Biblia de Algoritmos y Estructuras de Datos

Santiago Aguerre
Agostina Etchebarren
Thiago García
Facundo Píriz
Santiago Blanco
Eugenia Machado
Agustín García
Eugenia Guibernau
Diego De Olivera

Universidad Católica del Uruguay
2025

ÍNDICE

TDA LISTA ENLAZADA
- **CRUD**
- **Eliminar duplicados**

TDA COLA
- **CRUD**

TDA PILA:
- **Insertar corchetes**
- **modificarTope**

TDA ÁRBOLES GENÉRICOS
- **recorrerProfundidad**
- **agregarHijo**
- **buscar**
- **calcularGrado**
- **preOrden**
- **calcularAltura**

TDA ÁRBOL BINARIO
- **CRUD**
- **Get Tamaño**
- **Get Profundidad**
- **Get Altura**
- **Get Nivel**
- **Get Hojas**
- **Cantidad de Hojas**
- **Nodos completos**
- **Nodos internos**
- **Cant nodos de un nivel**
- **Eliminar**

TDA TRIE
- **InsertarNodo**
- **BuscarNodo**
- **BuscarPrefijo**
- **BuscarPalabra**
- **eliminar**
- **imprimir**
- **construirIndice**
-

TDA ARBOL BINARIO DE BUSQUEDA:
- **insertarOrdenado**
- **buscar**
- **eliminar**
- **minimo,max**

TDA Lista Enlazada Operaciones básicas
Agregar, buscar, modificar, eliminar

TDAListaEnlazada.Agregar(Nodo nodo, int posición) //O(n)
	Precondición: posición >= 0, posición <= largo lista, nodo != vacío
	Postcondición: nodo es insertado en la posición, el nodo en la posición - 1 referencia al nodo en posición +1, el nodo no referencia a ningún nodo, largo aumenta en 1

	Implementación
	i <- 0 //O(1)
	actual <- primero //O(1)

	Mientras i < posición -1: //O(n)
		actual <- actual.siguiente //O(1)
		i <- i + 1 //O(1)
	Fin Mientras
	Si i != 0:
		nodo.siguiente <- actual.siguiente //O(1)
		actual.siguiente <- nodo //O(1)
	Sino:
		nodo.siguiente <- primero //O(1)
		primero <- nodo //O(1)
	Fin Si
Fin Agregar

ListaEnlazadaTDA.Buscar(int posición) //O(n)
	Precondición: posición >= 0, posición <= largo lista
	Postcondición: devuelve la carga del elemento en la posición

	Implementación
	i <- 0 //O(1)
	actual <- primero //O(1)

	Mientras i < posición: //O(n)
		actual <- actual.siguiente //O(1)
		i <- i + 1 //O(1)
	Fin Mientras

	Devolver actual.carga //O(1)
Fin Buscar

```
De tipo TLista Ordenar
   COM
   Lista2 ← nuevaLista

   Mientras no (vacia()) hacer
      elementoActual ← Eliminar(primero)
      Lista2.InsertarOrdenado(elementoActual)
   finMientras

   devolver Lista2
FIN
```

```
TDAListaEnlazada.Modificar(Nodo nodo, int posición)  //O(n)
        Precondición: posición >= 0, posición <= largo lista, nodo != vacío
        Postcondición: la carga del nodo en la posición es modificada

        Implementación
        i <- 0   //O(1)
        actual <- primero  //O(1)

        Mientras i < posición:  //O(n)
                actual <- actual.siguiente  //O(1)
                i <- i + 1  //O(1)
        Fin Mientras

        actual.carga <- nodo.carga   //O(1)
Fin Modificar

TDAListaEnlazada.Eliminar(int posición)  //O(n)
        Precondición: posición >= 0, posición <= largo lista
        Postcondición: eliminar el nodo en la posición, el siguiente del nodo anterior
al nodo eliminado tiene que pasar a ser el siguiente del nodo eliminado, largo
disminuye en 1


   Si posición = 0:
      primero <- primero.siguiente  //O(1)
   Sino:
      i <- 0   //O(1)
      actual <- primero  //O(1)

      Mientras i < posición - 1:  //O(n)
         actual <- actual.siguiente  //O(1)
         i <- i + 1  //O(1)
      Fin Mientras

      aEliminar <- actual.siguiente   //O(1)
      actual.siguiente <- aEliminar.siguiente //O(1)
      aEliminar.siguiente <- vacío
   Fin Si
Fin Eliminar
```

