1. Nodos

La clase Nodo (genérico) se utilizó para almacenar los objetos de las listas enlazadas mediante apuntadores, dependiendo del tipo de lista cambian los atributos de la clase.

```
class Nodo {
    constructor(indice,objeto) {
        this.indice = indice
        this.objeto = objeto
        this.siguiente = null
        this.anterior = null
    }
}
```

2. Objetos

Los objetos se utilizaron para almacenar la información de sus contrapartes en los archivos de entrada (usuario, libro, autor) o bien en métodos que generan la creación de nuevos objetos como para compras o para la cola de disponibilidad.

```
class Libro {
    constructor(isbn,nombre_autor,nombre_libro,cantidad,fila,columna,paginas,categoria) {
        this.isbn = isbn
        this.nombre_autor = nombre_autor
        this.nombre_libro = nombre_libro
        this.cantidad = cantidad
        this.fila = fila
        this.columna = columna
        this.paginas = paginas
        this.categoria = categoria
    }
}
```

3. Listas Enlazadas

Las listas se usaron como estructuras de almacenamiento mediante la implementación de clases nodo con punteros los cuales almacenaban los objetos siguientes o anteriores dependiendo del tipo de lista. Los constructores tienen atributos primero y último según sea requerido.

3.1. Listas Simples

Los nodos que almacenan los apuntadores únicamente apuntan a un siguiente objeto.

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

3.2. Listas Doblemente Enlazadas

Los nodos que almacenan los punteros apuntan a un siguiente objeto y a un anterior, únicamente los apuntadores primeros y últimos tienen un solo apuntador, siguiente y anterior respectivamente.

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

3.3. Listas Doblemente Enlazadas Circulares

Los nodos interiores (después del primero y antes del último) que almacenan los apuntadores apuntan a un siguiente objeto y a un anterior. El apuntador primero tiene como apuntador anterior al nodo último y el nodo último tiene como apuntador siguiente al nodo primero.

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

4. 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(this.raiz,nuevo,0)
        this.id ++
    }
    add(actual,nuevo,nivel) {
        if(!actual) {
            return new NodoAB(nuevo,nivel,this.id)
        }
        if(nuevo.nombre_autor > actual.objeto.nombre_autor) {
            actual.derecha = this.add(actual.derecha,nuevo,nivel + 1)
        }else {
            actual.izquierda = this.add(actual.izquierda,nuevo,nivel + 1)
        }
        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.nombre_autor}"];`
      }else {
          etiqueta = `nodo${actual.id} [label="<C0> | ${actual.objeto.nombre_autor} | <C1>"];`
      }
      if(actual.izquierda) {
          etiqueta += `${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)}}`
  }
```

5. Pila

La clase tiene un atributo llamado primero que será siempre el último objeto que se inserte y último que será el que almacene auxiliarmente al primero cuando se sustituya el por el nuevo valor insertado.

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

Para insertar se utilizó la estructura de una lista simple enlazada, únicamente se cambió el concepto de inserción, a diferencia de la lista que inserta sus nuevos nodos en el apuntador siguiente de su último nodo insertado la pila inserta en el primero, el siguiente del primero pasa a ser el último y el último nuevamente es el primero.

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

Para eliminar se utiliza el método pop que lo que hace únicamente es reestablecer valores, es decir el primero pasa a ser el siguiente del actual y el actual se descarta.

```
pop() {
    if(this.primero) {
        let primero = this.primero
        this.primero = this.primero.siguiente
        this.indice --
        return primero
    }
    return null
}
```

6. Cola

La clase tiene un atributo llamado primero que será el punto de acceso y uno último cuyo puntero siguiente es el que almacena cada nuevo nodo insertado.

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

Para insertar un nuevo nodo se utilizó la estructura de una lista doblemente enlazada, ya que cada nuevo nodo insertado (último) debe referenciar a su antecesor (anterior) ya que debe ir detrás de él sin importar quién viene detrás (siguiente). El atributo siguiente se implementó con el propósito de recorrer la lista desde el punto de acceso primero hacia el último.

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

7. Matrices

Para la implementación de las matrices se utilizó el concepto de listas de listas. Ya que cada nodo insertado pertenece a una determinada fila y columna se crean sus cabeceras como identificadores (índices) los cuales tendrán un apuntador de acceso a la lista que contiene a cierta fila o columna. Cada nuevo nodo insertado (en la matriz) se inserta nuevamente en su respectiva lista de fila, según su identificador, y en una de columna, es decir que cada nodo es insertado en dos listas cabeceras de la matriz, en la de fila y en la de columna, de modo que se asignan los nodos a sus apuntadores siguiente y anterior y arriba y abajo.

