



Diseño con Listas Enlazadas

Ejercicio 1. Implementamos el TAD Secuencia sobre una lista simplemente enlazada usando

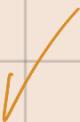
```
NodoLista<T> es struct<
    val: T,
    siguiente: NodoLista<T>,
>

Modulo ListaEnlazada<T> implementa Secuencia<T>{
    var primero: NodoLista<T> // "puntero" al primer elemento
    var ultimo: NodoLista<T> // "puntero" al primer elemento
    var longitud: Z // cantidad total de elementos
    ...
}
```

Escriba los algoritmos para los siguientes procs y calcule su complejidad

- a) nuevaListaVacia(): ListaEnlazada<T>
- b) agregarAdelante(inout l: ListaEnlazada<T>, in t: T)
- c) agregarAtras(inout l: ListaEnlazada<T>, in t: T)
- d) eliminar(inout l: ListaEnlazada<T>, in i: int)
- e) pertenece(in l: ListaEnlazada<T>, in t: T) : bool
- f) obtener(in l: ListaEnlazada<T>, in i: int) : T
- g) concatenar(inout l1: ListaEnlazada<T>, in l2: ListaEnlazada<T>)
- h) sublista(in l: ListaEnlazada<T>, in inicio : int, in fin: int): ListaEnlazada<T>

a) PROC NUEVALISTAVACIA(): LISTAENLAZADA<T>{
 var PRIMERO := NULL;
 var ULTIMO := NULL;
 var LONGITUD := 0;
}



Complejidad: $\Theta(1)$ para MEJOR y PEOR Caso. Solo OP
elementales.

b) PROC AGREGARADELANTE (inout l: LISTAENLAZADA<T>, in t: T){

VAR Nodo := NODOLISTA<T>:= NEW NODOLISTA<T>(x);

```

IF (X.LONGITUD == 0) THEN:
    l.PRIMERO := NODO;
    l.ULTIMO := NODO;
ELSE
    VAR PRIMANT:NODOLISTACT;:= l.PRIMERO;
    NODO.SIGUIENTE:= PRIMANT;
    l.PRIMERO := NODO;
    l.ULTIMO := PRIMANT;
ENDIF
l.LONGITUD ++
}

```



Complejidad: $O(1)$ para los tres operaciones elementales.

En el MEJOR CASO $l.\text{LONGITUD} = 0$, en el PEOR
en $l.\text{LONGITUD} \neq 0$ xq necesitas recorrer el NODO
PRIMERO ANTERIOR.

C) PROC AGREGARATRAS (INOUT LISTAENLAZADA<T>, IN X:T) {

```

IF (l.LONGITUD == 0) THEN:
    l.AGREGARADELANTE(l,X)
ELSE:
    VAR NODO:NODOLISTACT;:= NEW NODOLISTACT(X);
    VAR ULTANT:NODOLISTACT;:= l.ULTIMO;
    ULTANT.SIGUIENTE:=NODO;
    l.ULTIMO := NODO;
ENDIF

```

l. LONGITUD++

}

Complejidad mejor caso: $\Theta(1)$ las listas están vacías.



Complejidad peor caso: $\Theta(n^2)$ al tener el VLT todos los nodos necesitan recorrer la SLL siendo $O(n)$.



l. ULTIMO es puntero con SLL Original, lo muta directamente

→ REQ: l. LONGITUD > 0

d) PROC ELIMINAR(inout l: ListadenLazadas, in i: int){

VAR ACTUAL:NododenLazadas:= l.PRIMER;

VAR PREV:NododenLazadas:= l.PRIMER;

VAR j:int:=0;

IF (i==0) THEN:

si el primero es null no ejecuta

l.PRIMER:= l.PRIMER.SIGUIENTE

ELSE

WHILE(j < i) DO:

PREV:= ACTUAL;

ACTUAL:= ACTUAL.SIGUIENTE;

j:= j+1

END WHILE



PREV.SIGUIENTE:= ACTUAL.SIGUIENTE

ENDIF

l.LONGITUD--;

}

NULL, VACUO = EXP

$\ell.Prim = \ell.Prim.Sig$

NULL

MÉJOR CASO $\Rightarrow i=0$. PUEDO USAR $\ell.Primero$.

El coste es $\Theta(1)$.

POR CASO $\Rightarrow i=\ell.longitud - 1$ debemos ver mejor
Cosa también, pero al no tener el NODO previo no
tengo otra q ir recorriendo desde el PRIMERO.

El coste es $\Theta(n)$

PARA NO ENCONTRAR SIGUIENTE := NULL

e) PERTENECE

PROC PERTENECE(*inout* ℓ : LISTAENLAZADA<T>, *in* e:T): Bool {

VAR Encontrado: Bool := false

VAR NODO:NODOLISTACT >:= $\ell.Primero$

IF (NODO.VAL == e) THEN:

RETURN TRUE;

ENDIF

WHILE (NODO != NULL \wedge !Encontrado) DO:

IF (NODO.VAL == e) THEN:

Encontrado := true;

ELSE:

NODO := NODO.SIGUIENTE;

ENDIF

ENDWHILE

RETURN ENCONTRADO

}

$\geq \Theta(n)$

PEOR CASO \Rightarrow recorre la lista entera

MEJOR CASO \Rightarrow en el primer elem

$\hookrightarrow \Theta(1)$ /

f | PROC OBTENER (INOUT l: LISTAENLAZADA<T>, IN i:T) : T {

VAR j: INT := 0;

VAR NODO: NODO<LISTA> := l.PRIMERO;

WHILE (j < i) Do:

NODO := NODO.SIGUIENTE;

j := j + 1;

ENDWHILE

RETURN NODO.VAL

}

MEJOR CASO $O(1)$ ↗ PEOR $O(n)$ ↗

g |



|

TENGO

i



TENGO

Ent

Dime

USUL DE X EN EL PRIMER L2.
Luego, ULT de L EN ULT de L2.

PROC CONCATENAR (INOUT L1: LISTAENLAZADA<T>, IN L2: LISTAENLAZADA<T>){

L1.ULTIMO.SIGUIENTE := L2.PRIMERO;

L1.ULTIMO := L2.ULTIMO;

L1.LONGITUD := L1.LONGITUD + L2.LONGITUD

}

Océ te garantizo que L2 es
UNA COPIA.

POR CASO = MEJOR CASO $\Theta(1)$.

OJO ALIASING.