//O(n cuadrado)

- UnaLista.EliminaDuplicados
  //elimina los elementos con etiqueta duplicada de la lista
  nodoActual, otroNodo de Tipo Elemento

  COMIENZO
  nodoActual ← UnaLista.Primero
  Mientras nodoActual <> nulo hacer
  otroNodo ← nodoActual.Siguiente
  mientras otroNodo <> nulo hacer
  si otroNodo.Etiqueta = nodoActual.Etiqueta
  entonces UnaLista.Elimina(otroNodo)
  si no otroNodo ← otroNodo.Siguiente
  fin si
  fin mientras
  nodoActual ← nodoActual.Siguiente
  Fin mientras
  FIN

LISTAS: Algoritmos de ejemplo con seudocódigo TDA

- Lista.ImprimirEtiquetas
  //imprime las etiquetas de todos los elementos, del primero al último.

  nodoActual de TipoNodoLista
  COMIENZO
  nodoActual ← UnaLista.Primero
  Mientras nodoActual <> nulo hacer
  nodoActual.Etiqueta.Imprimir
  nodoActual ← nodoActual.Siguiente
  Fin mientras
  FIN

1. Ordenación por Intercambio (Bubble Sort)**

```
FUNCIÓN ordenarPorIntercambio(lista: Lista) -> Lista:
    PARA i DESDE 0 HASTA longitud(lista) - 1 HACER:
        PARA j DESDE 0 HASTA longitud(lista) - 2 - i HACER:
            SI lista[j] > lista[j + 1] ENTONCES:
```

```
            intercambiar(lista[j], lista[j + 1])
        FIN SI
    FIN PARA
  FIN PARA
  DEVOLVER lista
FIN FUNCIÓN
```
Explicación:
- Recorre la lista múltiples veces, comparando elementos adyacentes y **intercambiándolos** si están en desorden.
- En cada pasada, el elemento más grande "burbujea" hacia el final.

---

2. Ordenación por Selección (Selection Sort)
```
FUNCIÓN ordenarPorSeleccion(lista: Lista) -> Lista:
  PARA i DESDE 0 HASTA longitud(lista) - 1 HACER:
    min_idx = i
    PARA j DESDE i + 1 HASTA longitud(lista) - 1 HACER:
      SI lista[j] < lista[min_idx] ENTONCES:
        min_idx = j
      FIN SI
    FIN PARA
    intercambiar(lista[i], lista[min_idx])
  FIN PARA
  DEVOLVER lista
FIN FUNCIÓN
```
Explicación:
- En cada iteración, **selecciona el elemento más pequeño** del subarreglo no ordenado.
- Lo coloca en su posición correcta **intercambiándolo** con el primer elemento del subarreglo no ordenado.

---

3. Ordenación por Inserción (Insertion Sort)
```
FUNCIÓN ordenarPorInsercion(lista: Lista) -> Lista:
  PARA i DESDE 1 HASTA longitud(lista) - 1 HACER:
    clave = lista[i]
    j = i - 1
    MIENTRAS j >= 0 Y lista[j] > clave HACER:
      lista[j + 1] = lista[j]
      j = j - 1
    FIN MIENTRAS
    lista[j + 1] = clave
  FIN PARA
  DEVOLVER lista
```

FIN FUNCIÓN
```
**Explicación:**
- Toma un elemento de la lista (empezando desde el segundo) y lo **inserta en la posición correcta** dentro del subarreglo ya ordenado.
- Desplaza los elementos mayores hacia la derecha para hacer espacio.

**FUNCION seleccionDirecta(lista)**
    listaOrdenada ← lista vacía

    MIENTRAS lista NO esté vacía HACER
        menor ← lista[0]
        posicion ← 0

        PARA i DESDE 1 HASTA longitud(lista) - 1 HACER
            SI lista[i] < menor ENTONCES
                menor ← lista[i]
                posicion ← i
            FIN SI
        FIN PARA

        // Quitar el menor de la lista original
        ELIMINAR lista[posicion]

        // Insertar al final de la lista ordenada
        AGREGAR menor A listaOrdenada
    FIN MIENTRAS

    RETORNAR listaOrdenada
FIN FUNCION

📘 Explicación rápida:
Se crea una lista vacía llamada listaOrdenada.

Mientras haya elementos en la lista original:

Se busca el menor elemento.

Se quita de la lista original.

Se agrega al final de listaOrdenada.

Cuando se vacía la lista original, se devuelve la ordenada.
```

