta += `${this.getBranchesDot(actual.izquierda)}nodo${actual.id}:C0 -> nodo${actual.izquierda.id};`
    }
    if(actual.derecha) {
        etiqueta += `${this.getBranchesDot(actual.derecha)}nodo${actual.id}:C1 -> nodo${actual.derecha.id};`
    }
    return etiqueta
}
getDot() {
    return `digraph G{rankdir=TB;node [shape = record];${this.getBranchesDot(this.raiz)}}`
}
```

Árbol AVL

El constructor de la clase es igual al del árbol binario, la única diferencia del árbol AVL es que está balanceado, es decir que los subárboles de cada nodo únicamente pueden estar en diferentes alturas teniendo un máximo de 1 de diferencia.

```
    insertElement(nuevo,nodo) {
        if(!nodo) {
            return new NodoA(nuevo,this.id)
        }
        if(nuevo.id_pelicula < nodo.objeto.id_pelicula) {
            nodo.izquierda = this.insertElement(nuevo,nodo.izquierda)
            if(this.getHeight(nodo.izquierda) - this.getHeight(nodo.derecha) == 2) {
                if(nuevo.id_pelicula < nodo.izquierda.objeto.id_pelicula) {
                    nodo = this.rotateWithLeftChild(nodo)
                }else {
                    nodo = this.doubleWithLeftChild(nodo)
                }
            }
        }else if(nuevo.id_pelicula > nodo.objeto.id_pelicula) {
            nodo.derecha = this.insertElement(nuevo,nodo.derecha)
            if(this.getHeight(nodo.derecha) - this.getHeight(nodo.izquierda) == 2) {
                if(nuevo.id_pelicula > nodo.derecha.objeto.id_pelicula) {
                    nodo = this.rotateWithRightChild(nodo)
                }else {
                    nodo = this.doubleWithRightChild(nodo)
                }
            }
        }
        nodo.altura = this.getMaxHeight(this.getHeight(nodo.izquierda),this.getHeight(nodo.derecha)) + 1
        return nodo
    }
```

Para mantener balanceado el árbol es necesario realizar rotaciones a medida que se ingresan los nodos.

Lista Simple

El primer nodo que se inserta se almacenará en el atributo primero de la clase.

```
    constructor() {
        this.primero = null
        this.ultimo = null
        this.indice = 0
    }
```

El método de inserción consiste en almacenar el nuevo nodo en el apuntador siguiente del último nodo insertado. Luego el atributo último se le reasigna el valor de su apuntador siguiente.

```
    add(nuevo) {
        if(this.primero) {
            this.ultimo.siguiente = new NodoS(this.indice,nuevo)
            this.ultimo = this.ultimo.siguiente
            this.indice ++
            return
        }
        this.primero = new NodoS(this.indice,nuevo)
        this.ultimo = this.primero
        this.indice ++
    }
```

Tabla Hash

Se reutiliza el concepto de lista enlazada simple, con la diferencia de que sus nodos son insertados desde la instanciación recibiendo sus dimensiones iniciales, además recibe un parámetro que determina si se aplica rehasing o no.

```
    constructor(n,isrehashing) {
        this.primero = null
        this.ultimo = null
        this.llenos = 0
        this.size = n
        this.isrehashing = isrehashing
        for(let i = 0; i < n; i ++) {
            this.add(i)
        }
    }
```

```
    add(nuevo) {
        if(this.primero) {
            this.ultimo.siguiente = new NodoH(nuevo)
            this.ultimo = this.ultimo.siguiente
            return
        }
        this.primero = new NodoH(nuevo)
        this.ultimo = this.primero
    }
```

Los nodos que almacena tienen un atributo de acceso que es un apuntador en el que almacena una lista enlazada simple, es idéntico a una lista de listas. En el acceso es en donde se insertarán los objetos requeridos.

El método de inserción consiste en calcular el resultado de la división modular entre el identificador del objeto y el tamaño de la tabla hash. No puede haber identificadores repetidos por lo que se verifica que no haya coincidencia de identificadores, en el caso de que no esté repetido el nuevo identificador se inserta el objeto en su lista correspondiente, en caso contrario se ignora el nuevo nodo y no se inserta. Si se habilita el rehashing, luego de que se ocupe el 75% (en este caso) de la tabla respecto a su tamaño se insertan cinco posiciones más a la tabla.

```
    insert(nuevo) {
        let nuevoHash = this.search(nuevo.id_categoria % this.size)
        if(!nuevoHash.acceso.verifyId(nuevo.id_categoria)) {
            nuevoHash.acceso.add(nuevo)
            this.llenos ++
            if(this.isrehashing) {
                if(this.llenos / this.size >= 0.75) {
                    this.hashing(5)
                }
            }
        }
    }
```

El método search se utiliza para buscar y obtener el nodo con el índice igual a la división modular entre el identificador y el tamaño de la tabla para insertar el nuevo objeto.

```
    search(indice) {
        let actual = this.primero
        while(actual) {
            if(actual.indice == indice) {
                return actual
            }
            actual = actual.siguiente
        }
    }
```

El método hashing agrega n posiciones a la tabla.

```
    hashing(n) {
        for(let i = this.size; i < this.size + n; i ++) {
            this.add(i)
        }
        this.size += n
    }
```

Árbol Merkle

El constructor es igual al de los árboles anteriores con la única diferencia que tiene una lista para almacenar la data que se ingresarán en los nodos del último nivel.

```
    constructor() {
        this.raiz = null
        this.id = 0
        this.list_data = new LstDt()
    }
```

El método de inserción de la data es el mismo que el de una lista simple.