PROC CONCATENAR (INOUT L1: LISTAENLAZADA<T>, IN L2: LISTAENLAZADA<T>){ \rightarrow SIN ALIASING }

VAR L: LISTAENLAZADA<T> := NEW LISTAENLAZADA();

VAR i: INT := 0;

VAR NODO TO COPY: NODOLISTA<T> := L2.PRIMERO;

WHILE(i < L2.LONGITUD) DO:

L.AGREGARATRAS(L, NODO TO COPY.VAL);

NODO TO COPY := NODO TO COPY.SIGUIENTE

i++;

END WHILE

L1.ULTIMO.SIGUIENTE := L.PRIMERO;

L1.ULTIMO := L.ULTIMO;

L1.LONGITUD += L.LONGITUD

}

L SUBLISTA : POR CASO se nos hace hacer el ULT (no incluido)

Ejercicio 2. Para el módulo ListaEnlazada definido en el ejercicio anterior

- Escriba el invariante de representación para este módulo en castellano
- Dedo el siguiente invariante de representación, indique si es correcto. En caso de no serlo, corrijalo:

```
pred InvRep(l: ListaEnlazada<T>)
    {accesible(l.primer, l.ultimo) ∧ largoOK(l.primer, l.longitud)}

pred accesible(n0: NodoLista<T>, n1: NodoLista<T>)
    {n1 = n0 ∨ (n0.siguiente ≠ null ∧L accesible(n0.siguiente, n1))}

pred largoOK(n: NodoLista<T>, largo: ℤ)
    {(n = null ∧ largo = 0) ∨ (largoOK(n.sigüiente, largo - 1))}
```

a) INVREP: Reglas sobre VARIÁBLES DE ESTADO
Al cumplir se informa a un PROC de salir.

LONGITUD

↳ CANT NODOS PRIM A ÚLTIMO

S: PRIMER TIENE VALOR, ENT ULT TAMBIÉN.

SI ULT TIENE VALOR, ENT PRIM TAMBÍEN

PUEDE LLEGAR DE PRIMER A ÚLTIMO.

NO HAY CICLOS

Con revisión xg
No tiene.

- Dedo el siguiente invariante de representación, indique si es correcto. En caso de no serlo, corrijalo:

```
pred InvRep(l: ListaEnlazada<T>)
    {accesible(l.primer, l.ultimo) ∧ largoOK(l.primer, l.longitud)}
```

```

pred accesible(n0: NodoLista<T>, n1: NodoLista<T>)
  {n1 = n0 V (n0.siguiente ≠ null AL accesible(n0.siguiente, n1))} ^ AL AÚN UST S:6 EN NULL

pred largoOK(n: NodoLista<T>, largo:  $\mathbb{Z}$ )
  {(n = null A largo = 0) V (largoOK(n.siguiente, largo - 1))}

```

- LargoOK y Accesible no tienen ciclos (recursivo)
- LARGOK: Puede el largo hacer BIEN y el punto recursivo también.
- En Accesible se hace el recorrido primero hasta último pero folio decir que en el último se da que nodo.sigüiente=null

? ¿ TENDRÉ QUE HACER PRED ABS Y INVREP?

Ejercicio 3. Defina el módulo de las siguientes alternativas a una lista simplemente enlazada. Para eso será necesario:

- Elegir las variables de la estructura
- Escribir el invarianta de representación en castellano
- En caso de ser necesario, reescribir los algoritmos de las operaciones implementadas en el ejercicio anterior
- Calcular las complejidades de las operaciones

1

a) Lista doblemente enlazada que usa la siguiente definición del módulo NodoLista

```

NodoLista<T> es struct<
  val: T,
  siguiente: NodoLista<T>,
  anterior: NodoLista<T>,
>
```

b) Lista circular (simple o doblemente enlazada) donde el "último" nodo está conectado con el primero

c) Lista de arreglos: una lista doblemente enlazada cuyos valores son arreglos de tamaño fijo, es decir que el nodo es

```

NodoLista<T> es struct<
  data: Array<T>,
  siguiente: NodoLista<T>,
  anterior: NodoLista<T>,
>
```

- a) La diferencia la que ocurre cuando: • Invierte Adelante, si habia uno anterior ahora sera el 2do y el PREV sera el nuevo.
 • Invierte al agregar al final q el PREV es el ULT anterior.
 • Invierte en caso de eliminar.
 Lo demás, igual.

INVREP:

- No tiene ciclos.
- La longitud es la misma que gente PRIMERO.SIGUIENTE hasta que sea null
 Que haciendo ULTIMO.ANTERIOR hasta que sea null.
- Del primero lleva al ultimo borra el siguiente que gente del ultimo
 Al primero llevando el PREVIO.
- PREVIO A ULTIMO. BORRAR UNO. Siendo el ULTIMO.

PRIMER). ANTERIOR = NULL & ULTIMO SIGUIENTE = MB.

LISTA ENLAZADA

''

MODULO LISTA DOBLEMENTE ENLAZADA $\langle T \rangle$ IMPLEMENTA SECUENCIAS $\{$

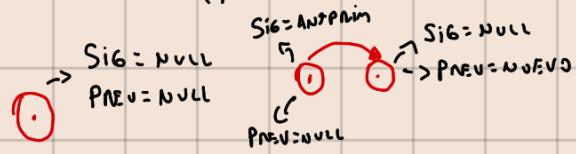
VAR PRIMERO: NODO LISTA $\langle T \rangle$;

VAR ULTIMO: NODO LISTA $\langle T \rangle$;

VAR LONGITUD: INT

Costo $\Theta(1)$

PROC NUEVALISTA VACIA(): LISTA ENLAZADA $\langle T \rangle$ EN SLL



Costo $\Theta(1)$, los manejos de punteros que ya tienen son Ocio $\Theta(1)$.

PROC AGREGAR ADELANTE (INOUT l: LISTA ENLAZADA $\langle T \rangle$, IN x: T) {

VAR NUEVONODO: NODO LISTA $\langle T \rangle$:= NEW NODO LISTA $\langle T \rangle(x)$;

IF (l.PRIEROS == 0) THEN

l.PRIEROS := NUEVONODO;

l.ULTIMO := NUEVONODO;

ELSE

VAR PREV: NODO LISTA $\langle T \rangle$:= l.PRIEROS;

NUEVONODO.SIGUIENTE := PREV;

PREV.ANTERIOR := NUEVONODO;

l.PRIEROS := NUEVONODO;

ENDIF

l.LONGITUD++

}

Costo $\Theta(1)$ al usar l.ultimo.