FUNCION insercionDirecta(lista)
    listaOrdenada ← lista vacía

    MIENTRAS lista NO esté vacía HACER
        elemento ← lista[0]
        ELIMINAR lista[0]

        // Buscar la posición correcta en listaOrdenada
        posicion ← 0
        MIENTRAS posicion < longitud(listaOrdenada) Y listaOrdenada[posicion] < elemento HACER
            posicion ← posicion + 1
        FIN MIENTRAS

        // Insertar el elemento en la posición correcta
        INSERTAR elemento EN listaOrdenada EN posicion
    FIN MIENTRAS

    RETORNAR listaOrdenada
FIN FUNCION

🧠 ¿En qué se diferencia de selección directa?
Selección directa busca el mínimo de toda la lista en cada pasada.

Inserción directa toma el primer elemento disponible y lo inserta en el lugar correcto dentro de una lista que ya está en orden parcial.

COLA CIRCULAR

```
PoneEnCola(unElemento) //O(1)
Precondiciones: cola no llena, unElemento != nulo
Postcondiciones: se agrega unElemento a la cola
COMIENZO
Si primero = nulo entonces
        primero <- unElemento
        ultimo <- unElemento
        ultimo.siguiente <- primero
Sino
        ultimo.siguiente <- unElemento
        ultimo <- ultimo.siguiente
        ultimo.siguiente <- primero
FinSi
FIN

QuitaDeCola()//O(1)
Precondiciones:
Postcondiciones: Se devuelve el elemento eliminado
COMIENZO
Si primero = nulo entonces
        devuelve nulo
FinSi

actual <- primero

Si primero = ultimo entonces
        primero <- nulo
        ultimo <- nulo
Sino
        primero <- primero.siguiente
        ultimo.siguiente <- primero
FinSi
devuelve actual
FIN
```

PILA

Push(pila, elemento)//O(1)

Precondición: elemento ≠ vacío
Postcondición: el elemento se agrega al tope de la pila.

```
    nodo <- nuevo Nodo
    nodo.carga <- elemento
    nodo.siguiente <- pila.tope
    pila.tope <- nodo
    pila.largo <- pila.largo + 1
Fin Push
```

Peek(pila)//O(1)
Precondición: pila.tope ≠ vacío
Postcondición: se devuelve el elemento en el tope sin eliminarlo.

```
    Si pila.tope = vacío:
        Error "Pila vacía"
    Fin Si
    Devolver pila.tope.carga
Fin Peek
```

ModificarTope(pila, nuevoElemento) //O(1)

Precondición: pila.tope ≠ vacío, nuevoElemento ≠ vacío

Postcondición: se cambia el valor del elemento en el tope.

```
    Si pila.tope = vacío:
        Error "Pila vacía"
    Fin Si
    pila.tope.carga <- nuevoElemento
Fin ModificarTope
```

Pop(pila) //O(1)
Precondición: pila.tope ≠ vacío
Postcondición: se elimina el elemento del tope de la pila.

```
    Si pila.tope = vacío:
        Error "Pila vacía"
    Fin Si
    nodoEliminado <- pila.tope
    pila.tope <- nodoEliminado.siguiente
    nodoEliminado.siguiente <- vacío
    pila.largo <- pila.largo - 1
```

Fin Pop

1) Se necesita construir un analizador sintáctico para cierto lenguaje de programación y para ello es necesario escribir un método que, dada una entrada representada por la lista de caracteres del código fuente, controle si la secuencia de corchetes es correcta o no. Por ejemplo, Es una secuencia bien formada: {}{{}} Es una secuencia mal formada: {{}{{} 1. Con la ayuda del tipo de datos abstracto PILA, escribe en seudocódigo la siguiente funcionalidad: De tipo booleano controlCorchetes(de tipo lista de caracteres listaDeEntrada) Que devuelve VERDADERO si la secuencia de los corchetes en la lista de entrada es correcta, y FALSO en caso contrario.