```
    insert(objeto) {
        let nodoInterno = new NodoInterno(objeto.fila,objeto.columna,objeto)
        let encabezadoX = this.filas.getHeader(nodoInterno.x)
        let encabezadoY = this.columnas.getHeader(nodoInterno.y)
        if(!encabezadoX) {
            encabezadoX = new NodoEncabezado(nodoInterno.x)
            this.filas.insertHeader(encabezadoX)
        }
        if(!encabezadoY) {
            encabezadoY = new NodoEncabezado(nodoInterno.y)
            this.columnas.insertHeader(encabezadoY)
        }
        if(!encabezadoX.acceso) {
            encabezadoX.acceso = nodoInterno
        }else {
            if(nodoInterno.y < encabezadoX.acceso.y) {
                nodoInterno.derecha = encabezadoX.acceso
                encabezadoX.acceso.izquierda = nodoInterno
                encabezadoX.acceso = encabezadoX.acceso.izquierda
            }else {
                let actual = encabezadoX.acceso
                while(actual) {
                    if(nodoInterno.y < actual.y) {
                        nodoInterno.derecha = actual
                        nodoInterno.izquierda = actual.izquierda
                        actual.izquierda.derecha = nodoInterno
                        actual.izquierda = actual.izquierda.derecha
                        break
                    }else {
                        if(!actual.derecha) {
                            actual.derecha = nodoInterno
                            actual.derecha.izquierda = actual
                            break
                        }else {
                            actual = actual.derecha
                        }
                    }
                }
            }
        }
        if(!encabezadoY.acceso) {
            encabezadoY.acceso = nodoInterno
        }else {
            if(nodoInterno.x < encabezadoY.acceso.x) {
                nodoInterno.abajo = encabezadoY.acceso
                encabezadoY.acceso.arriba = nodoInterno
                encabezadoY.acceso = encabezadoY.acceso.arriba
            }else {
                let actual2 = encabezadoY.acceso
                while(actual2) {
                    if(nodoInterno.x < actual2.x) {
                        nodoInterno.abajo = actual2
                        nodoInterno.arriba = actual2.arriba
                        actual2.arriba.abajo = nodoInterno
                        actual2.arriba = actual2.arriba.abajo
                        break
                    }else {
                        if(!actual2.abajo) {
                            actual2.abajo = nodoInterno
                            actual2.abajo.arriba = actual2
                            break
                        }else {
                            actual2 = actual2.abajo
                        }
                    }
                }
            }
        }
    }
```

8. Algoritmos de Ordenamiento

8.1. Bubble Sort

Para un ordenamiento de este tipo es necesario únicamente un bucle anidado dentro de otro, esto con el fin de garantizar llevar el número, ya sea más grande o más pequeño, hasta el final de la lista.

La implementación se hizo con dos funciones recursivas que funcionaron como los dos bucles y la asignación de índices a cada nodo de la lista al momento de insertarlos. De modo que se necesitaba ejecutar recursivamente las funciones dentro de la cantidad de nodos de manera exacta. El intercambio se realiza cuando se cumpla la condición, si se cumple se guarda temporalmente el nodo actual y se reasigna el nodo actual como el nodo siguiente y el nodo siguiente se reasigna como la variable temporal.

```
 bubbleSortByName() {
    this.bubbleSortR1(this.primero)
}
bubbleSortR1(nodoI) {
    if(nodoI.indice < this.ultimo.indice) {
        this.bubbleSortR2(nodoI,this.primero)
        this.bubbleSortR1(nodoI.siguiente)
    }
}
bubbleSortR2(nodoI,nodoX) {
    if(nodoX.indice < this.ultimo.indice - nodoI.indice) {
        if(nodoX.objeto.nombre_libro > nodoX.siguiente.objeto.nombre_libro) {
            let temporal = nodoX.objeto
            nodoX.objeto = nodoX.siguiente.objeto
            nodoX.siguiente.objeto = temporal
        }
        this.bubbleSortR2(nodoI,nodoX.siguiente)
    }
}
```

8.2. Quick Sort

Para este ordenamiento es necesario avanzar hacia adelante (al nodo siguiente) y hacia atrás (al nodo anterior) por lo que es necesario que sea una lista doblemente enlazada para poder aplicar el algoritmo. Si el índice del nodo izquierdo es menor al del nodo derecho (último y primero en primera instancia) se hará una partición tomando como pivote el valor a comparar del nodo izquierdo y luego se hace una doble llamada recursiva reasignando parámetros izquierdo y derecho del nodo de donde se hará la partición. Las particiones se hacen de forma recursiva. Se reasignará el nodo izquierdo a su siguiente hasta que el valor del nodo izquierdo sea menor al del pivote. Se reasignará el nodo derecho a su anterior hasta que el valor del nodo derecho sea menor al del pivote. Si el índice de izquierda es mayor o igual al de derecha se retorna el nodo derecha, de lo contrario se hará un intercambio y se repetirá el proceso de partición con el mismo pivote, nodo izquierdo siguiente y nodo derecho anterior.

```
quickSortByName() {
    this.quickSortR1(this.primero,this.ultimo)
}
quickSortR1(izquierda,derecha) {
    if(izquierda.indice < derecha.indice) {
        let nodoParticion = this.partition(izquierda.objeto.nombre_libro,izquierda,derecha)
        this.quickSortR1(izquierda,nodoParticion)
        this.quickSortR1(nodoParticion.siguiente,derecha)
    }
}
partition(pivote,izquierda,derecha) {
    while(izquierda.objeto.nombre_libro > pivote) {
        izquierda = izquierda.siguiente
    }
    while(derecha.objeto.nombre_libro < pivote) {
        derecha = derecha.anterior
    }
    if(izquierda.indice >= derecha.indice) {
        return derecha
    }
    let temporal = izquierda.objeto
    izquierda.objeto = derecha.objeto
    derecha.objeto = temporal
    return this.partition(pivote,izquierda.siguiente,derecha.anterior)
}
```