PROC AGREGAR ATRAS (INOUT l: LISTA ENLAZADA $\langle T \rangle$, IN x: T) {

VAR NODO: NODO LISTA $\langle T \rangle$:= NEW NODO LISTA $\langle T \rangle(x)$;

IF (l.LONGITUD == 0) THEN

l.AGREGAR ADELANTE(l,x);

ELSE

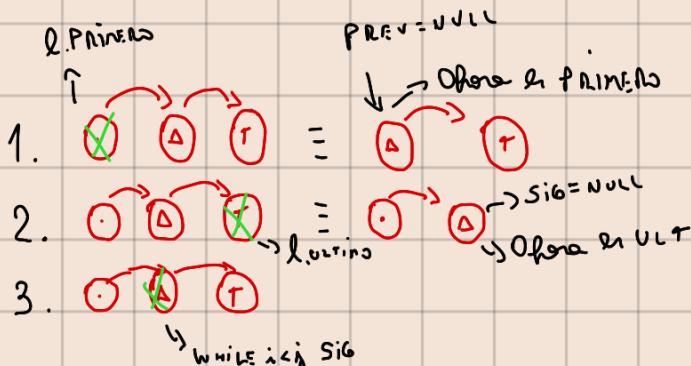
VAR PREVULY: NODO LISTA $\langle T \rangle$:= l.ULTIMO;



```

NODO.ANTERIOR := PREVULT;
PREVULT.SIGUIENTE := NODO;
l.ULTIMO := NODO;
ENDIF
l.LONGITUD++;
}

```



$i=0$

OJO: Aquí mismo l.ULTIMO hace borrar el último
 $n.c = \Theta(1)$ $P.C = \Theta(M)$. xq la complejidad se complica para calcularla

PROC ELIMINAR (inout l: LISTADENLAZADA<T>, in i: int) {

```

VAR j:int := 0
VAR NODO := l.PRIMERO
IF (i == 0) THEN
    VAR NUENSPRIM := l.PRIMERO.SIGUIENTE;
    NUENSPRIM.ANTERIOR := NULL;
    l.PRIMERO := NUENSPRIM;
ELSE
    WHILE (j < i) DO:
        NODO := NODO.SIGUIENTE;
        j := j + 1;
        i := i + 1;
    EndWHILE

```

VAR NODOPREV := NODO.ANTERIOR;

```

    IF (y == l.LONGITUD - 1):
        NODOPREV.SIGUIENTE = NULL;
        l.ULTIMO = NODOPREV;
    ELSE:
        VAR Nodosig := NODO.SIGUIENTE;
        NODOPREV.SIGUIENTE = Nodosig;
        Nodosig.ANTERIOR = NODOPREV;
    ENDIF
    l.LONGITUD --
}

```

```

PROC PERTENECE = SLL
PROC OBTENER = SLL
PROC CONCATENAR = SLL
PROC SUBLISTA = SLL
}

```

El c) es igual a la xq no tiene modo mas q el tipo de la lista.

El b) si cambia algo en los:

- Al agregar adelante, si tiene lista vacía, ULT.SIGUIENTE = PRIMEROS
- Al agregar otro, si tiene lista vacía para ⁵ _{NUEVO}, l.ULTIMO.SIGUIENTE = l.PRIMEROS
- Al eliminar nos irá en el vrt, q el PREV q ahora es el vrt apunta a _{NUEVO ULT.} PRIMEROS.

Lo demás quedó igual.

Ejercicio 4. Implementar los siguientes TADs (cuyas especificaciones están en el apunte) sobre alguna forma de ListaEnlazada (simple, doble, etc.), explicando por qué eligió esa variante. Calcule las complejidades de los procs

- a) Secuencia<T>
- b) Cola<T>
- c) Pila<T>
- d) Conjunto<T>
- e) Diccionario<K,V>
- f) Multiconjunto<T>

Hago a), b), c), d) y e)

a) Hacé una DOUBLINKEDLIST xq para modificar posición tiene que tener ELEMAPISAR.PREV ^ ELEMAPISAR.SIG xq operación al nuevo modo \Rightarrow Coste $O(n)$ en peor caso ✓

b) Hacé una SINGLELINKEDLIST con el primer nodo xq al recorrer solo el primero, y el ult podés guardar refs para que el operador \rightarrow tenga costo $O(1)$. Como me necesitas redondear la lista, o el Arit no me importa el PREV me hice una doblemente enlazada. ✓

c) Mismo b.

d) Hacé DLL xq la unir me hace mismo que a) en CONCATENAR.

e) Hacé SLL xq no tengo ninguna OP q implique orden.

El a) es el ejercicio 3a

Implementar b) y c.

```
TAD Cola<T> {
    ob: s: seq<T>
    proc colaNueva(): Cola<T>
        asegura {res.s = ()}
    proc vacia(in c: Cola<T>): bool
        asegura {res = true  $\leftrightarrow$  c.s = ()}
    proc encolar(inout c: Cola<T>, in e: T)
        requiere {c.s = C0.s}
        asegura {c.s = concat(C0.s, {e})}
    proc desencolar(inout c: Cola<T>): T
        requiere {c.s = C0}
        requiere {c.s  $\neq$  ()}
        asegura {c.s = subseq(C0.s, 1, |C0.s|)}
        asegura {res = C0[0]}
    proc proximo(in c: Cola<T>): T
        requiere {c.s = C0}
        requiere {c.s  $\neq$  ()}
        asegura {res = C0.s[0]}}
}
```

b)

AGREGO ATRS, SACO PRIMEROS

MÓDULO COLASLL<T> IMPLEMENTA COLA<T>{

VAR S: LISTAENLAZADA<T>

COLAVACIA \rightarrow Coste $O(1)$

PROC NUEVACOLASLL(): COLASLL<T>{

 CL.S := NEW LISTAENLAZADA<T>();

}

→ Costo Θ(1)

PROC VACIA(IN CL: COLASLL<T>): bool {

 RETURN CL.S.LONGITUD == 0

}

→ Costo Θ(1)

PROC ENCOLAR(INOUT CL: COLASLL<T>, IN e: T) {

 CL.S.AGREGARATRAS(CL.S, e);

}

↳ Iniciamos SLL

→ Costo Θ(n)

PROC DESENCOLAR(INOUT CL: COLASLL<T>): T {

 VAR DESENCOLADO := NEW CL.S.NODO(CL.S.PRIMERO);

 CL.S.ELIMINAR(CL.S, 0);

↳ Lo hago así xq si quita & el

no es primitivo, y lo bonito pasa por

REF y devolver algo NULL.

 RETURN DESENCOLADO.VALOR

}

→ Costo Θ(1)

PROC PROXIMO(IN CL: COLASLL<T>): T {

 VAR PROXIMO := NEW CL.S.NODO(CL.S.PRIMERO);

 RETURN PROXIMO.VALOR

}

}

```
TAD Pila<T> {
    obs s: seq<T>
    proc pilavacia(): Pila<T>
        asegura {res.s = ()}
```

2

```
proc vacia(in p: Pila<T>): bool
    asegura {res = true ↔ p.s = ()}
proc apilar(inout p: Pila<T>, in e: T)
    requiere {p = P0}
    asegura {p.s = concat(P0.s, {e})}
proc desapilar(inout p: Pila<T>): T
    requiere {p = P0}
    asegura {p.s ≠ ()}
    asegura {p.s = subseq(P0.s, 0, |P0.s| - 1)}
    asegura {res = P0.s[|P0.s| - 1]}
proc tope(in p: Pila<T>): T
    requiere {p = P0}
    requiere {p.s ≠ ()}
    asegura {res = P0.s[|P0.s| - 1]}
```

C)

AGG ATMS, SACO ULT.