   Tipo booleano controlCorchetes (parámetro: una lista de caracteres llamada entrada)
   Inicio:
      Declarar una pila llamada pilaTemporal // O(1)

      Si entrada está vacía ENTONCES // O(1)
         Retornar error: "la expresión está vacía" // O(1)

      Asignar a actual el primer nodo de entrada // O(1)

      Si actual.siguiente es nulo ENTONCES // O(1)
         Retornar FALSO // hay solo un carácter // O(1)

      Mientras actual.siguiente no sea nulo HACER // O(n), donde n es la cantidad de nodos
         Si el contenido de actual es="{" ENTONCES // O(1)
            Si insertar un valor en pilaTemporal falla ENTONCES // O(1)
               Retornar FALSO // la pila ya estaba llena // O(1)
         Sino si el contenido de actual es ="}" ENTONCES} // O(1)
            Si sacar un valor de pilaTemporal da como resultado nulo ENTONCES // O(1)
               Retornar FALSO // intento de sacar de una pila vacía // O(1)
         Avanzar al siguiente nodo: actual ← actual.siguiente // O(1)
      Fin mientras

      Si la pilaTemporal está vacía ENTONCES // O(1)
         Retornar VERDADERO // O(1)
      Sino
         Retornar FALSO // O(1)
   Fin

ÁRBOLES GENÉRICOS

```
TNodoArbolGenerico.postOrden; //O(n)
COM
        unHijo <- PrimerHijo
        MIENTRAS unHijo <> nulo hacer //O(n)
                unHijo.postOrden
                unHijo <- unHijo.HermanoDerecho
        FINMIENTRAS
        devolver etiqueta
FIN

preOrden //O(n)
COM
        devolver etiqueta
        unHijo <- PrimerHijo
        MIENTRAS unHijo ≠ nulo hacer
                unHijo.preOrden
                unHijo <- unHijo.HermanoDerecho
        FINMIENTRAS
FIN
```

LISTAR INDENTADO

```
FUNCIÓN listarIndentado(nivel: Entero) -> TNodoArbolGenerico:  //O(n)
   SI dato != NULO ENTONCES:
      ESCRIBIR (repetir("  ", nivel) + dato)
   FIN SI

   PARA CADA hijo EN hijos HACER: //O(n)
      SI hijo != NULO ENTONCES:
         hijo.listarIndentado(nivel + 1)
      FIN SI
   FIN PARA

   DEVOLVER este_nodo
FIN FUNCIÓN
```

TRIE

BUSCAR:
NodoTrie Buscar(String unaPalabra)

```
Comienzo
        nodoActual <- this
        para cada caracter car de unaPalabra hacer
                unHijo <- nodoActual.obtenerHijo(car)
                si unHijo = nulo entonces
                        devolver nulo
                sino
                        nodoActual <- unHijo
                fin si
        fin para cada

        sin nodoActual.esPalabra entonces
                devolver nodoActual
        sino
                devolver nulo
        fin si
Fin
```

CONSTRUIR INDICE
COM:
construirIndice(nodo:TNodo,palabra:String)-->void

```
        SI nodo= nulo ENTONCES
                imprimir " "
        SI nodo.esPalabra ENTONCES
                imprimir "palabra" + palabra + "paginas: " + nodo.paginas
        FIN SI
        PARA CADA elemento i: entero,  QUE NO SUPERE los 26 elementos
                SI nodo.hijos[i] <>nulo ENTONCES
                siguienteLetra←CARACTER(CODIGO('a')
                construirIndice(nodo.hijos[i],palabra + siguienteLetra)
        FIN PARA CADA
FIN
```

INSERTAR:

```
insertarTrie(unaPalabra)
   nodoActual ← this

   Para cada caracter car en unaPalabra hacer
      unHijo ← nodoActual.obtenerHijo(car)

      Si unHijo = nulo entonces
         unHijo ← nuevo NodoTrie
         nodoActual.agregar(unHijo, car)  // depende de la estructura
      Fin Si

      nodoActual ← unHijo
   Fin Para

   nodoActual.esPalabra ← VERDADERO
Fin insertar
```