```
    insert(nuevo) {
        this.list_data.insert(nuevo)
    }
```

El método construye la estructura del árbol siempre que existan datos almacenados en la data de la clase. El árbol debe estar completo por lo que la cantidad de data debe ser potencia de 2, en caso de que no sea potencia de dos se busca el próximo número que si lo sea y se agregan los nodos restantes con valores del índice multiplicado por 100. En caso de que solo se inserte un dato automáticamente se completa el árbol y el nodo hermano contendrá el valor de 100.

```
    buildTree() {
        if(this.list_data.primero) {
            let tmp = this.list_data.getNData()
            if(!this.isPowerOfTwo(tmp)) {
                tmp = this.getNextPowerOfTwo(tmp)
            }
            if(tmp == 1) {
                this.niveles = 1
                tmp = 2
            }
            for(let i = this.list_data.getNData(); i < tmp; i ++) {
                this.list_data.insert(new NodoN(i*100))
            }
            this.niveles = this.getPowerOfTwo(tmp)
            this.buildBranches()
        }
    }
```

La cantidad de niveles está definido por la función $niveles = \log_2 N$, donde N es la cantidad potencia de 2 de nodos almacenados en la lista data.

```
    getPowerOfTwo(n) {
        let power = 0
        while(n % 2 == 0) {
            n /= 2
            power ++
        }
        return power
    }
```

Los nodos internos del árbol se insertan de manera recursiva, bajan de la raíz hacia los hijos hasta encontrar un apuntador desocupado, también se inserta hasta que se alcanza la cantidad de niveles establecidos anteriormente. Y se define el atributo hash de cada nodo interno como el hash de la concatenación del hash de sus dos hijos.

```
buildBranches() {
    this.raiz = this.addNodesBranch(this.raiz,0)
}
addNodesBranch(nodo,nivel) {
    if(!nodo) {
        this.id ++
        if(nivel < this.niveles) {
            nodo = new NodoM(this.id,nivel)
            nodo.izquierda = this.addNodesBranch(nodo.izquierda,nivel + 1)
            nodo.derecha = this.addNodesBranch(nodo.derecha,nivel + 1)
            nodo.hash = Sha256.hash(`${nodo.izquierda.hash}${nodo.derecha.hash}`)
        }else if(nivel == this.niveles) {
            nodo = this.addLeafNode(nodo,nivel)
        }
    }
    return nodo
}
```

Al llegar al último nivel se agregan los nodos hoja los cuales almacenaran la data de sus correspondientes datos almacenados en la lista data, de modo que se vaya extrayendo el nodo cabeza de la lista por cada nodo hoja insertado, la estructura de la data es la de una cola, el hash del nodo hoja se calcula mediate el hash de la concatenación de la data insertada en el nodo (“nombreCliente - nombrePelicula”). Los hashes de los nodos hoja se utilizarán para calcular el hash del nodo padre.

```
addLeafNode(nodo,nivel) {
    nodo = new NodoM(this.id,nivel)
    let head = this.list_data.getHead()
    if(head) {
        nodo.hash = head.objeto.hash
        nodo.data = {id: head.id_data,transaccion: head.objeto}
    }
    return nodo
}
```

Blockchain

Para la Blockchain se implementó una estructura de lista doblemente enlazada

```
    constructor() {
        this.primero = null
        this.ultimo = null
    }
```

Al insertar un nuevo bloque (nodo) se almacena en el atributo siguiente del último bloque insertado, posteriormente se asigna como anterior del bloque siguiente (nuevo bloque) al último bloque insertado, por último se reasigna el valor del último bloque como igual a su atributo siguiente (nuevo bloque). Al primer bloque insertado se le asigna como hash del bloque previo ‘00’, para los siguientes bloques se toma el atributo hash del bloque anterior para asignarlo al atributo prev_hash del nuevo bloque. Para calcular el hash de un nuevo bloque se realiza la prueba de trabajo.

```
    insert(nuevo) {
        if(this.primero) {
            this.ultimo.siguiente = new NodoBC(nuevo)
            this.ultimo.siguiente.anterior = this.ultimo
            this.ultimo = this.ultimo.siguiente
            this.ultimo.objeto.prev_hash = this.ultimo.anterior.objeto.hash
            let proof = this.proofOfWork(this.ultimo.objeto,0)
            this.ultimo.objeto.nonce = proof.nonce
            this.ultimo.objeto.hash = proof.hash
            return
        }
        this.primero = new NodoBC(nuevo)
        this.primero.objeto.prev_hash = '00'
        let proof = this.proofOfWork(this.primero.objeto,0)
        this.primero.objeto.nonce = proof.nonce
        this.primero.objeto.hash = proof.hash
        this.ultimo = this.primero
    }
```

La prueba de trabajo se realiza iterando un entero n que cumpla con la siguiente condición:
Si el resultado de realizar el hash de la concatenación del índice del bloque, la hora y fecha de generación del bloque (con formato específico), el hash del bloque previo, el hash de la raíz del árbol merkle que almacena y el número n es igual a una cadena cuyos primeros dos caracteres sean ceros. De lo contrario se sigue iterando hasta que se cumpla la condición.

```
proofOfWork(nodo,n) {
    let hash
    do {
        hash = Sha256.hash(`${nodo.index}${nodo.timestamp}${nodo.prev_hash}${nodo.root_merkle}${n}`)
        if(hash.toString().charAt(0) == 0 && hash.toString().charAt(1) == 0) {
            return {nonce: n,hash: hash}
        }
        n ++
    } while(true)
}
```