MÓDULO PILA.SLL<T> IMPLEMENTA PILA<T> {

 VAR S: LISTAENLAZADA<T>

 COLAVACIA

→ Costo Θ(1)

PROC NUEVAPILA SLL(): COLASLL<T> {

 RET. S := NEW LISTAENLAZADA<T>();

}

 → Costo $\Theta(1)$

PROC VACIA(IN PL:PILASLL<T>):bool {

 RETURN PL.S.LONGITUD == 0

}

 → Costo $\Theta(1)$

PROC APILAR (INOUT PL:PILASLL<T>, IN e:T) {

 PL.S.AGREGARATRAS (PL.S,e);

}

 ↳ Inserción SLL

 → Costo $\Theta(n)$

PROC DESAPILAR (INOUT PL:PILA SLL<T>): T {

 VAR DESAPILADO := NEW PL.S.NODO(PL.S.ULTIMO);

 PL.S.ELIMINAR(PL.S,PL.S.LONGITUD - 1);
 → Lo hace así xq si queremos q el
 no sea primitivo, q lo bonito pase por
 REF q devolvería algo NULL.

 RETURN DESAPILADO.VALOR

}

 → Costo $\Theta(1)$

PROC TOPE (IN PL:PILA SLL<T>): T {

 VAR TOPE := NEW PL.S.NODO(PL.S.ULTIMO);

 RETURN TOPE.VALOR

}

 → Lo hace así xq si queremos q el
 no sea primitivo, q lo modif pase por
 REF q cambia la REF original.

}



Ejercicio 6. Implementar el TAD PlayaDeManiobras usando listas enlazadas.

Vagón es string
Tren es seq<Vagón>

TAD PlayaDeManiobras {
 obs trenes: seq<Tren>}

 CAPACIDAD PLAYA ES FIJA
 ABRIRPLAYA = NUEVAPLAYA

proc abrirPlaya(in capacidad: ZZ) : PlayaDeManiobras
 requiere { capacidad > 1 }

```

asegura { |ret.trenes| = capacidad  $\wedge$  ( $\forall i : \mathbb{Z}$ ) $(0 \leq i < \text{capacidad}) \rightarrow_L (\text{ret.trenes}[i] = \square)$  }
proc recibirTren(inout pdm: PlayaDeManiobras, in t: Tren) : Z
    requiere { ( $\exists v$ ) $(0 \leq v < |\text{pdm.trenes}|) \wedge_L (\text{pdm.trenes}[v] = \square) \wedge \text{pdm} = \text{pdm}_0$  }
    asegura { ( $\exists v$ ) $(0 \leq v < |\text{pdm}_0.\text{trenes}|) \wedge_L (\text{pdm}_0.\text{trenes}[v] = \square \wedge \text{d.m.trenes} = \text{setAt}(\text{pdm}_0.\text{trenes}, v, t)) \wedge \text{ret} = v$  }

```

AGREGA UN INDICE V (DONDE NO HAY NADA) AL TREN

```

proc despacharTren(inout pdm: PlayaDeManiobras, in v: Z)
    requiere { ( $0 \leq v < |\text{pdm.trenes}|) \wedge_L (\text{pdm}[v] \neq \square) \wedge \text{pdm} = \text{pdm}_0$  }
    asegura {  $\text{pdm.trenes} = \text{setAt}(\text{pdm}_0.\text{trenes}, v, \square)$  }

    ↳ PONE EN NULL EL INDICE DE VIAS DEL TREN?

```

```

proc unirTrenes(inout pdm: PlayaDeManiobras, in via1: Z, in via2: Z)
    requiere { ( $0 \leq \text{via1} < |\text{pdm.trenes}|) \wedge (0 \leq \text{via2} < |\text{pdm.trenes}|)$  }
    requiere {  $\text{pdm.trenes}[\text{via1}] \neq \square \wedge \text{pdm.trenes}[\text{via2}] \neq \square$  }

    ↳ PUEDE HACERLO?

```

? NO ES LO MISMO? DESPUES DE QUE?

```

requiere {  $\text{pdm} = \text{pdm}_0$  }
asegura {  $|\text{pdm}| = |\text{pdm}_0|$  }  $\square \square$ 
asegura {  $\text{pdm.trenes}[\text{via1}] = \text{concat}(\text{pdm}_0.\text{trenes}[\text{via1}], \text{pdm}_0.\text{trenes}[\text{via2}])$  }
asegura {  $\text{pdm.trenes}[\text{via2}] = \square \rightarrow \text{BONAS LAS VAGONES DEL TREN} \rightarrow \text{PUNTOS NULL?}$  }
asegura {  $(\forall v : \mathbb{Z})(0 \leq v < |\text{pdm.trenes}|) \wedge v \neq \text{via1} \wedge v \neq \text{via2} \rightarrow_L$ 
             $(\text{pdm.trenes}[v] = \text{pdm}_0.\text{trenes}[v]) \Rightarrow \text{LOS DEMAS Q. SON IGUALES}$  }

proc moverVagon(inout pdm: PlayaDeManiobras, in vagon: Vagon, in viaDestino: Z)
    requiere {  $(0 \leq \text{viaDestino} < |\text{pdm.trenes}|)$  }
    requiere {  $(\exists v : \mathbb{Z})(0 \leq v < |\text{pdm.trenes}|) \wedge_L (\text{vagon} \in \text{pdm.trenes}[v])$  }
    requiere {  $\text{pdm} = \text{pdm}_0$  }
    asegura {  $|\text{pdm}| = |\text{pdm}_0|$  }
    asegura {  $(\exists v : \mathbb{Z})(0 \leq v < |\text{pdm}_0.\text{tren}|) \wedge_L (\text{vagon} \in \text{pdm}_0.\text{trenes}[v] \wedge \text{vagon} \notin \text{pdm.trenes}[v])$  }  $\Rightarrow \text{LE PASA UN VAGON DEL TREN}[v]$ 
    asegura {  $\text{vagon} \in \text{pdm.trenes}[\text{viaDestino}]$  }
    asegura {  $(\forall v : \mathbb{Z})(0 \leq v < |\text{pdm.trenes}|) \wedge v \neq \text{viaDestino} \wedge \text{vagon} \notin \text{pdm.trenes}[v] \rightarrow_L$ 
             $(\text{pdm.trenes}[v] = \text{pdm}_0.\text{trenes}[v])$  }

    ↳ EXISTEN O NO VAGONES EN TRENES[V]

```

En este momento me dice el IND.