PREDECIR:

```
CLASE ÁrbolTrie
COMIENZO
Predecir(prefijo: Cadena) : Lista de Cadenas
   resultados ← nueva lista vacía
   Si raiz ≠ nulo Entonces
      raiz.Predecir(prefijo, resultados)
   Fin Si
   Retornar resultados
Fin Predecir
FIN

CLASE NodoTrie
COMIENZO
Predecir(String prefijo, LinkedList palabras)
   nodo ← BuscarNodoTrie(prefijo)
   Si nodo ≠ nulo Entonces
      PredecirAux(prefijo, palabras, nodo)
   Fin Si
Fin Predecir

PredecirAux(String s, LinkedList palabras, nodo)
   Si nodo ≠ nulo Entonces
      Si nodo.esPalabra Entonces
         Agregar s a la lista palabras
      FinSi

      Para c desde 0 hasta CANT_CHR_ABECEDARIO - 1 hacer
```

```
            Si nodo.hijos[c] ≠ nulo Entonces
                letra ← carácter correspondiente a (c + 'a')
                PredecirAux(s + letra, palabras, nodo.hijos[c])
            Fin Si
        Fin Para
    Fin Si
Fin PredecirAux
FIN
```

BUSCAR

```
FUNCIÓN buscar(palabra: String) -> void:
    comparaciones = 0
    resultado: Lista<Integer> = buscarRec(palabra, 0)

    SI resultado != NULO ENTONCES:
        ESCRIBIR "Palabra encontrada: " + palabra + " "
        ESCRIBIR "Páginas: " + resultado
    SINO:
        ESCRIBIR "Palabra no encontrada: " + palabra + " "
    FIN SI

    ESCRIBIR "Comparaciones: " + getComparaciones()
FIN FUNCIÓN

FUNCIÓN buscarRec(palabra: String, indice: Integer) -> Lista<Integer>:
    SI indice == longitud(palabra) ENTONCES:
        SI esPalabra ENTONCES:
            DEVOLVER paginas
        SINO:
            DEVOLVER NULO
        FIN SI
    FIN SI

    caracter: Char = aMinúscula(palabra[indice])
    posicion: Integer = ASCII(caracter) - ASCII('a')
    INCREMENTAR comparaciones

    SI posicion < 0 O posicion > 25 O hijos[posicion] == NULO ENTONCES:
        DEVOLVER NULO
    FIN SI

    DEVOLVER hijos[posicion].buscarRec(palabra, indice + 1)
FIN FUNCIÓN
```

ARBOL BINARIO

```
TipoArbolBB.eliminar(unaEti de tipo etiqueta) //O(n)
COMIENZO
   Si raíz <> nulo entonces
      Raíz ← raíz.eliminar(unaEti)
   Fin si
FIN

TipoNodoABB.eliminar(unaEti de tipo etiqueta): de TipoNodoABB //O(n)
COMIENZO
   Si unaEti < etiqueta entonces // si está, está en el subárbol izquierdo
      Si hijoIzq <> nulo entonces
         hijoIzq ← hijoIzq.eliminar(unaEti) // actualiza el hijo, con el mismo u
otro valor
      Fin si
      retornar (this) // al padre le devuelve el mismo hijo
   Fin si

   Si unaEti > etiqueta entonces // si está, está en el subárbol derecho
      Si hijoDer <> nulo entonces
         hijoDer ← hijoDer.eliminar(unaEti) // actualiza el hijo con el mismo u
otro valor
      Fin si
      retornar (this) // al padre le devuelve el mismo hijo
   Fin si

   retornar quitaElNodo() // está, hay que eliminarlo; al padre le devuelve el
nuevo hijo
FIN

// Cuando encuentra el nodo a eliminar, llama, por claridad, al método que
hace el trabajo

TipoNodoABB.quitaElNodo: de TipoNodoABB //O(n)
COMIENZO
   1) Si hijoIzq = nulo entonces // le falta el hijo izquierdo o es hoja
      retornar hijoDer // puede retornar un nulo

   2) Si hijoDer = nulo entonces // le falta el hijo derecho
      retornar hijoIzq

   3) // es un nodo completo
      elHijo ← hijoIzq // va al subárbol izquierdo
      elPadre ← this
      mientras elHijo.hijoDer <> nulo hacer
         elPadre ← elHijo
         elHijo ← elHijo.hijoDer
      fin mientras
```

```
            // elHijo es el más a la derecha del subárbol izquierdo
            Si elPadre <> this entonces
               elPadre.hijoDer ← elHijo.hijoIzq
               elHijo.hijoIzq ← hijoIzq
            Fin si
            elHijo.hijoDer ← hijoDer
            retornar elHijo // elHijo quedará en lugar de this
        FIN
```