↳ FINAL
↳ DEVUELVE IND
TREN

→ PODRIA USAR CAPACIDAD - 1.

¿ Que pasa si CAPACIDAD es full?

LUYQUIERA

AL IN[N][VIADESTINO]

Los demas quedan igual.

Nota: dada que hay operaciones que no tienen en la LSE \rightarrow LDE, lo "bien"

No en el sentido herencia, sino que la copia y le agrega lo nuevo.

Rentabilo 3.a)

MÓDULO LISTADOBLEMENTEENLAZADA \leftarrow IMPLEMENTA SECUENCIA \leftarrow {

VAR PRIMERO: NODOLISTA \leftarrow

VAR ULTIMO: NODOLISTA \leftarrow

VAR LONGITUD: INT

PROC ELIMINARTODOS(inout l: LISTADOBLEMENTEENLAZADA \leftarrow) {

$\begin{cases} l.\text{PRIMERO} := \text{NULL}; \\ l.\text{ULTIMO} := \text{NULL}; \\ l.\text{LONGITUD} := 0 \end{cases}$

↓ $l = \text{NEW LISTADOBLEMENTEENLAZADA}()$?
que no sé ni quiere pedir
la instancia

}

one base nula sera OK

O(m)

PROC AGREGARENLUGARVACIO(inout l: LISTADOBLEMENTEENLAZADA \leftarrow , in t: LDE \leftarrow): INT {

VAR NODO: NODOLISTA \leftarrow := l.PRIMERO; // TREN 1.

En q. necesario? q. Dahan

VAR COPIALDE: LISTADOBLEMENTEENLAZADA \leftarrow := t.COPiar(t); Funcion q. requiere. La usa THIS.

VAR AGREGADO: BOOL := false; se que al llenar, se garantiza q. el indice libre. No me preocupa

→ solo inicial

VAR INDELEM: INT := 0;

```
WHILE(Nodo != NULL ^ !AGREGADO) DO:  
    IF(Nodo.LONGITUD() == 0) THEN:  
        Nodo.PRIMERO := COPIALDE.PRIMEROS;  
        Nodo.ULTIMO := COPIALDE.ULTIMO;  
        Nodo.LONGITUD := COPIALDE.LONGITUD;  
        AGREGADO := TRUE;  
    ELSE:  
        Nodo.PRIMERO := Nodo.PRIMERO.SIGUIENTE;  
        INDELEM += 1;  
    ENDIF;  
ENDWHILE;  
RETURN INDELEM;  
}
```

```
PROC PRIMERO (in & l: LISTADOBLEMENTEENLAZADA< T >): NODO< T >{  
    VAR COPIA: NODO< T > := NEW NODO< T >(l.PRIMERO);  
    RETURN COPIA;  
}
```

```
PROC ULTIMO (in & l: LISTADOBLEMENTEENLAZADA< T >): NODO< T >{  
    VAR COPIA: NODO< T > := NEW NODO< T >(l.ULTIMO);  
    RETURN COPIA;  
}
```

```
PROC LONGITUD (in & l: LISTADOBLEMENTEENLAZADA< T >): INT{  
    RETURN l.LONGITUD;  
}
```

```
PROC COPIAR (in & l: LISTADOBLEMENTEENLAZADA< T >): LISTADOBLEMENTEENLAZADA< T >{
```

```

VAR COPIA: LISTADOBLEMENTEENLAZADA<T> := NEW LISTADOBLEMENTEENLAZADA<T>();
VAR i: INT := 0;
VAR NODOCOPY: NODO<LISTA<T>> := l.PRIMERO;
WHILE(i < l.LONGITUD) DO:
    COPIA.AGREGARATRAS(COPIA, NODOCOPY);
    NODOCOPY := NODOCOPY.SIGUIENTE;
    i++;
END WHILE;
COPIA.ULTIMO := NEW NODO<LISTA<T>>(l.ULTIMO);
COPIA.LONGITUD := l.LONGITUD;
RETURN COPIA;
}

```

PROC CONCATENAR (INOUT l1: LDE<T>, IN l2: LDE<T>) {

```

VAR COPIA: LDE<T> := l2.COPIAN(l2);
l1.ULTIMO.SIGUIENTE := COPIA.PRIMERO;
l1.ULTIMO := COPIA.ULTIMO;
l1.LONGITUD += COPIA.LONGITUD;
}

```

}

PDM

MÓDULO PLAYAMANIOBRAS IMPLEMENTA PLAYADEMANIOBRAS {

VAR TRENES : LDE<LDE<STRING>> → TRENES.PRIMEROS.VALOR.PRIMEROS.VALOR
 ↗ LISTA DOBLEMENTE ENLAZADA
 ↗ DE TRENES X INSTANCIA
 VAR CAPMAX: INT

Complejidad O(1)

PROC ABRIRPLAYA (IN CAPACIDAD: INT): PLAYAMANIOBRAS {

NOD.TRENES := NEW LISTADOBLEMENTEENLAZADA<LDE<STRING>>();

NOD.CAPMAX := CAPACIDAD

}

↑ ACORTADO A CAPMAX => NÚMERO CHICO.

PROC RECIBIRTREN (INOUT P: PDM, IN VAGONES : LDE<STRING>): INT {
 VAR IND: INT := P.TRENES.AGREGAR EN LUGAR VACIO (P.TRENES, VAGONES);
 RETURN IND;
 }

Complejidad O(n) x la BÚSQUEDA ESPACIO LIBRE.

PROC DESPACHARTREN (INOUT P: PDM, IN IND TREN: INT) {
 VAR TREN := P.TRENES.OBTENER (P.TRENES, IND TREN);
 TREN.ELIMINARTODOS(TREN); \Rightarrow BOMBA VAGONES DEL TREN.
 } $\hookrightarrow O(1)$

Complejidad $O(P.TRENES.OBTENER(P.TRENES, IND TREN)) = O(m)$

PROC UNIRTRENES (INOUT P: PDM, IN IND TREN1: INT, IND TREN2: INT) {
 VAR TREN := P.TRENES.OBTENER (P.TRENES, IND TREN1); INSTANCIA DE 1 TREN. TIENE VAGONES
 VAR TREN2 := P.TRENES.OBTENER (P.TRENES, IND TREN2); INSTANCIA DE 2DO TREN. TIENE VAGONES
 TREN.CONCATENAR (TREN, TREN2);
 \hookrightarrow se concatena copia.

TREN2.ELIMINARTODOS(TREN2);
 } \checkmark

\nearrow TENDRÁ
SE SACA VAGON DE TREN [v] BOSCAR QUIEN TIENE EL VAGON.

\nearrow EL VAGON SE LE AGREGA A TREN [viADESTINO] \Rightarrow AL final BOMBA DE TREN ANT.
CUALQUIER POS.

PROC MOVERVAGON (INOUT P: PDM, IN VAGON: STRING, IN VIADESTINO: INT) { Complejidad $O(m \cdot m)$
 } \hookrightarrow TRENES VAGON
 VAR TREN: LDE<STRING> := P.TRENES.PRIMERO; \Rightarrow 1er TREN, TENDRÁ VAGONES.

VAR BORRADO: BOOL := false

VAR i: INT := 0;

VAR CopiaVagon: NODO LISTACT >

\nearrow TREN DONDE DEBEN PASAR VAGON

VAR TREN_DESTINO := P.TRENES.OBTENER (P.TRENES, VIADESTINO); \nearrow "VAGON E P.TRENES[VIADESTINO]"

WHILE (TREN != NULL AND !BORRADO) DO:

VAR VAGON_TREN := TREN.PRIMERO;

i := 0;

WHILE (VAGON != NULL AND !BORRADO) DO:

IF (VAGON_TREN == VAGON) THEN

COPIAVAGON := TREN. OBTENER(i)

TREN. ELIMINAR(TREN, i); *borro el vagón*

BORRADO := true

ELSE:

VAGONTREN := VAGONTREN. SIGUIENTE;

ENDIF

i++;

ENDWHILE

TREN := TREN. SIGUIENTE;

ENDWHILE

TRENDESTINO. AGREGARATRAS(TRENDESTINO, COPIAVAGON);

}

}

Diseño con Árboles binarios

Ejercicio 7. Implementamos un Árbol Binario (AB) con

```
struct<T> NodoAB<T>{
    val: T,
    izquierda: NodoAB<T>,
    derecha: NodoAB<T>,
}

class ArbolBinario<T> {
    var raiz: NodoAB<T> // "puntero" a la raíz del árbol
    ...
}
```

(El TAD está definido en el anexo al final de la práctica)

- Escriba en castellano el invariant de representación para este módulo
- Escriba los algoritmos para los siguientes procs y, de ser posible, calcule su complejidad
 - altura(in ab: ArbolBinario<T>): int // devuelve la distancia entre la raíz y la hoja más lejana
 - cantidadHojas(in ab: ArbolBinario<T>): bool
 - está(in ab: ArbolBinario<T>, int t: T): bool // devuelve true si el elemento está en el árbol
 - cantidadApariciones(in ab: ArbolBinario<T>, int t: T): int
 - ultimoNivelCompleto(in ab: ArbolBinario<T>): int // devuelve el número del último nivel que está completo (es decir, que tiene todos los nodos posibles). Considera la raíz como nivel 1

a) RAÍZ: NODO.

INVREP:

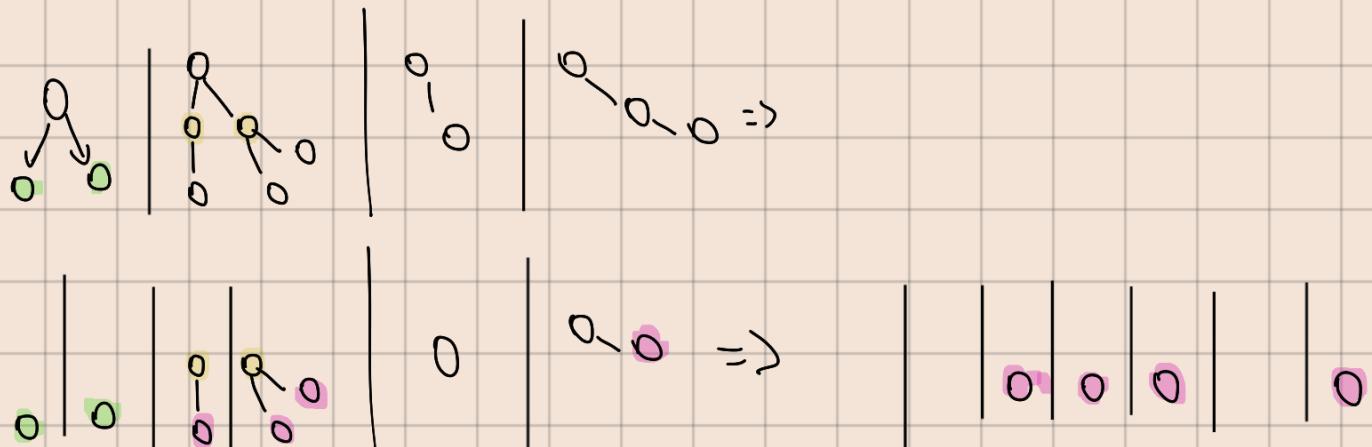
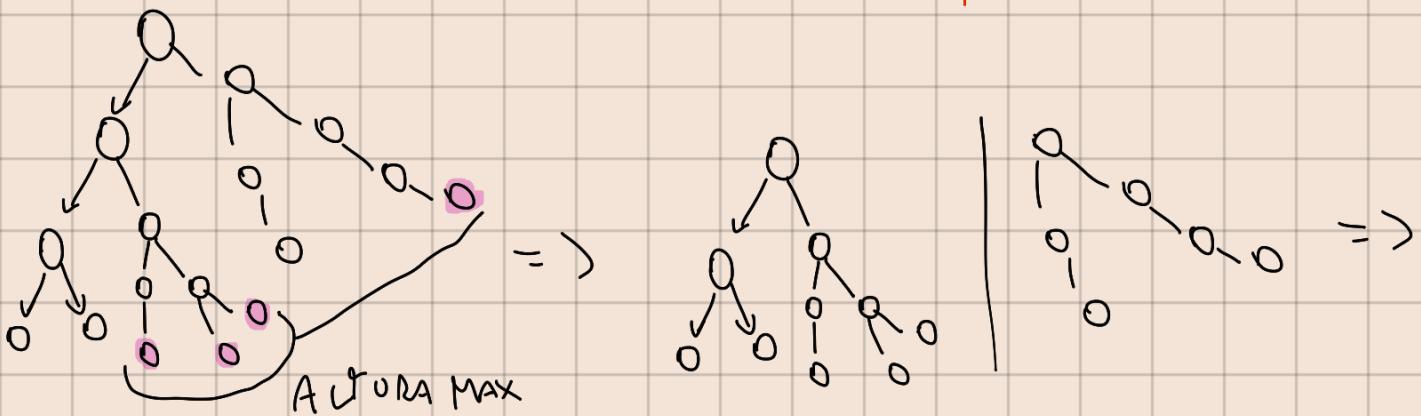
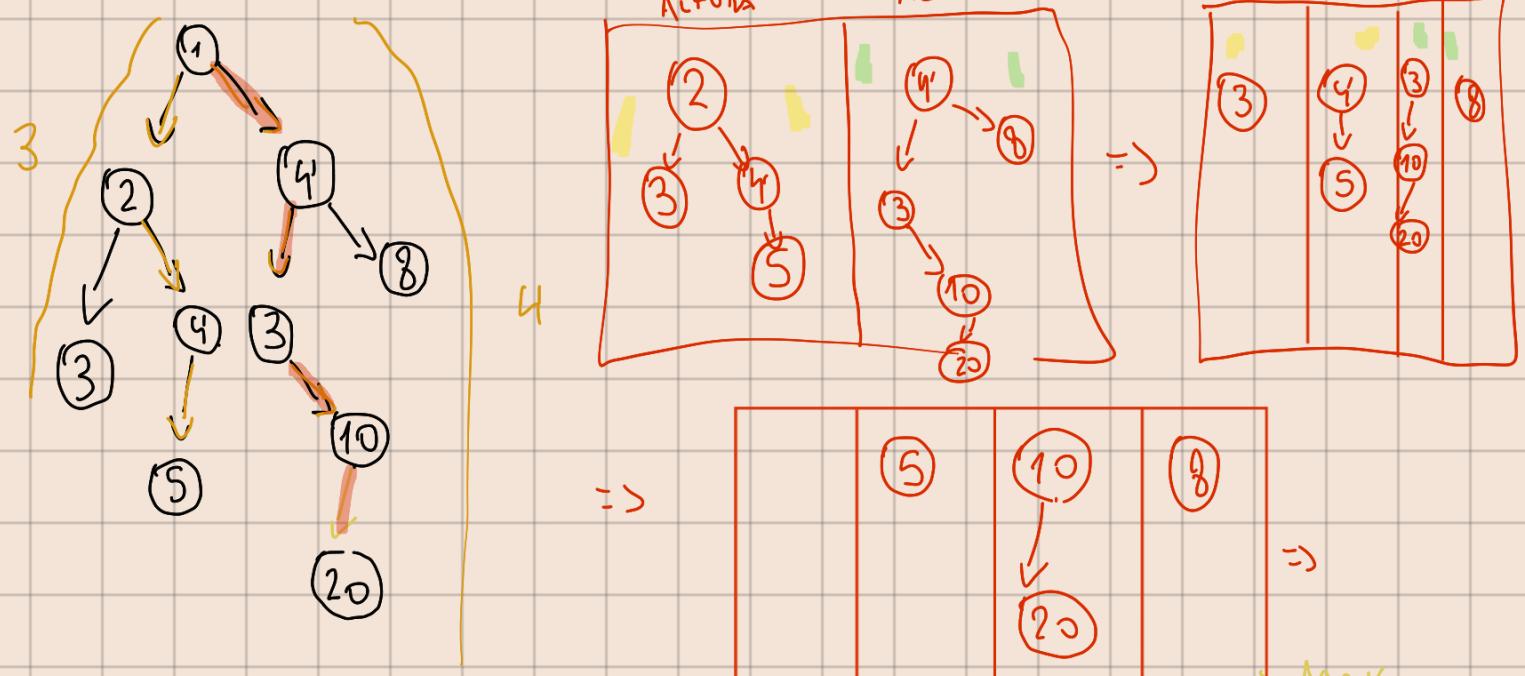
No tiene hijos. La única manera de ir desde raíz a una hijo es