BUSCA

```
// de TArbolBB
        Buscar(unaEtiqueta: Comparable): TElementoAB //O(n)
        Comienzo
           SI esVacio() ENTONCES
              devolver nulo
           SINO
              devolver raiz.buscar(unaEtiqueta)
           FIN SI
        Fin



// de TElementoAB Buscar(unaEtiqueta) : TElementoABinario   //O(n)
        Comienzo
           RESULTADO ← nulo

           SI unaEtiqueta = etiqueta ENTONCES
              RESULTADO ← THIS
           SINO
              SI unaEtiqueta < etiqueta ENTONCES
                 SI hijoIzquierdo <> nulo ENTONCES
                    RESULTADO ← hijoIzquierdo.Buscar(unaEtiqueta)
                 FIN SI
              SINO
                 SI hijoDerecho <> nulo ENTONCES
                    RESULTADO ← hijoDerecho.Buscar(unaEtiqueta)
                 FIN SI
              FIN SI
           FIN SI

           devolver RESULTADO
        Fin
```

VERIFICAR SI ES DE BÚSQUEDA

Para cumplir con lo solicitado tenemos que revisar que si cada clave del arbol cumple que si la clave es menor que la raiz va a la izquierda sino va a la derecha

EsArbolDeBusqueda(nodo, minimo, maximo) o(n)

Pre: nodo != null, minimo <= maximo.

Post: Devolvera true si el árbol es un árbol de búsqueda binaria, de lo contrario false.

```
   si (nodo == null)
      devolver true
   fin si

   si (nodo.clave <= minimo) o (nodo.clave >= maximo)
      devolver false
   fin si

   devolver EsArbolDeBusqueda(nodo.hijoIzq, minimo, nodo.clave)
      y EsArbolDeBusqueda(nodo.hijoDer, nodo.clave, maximo)
fin
```

TAMAÑO(nodo:TNodo)--> entero

```
        SI nodo = nulo ENTONCES
                retornar 0
        FIN SI
        SINO
                tamañoIzq←tamaño(nodo.izquierdo)
                tamañoDer←tamaño(nodo.der)
                retornar 1 + tamañoIzq + tamañoDer
        FIN SINO
FIN
```

ALTURA (nodo:TNodo)-->entero

```
        SI nodo= nulo ENTONCES
                retornar -1
        FIN SI
        SINO
                alturaIzq←altura(nodo.izq)
                alturaDer←altura(nodo.der)
                retornar 1+ MAX(alturaIzq,alturaDer)
        FIN SINO
FIN
```

NIVEL(nodo:TNodo, etiqueta:Comparable)--> entero

```
        SI nodo=nulo ENTONCES
                retornar -1
        FIN SI
        SI nodo.etiqueta=etiqueta ENTONCES
                retornar 0
        FIN SI
        SINO
                nivelIzq←nivel(nodo.izq,etiqueta)
                nivelDer←nivel(nodo.der,etiqueta)
                SI nivelIzq.etiqueta=>0 ENTONCES
```

```
                retornar 1 + nivelIzq.etiqueta
            FIN SI
            SI nivelDer.etiqueta=>0 ENTONCES
                retornar 1 + nivelDer.etiqueta
            FIN SI
            retorna -1

FIN
```

CANTHOJAS(nodo:Tnodo)--> entero

```
    SI nodo= nulo ENTONCES
        retornar 0
    FIN SI
    SI nodo.izq= nulo Y nodo.der = nulo ENTONCES
        retornar 1
    FIN SI
    hojasIzq=cantHojas(nodo.izq)
    hojasDer=cantHojas(nodo.der)
    retornar hojasIzq+hojasDer
FIN
```

NODOSCOMPLETOS(nodo:TNodo)--> entero

```
    SI nodo=nulo ENTONCES
        retornar 0
    FIN SI
    SI nodo.izq<> nulo Y nodo.der<> nulo ENTONCES
        contador = 1
    FIN SI
    nodosIzq=nodosCompletos(nodo.izq)
    nodosDer=nodosCompletos(nodo.der)
    retornar nodosIzq+nodosDer + contador
FIN
```