parte uno por uno.

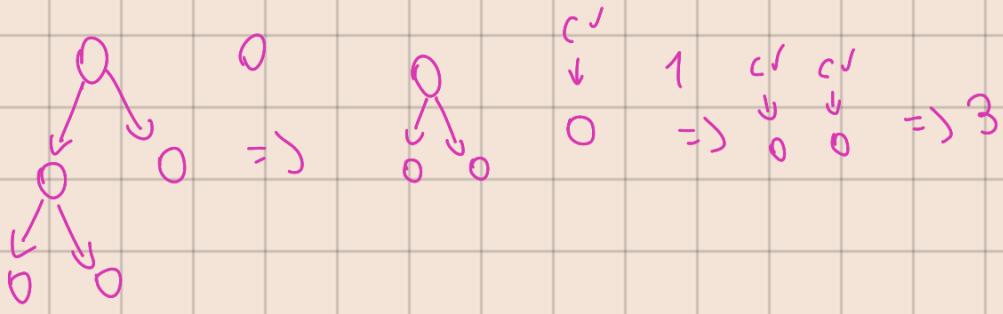
b)

1) Altura: Voz hacia izq y der. La cantidad mayor es la altura.

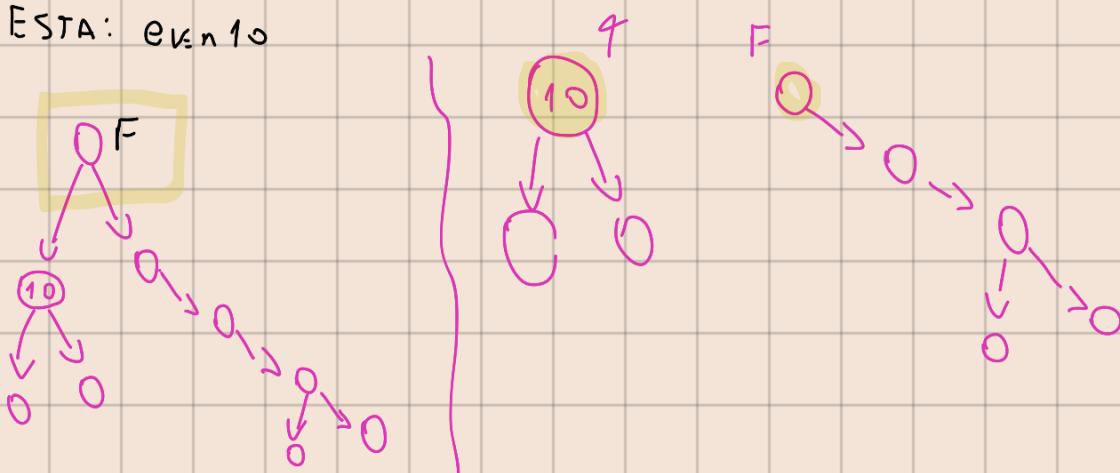
No tiene garantías misma altura de ambos lados (BALANCEADO).