NODOSINTERNOS(nodo:TNodo)-->entero

```
    SI nodo=nulo ENTONCES
        retornar 0
    FIN SI
    SI nodo.izq=nulo Y nodo.der=nulo ENTONCES
        retornar 0
    FIN SI
    nodosIzq=nodosInternos(nodo.izq)
    nodosDer=nodosInternos(nodo.der)
    retormar 1 +  nodosIzq+nodosDer
FIN
```

```
De TElementoAB
Insertar(UnElementoArbolBinario)
COM
 SI Etiqueta = unElementoArbolBinario.Etiqueta ENTONCES
          SALIR // ya está en el árbol
 FINSI
 SI unElementoArbolBinario.Etiqueta < Etiqueta ENTONCES
    SI HijoIzquierdo = nulo ENTONCES
          HijoIzquierdo ← unElementoArbolBinario
    SINO HijoIzquierdo.Insertar(unElementoArbolBinario)
    FINSI
 SINO
    SI HijoDerecho = nulo ENTONCES
          HijoDerecho ←unElementoArbolBinario
    SINO HijoDerecho.Insertar(unElementoArbolBinario)
    FINSI
  FINSI
FIN
```

Algoritmos y Estructuras de Datos

ARBOL BINARIO DE BUSQUEDA:

BUSCAR

```
De TElementoAB
Buscar(UnaEtiquetaqueta) : TElementoABinario

COM
 RESULTADO = nulo
 SI UnaEtiquetaqueta = Etiqueta ENTONCES
    RESULTADO = THIS
  SINO
    SI UnaEtiquetaqueta < Etiqueta ENTONCES
        SI HijoIzquierdo <> nulo ENTONCES
              RESULTADO = HijoIzquierdo.Buscar(UnaEtiquetaqueta)
        FINSI
    SINO
        SI HijoDerecho <> nulo ENTONCES
              RESULTADO = HijoDerecho.Buscar(UnaEtiquetaqueta)
    FINSI
    FINSI
 FINSI
 devolver RESULTADO
FIN
```

```
public TElementoAB buscar(Comparable UnaEtiquetaqueta) {
   if (UnaEtiquetaqueta.compareTo(etiqueta) == 0) {
       return this;
   } else {
       if (UnaEtiquetaqueta.compareTo(etiqueta) < 0) {
           if (hijoIzq != null) {
               return hijoIzq.buscar(UnaEtiquetaqueta);
           } else {
               return null;
           }
       } else {
           if (UnaEtiquetaqueta.compareTo(etiqueta) > 0) {
               if (hijoDer != null) {
                   return hijoDer.buscar(UnaEtiquetaqueta);
               } else {
                   return null;
               }
           } else {
               return null;
           }
   }
}
}
```

INSERTAR

INORDEN

```
public String inOrden() {
    String tempStr = "";
    if (hijoIzq != null) {
        tempStr = hijoIzq.inOrden();
    }
    tempStr = tempStr + imprimir();
    if (hijoDer != null) {
        tempStr = tempStr + hijoDer.inOrden();
    }
    return tempStr;
}
```

PREORDEN

```
tempStr=""
tempStr=tempStr + imprimir
SI (hijoIzq!=null) ENTONCES
        tempStr← tempStr+hijoIzq.preOrden
FIN SI
SI (hijoDer!=null) ENTONCES
        tempStr←tempStr + hijoDer.preOrden
FIN SI
retorna tempStr
FIN
```

POSTORDEN

```
tempStr=""
SI (hijoDer!=null) ENTONCES
        tempStr←tempStr + hijoDer.preOrden
FIN SI
SI (hijoIzq!=null) ENTONCES
        tempStr← tempStr+hijoIzq.preOrden
FIN SI
tempStr=tempStr + imprimir
retorna tempStr
FIN
```

INDIZAR POR APELLIDO

```
FUNCIÓN indizarPorApellido(carrera: String) -> TElemento:
   arbol: TElemento = NULO
   actual: TElemento = este_elemento

   MIENTRAS actual != NULO HACER:
      SI actual.alumno.getCarrera() == carrera Y actual.alumno.esMatriculado()
ENTONCES:
         SI arbol == NULO ENTONCES:
            arbol = NUEVO TElemento(actual.alumno)
         SINO:
            arbol.insertarABB(actual.alumno)
         FIN SI
```

```
        FIN SI
        actual = actual.siguiente
    FIN MIENTRAS

    DEVOLVER arbol
FIN FUNCIÓN
```

ELIMINAR:

```
ElementoAB.Eliminar (UnaEtiquetaqueta) : de Tipo TElementoAB
COM
(1) Si UnaEtiqueta < etiqueta entonces                    // si esta, está en el subárbol izquierdo
    Si hijoIzq <> nulo entonces
          hijoIzq ← hijoIzq.eliminar(UnaEtiqueta)         //actualiza el hijo, con el mismo u otro valor
       Finsi
    retornar (this)                                       //al padre le devuelve el mismo hijo
    Finsi

  (2) Si UnaEtiqueta > etiqueta entonces                  // si esta, está en el subárbol derecho
       Si hijoDer <> nulo entonces
         hijoDer ← hijoDer.eliminar(UnaEtiqueta)          //actualiza el hijo, con el mismo u otro valor
       Finsi
       retornar (this)                                    // al padre le devuelve el mismo hijo
     Finsi

    (3) retornar quitaElNodo                              // esta, hay que eliminarlo
                                                          // al padre le devuelve el nuevo hijo
Fin

// Cuando encuentra el nodo a eliminar llama, por claridad, al método que hace el trabajo
```

SUMA DE NIVELES DE TODOS LOS NODOS

```
SumaDeNiveles(nivel:entero)-->entero
suma←nivel
SI this.izq<>nulo ENTONCES
        suma←suma + this.izq.sumaDeNiveles(nivel+1)
FIN SI
SI this.der<> nulo ENTONCES
        suma←suma+this.der.sumaDeNiveles(nivel+1)
FIN SI
retornar suma
```

AVL

```
FUNCION obtenerAltura(nodo)
        SI nodo = NULL ENTONCES
           RETORNAR 0
        SINO
           RETORNAR nodo.altura
        FIN SI
      FIN FUNCION

FUNCION obtenerBalance()
        RETORNAR obtenerAltura(izquierda) - obtenerAltura(derecha)
      FIN FUNCION

FUNCION rotacionDerecha()
        k1 ← ESTE
        k2 ← ESTE.izquierda

        k1.izquierda ← k2.derecha
        k2.derecha ← k1

        k1.actualizarAltura()
        k2.actualizarAltura()

        RETORNAR k2
      FIN FUNCION

FUNCION rotacionIzquierda()
        k1 ← ESTE
        k2 ← ESTE.derecha

        k1.derecha ← k2.izquierda
        k2.izquierda ← k1

        k1.actualizarAltura()
        k2.actualizarAltura()

        RETORNAR k2
      FIN FUNCION

PROCEDIMIENTO actualizarAltura()
        ESTE.altura ← 1 + MAX(obtenerAltura(izquierda), obtenerAltura(derecha))
      FIN PROCEDIMIENTO

FUNCION insertar(clave)
        SI clave < ESTE.clave ENTONCES
           SI izquierda = NULL ENTONCES
              izquierda ← NUEVO TNodoAVL(clave)
```

```
            SINO
               izquierda ← izquierda.insertar(clave)
            FIN SI
         SINO SI clave > ESTE.clave ENTONCES
            SI derecha = NULL ENTONCES
               derecha ← NUEVO TNodoAVL(clave)
            SINO
               derecha ← derecha.insertar(clave)
            FIN SI
         SINO
            // Duplicados no permitidos
            RETORNAR ESTE
         FIN SI

actualizarAltura()
         balance ← obtenerBalance()

         SI balance > 1 Y clave < izquierda.clave ENTONCES
            MOSTRAR "Desbalance II en nodo " + ESTE.clave
            RETORNAR rotacionDerecha()
         FIN SI

         SI balance < -1 Y clave > derecha.clave ENTONCES
            MOSTRAR "Desbalance DD en nodo " + ESTE.clave
            RETORNAR rotacionIzquierda()
         FIN SI

         SI balance > 1 Y clave > izquierda.clave ENTONCES
            MOSTRAR "Desbalance ID en nodo " + ESTE.clave
            izquierda ← izquierda.rotacionIzquierda()
            RETORNAR rotacionDerecha()
         FIN SI

         SI balance < -1 Y clave < derecha.clave ENTONCES
            MOSTRAR "Desbalance DI en nodo " + ESTE.clave
            derecha ← derecha.rotacionDerecha()
            RETORNAR rotacionIzquierda()
         FIN SI

         RETORNAR ESTE
      FIN FUNCION
```