CHASAC

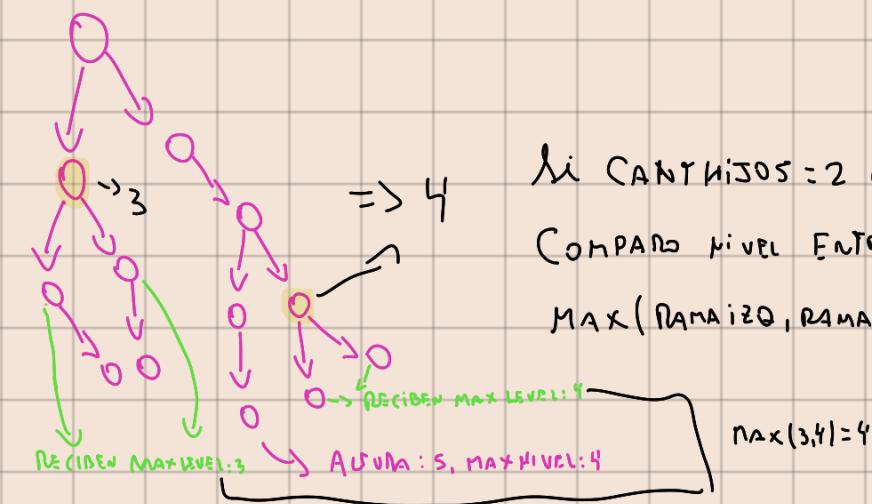


ESTA: $e \in n_{10}$



$$F \vee T \vee F = T$$

ULTIMO NIVEL COMPLETO



MÓDULO ARBOLBINARIO < \rightarrow IMPLEMENTA ARBOLBINARIO < \rightarrow {

VAR RAIZ: NODOAB< \rightarrow

PROC ALTURA(in AB: ARBOLBINARIO < \rightarrow): int {

VAR Nodo := AB.RAIZ;

VAR RES:int := AB.ALTRURAAUX(Nodo);

}

PROC ALTURAAUX(in Nodo: NODOAB < \rightarrow): int {

```

IF (NODO == NULL)
    RETURN -1;

ELSE:
    RETURN 1 + MAX(ALTURA(NODO.izquierda), ALTURA(NODO.derecha));
ENDIF

}

```

PROC CANTIDADHOJAS(*in* AB:ARBOLBINARIO<T>):int{

```

VAR NODO:NODOAB<T>:= AB.RAIZ;
RETURN CHOSASAUX(NODO);
}
```

PROC CHOSASAUX(*in* NODO:NODOAB<T>):int {

```

if (NODO == NULL) THEN
    RETURN 0;
ENDIF
if (NODO.derecha == NULL ^ NODO.izquierda == NULL) THEN
    RETURN 1;
ENDIF
```

```
RETURN CHOSASAUX(NODO.izquierda) + CHOSASAUX(NODO.derecha);
```

PROC ESTA(*in* AB:ARBOLBINARIO<T>, *in* e:T):bool {

```

VAR NODO:NODOAB<T>:= AB.RAIZ;
RETURN ESTAAUX(NODO, e);
}
```

PROC ESTAAUX(*in* NODO:NODOAB<T>, *in* e:T):bool{

```
if (NODO == NULL || NODO.VAL != e) THEN
```

```

RETUN FALSE;

ENDIF

IF(NODO.VAL == e) THEN
    RETURN TRUE;
ENDIF

RETURN ESTAAUX(NODO.izquierda,e) || ESTAAUX(NODO.derecha,e);
}

```

```

PROC CANTIDADAPARICIONES (IN AB:ARBOLBINARIO<T>, IN e:T):INT{
    VAR NODO:NODOAB<T>:= AB.RAIZ;
    RETURN CAPARICIONESAUX(AB,e);
}

```

```

PROC CAPARICIONESAUX (IN NODO:NODOAB<T>, IN e:T):INT{
    IF(NODO==NULL || NODO.VAL != e) THEN
        RETURN 0;
    ENDIF

    IF(NODO.VAL == e) THEN
        RETURN 1;
    ENDIF

    RETURN CAPARICIONESAUX(NODO.izquierda,e) + CAPARICIONESAUX(NODO.derecha,e);
}

```

```

PROC ULTIMONIVELCOMPLETO(IN AB:ARBOLBINARIO<T>):INT{
    VAR NODO:NODOAB<T>:= AB.RAIZ;
    RETURN UNC(NODO,1);
}

```

```

PROC UNC(IN NODO:ARBOLBINARIO<T>, IN NIVEL:INT):INT{
    IF(NODO==NULL) THEN
        RETURN NIVEL;
}

```

ENDIF

IF(NODO.DERECHA != NULL ^ NODO.IZQUIERDA != NULL) THEN

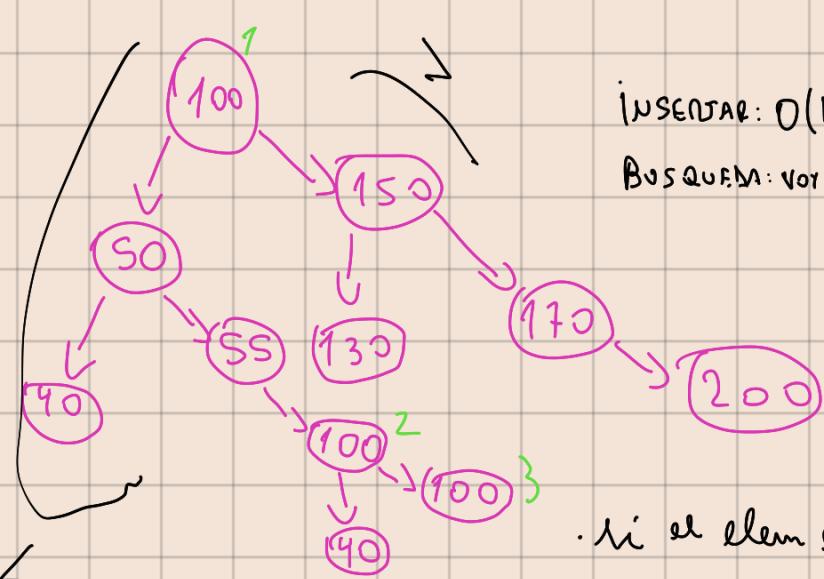
NIVEL:=NIVEL+1;

ENDIF

RETURN MAX(UNC(NODO.IZQUIERDA, NIVEL), UNC(NODO.DERECHA, NIVEL))

}

ABB:



INSERIR: $O(n)$

BUSQUEDA: VOT IZQ ODER

• Si el elem en raíz pude ser en izq.

• Si el elem no en raíz, me fijo si es mirroro

MENOR & ACT & VOT VIENDO CAMINOS.

ELIMINAR:

• En hoja: Bonito } • En niz un HIJO: Bonito

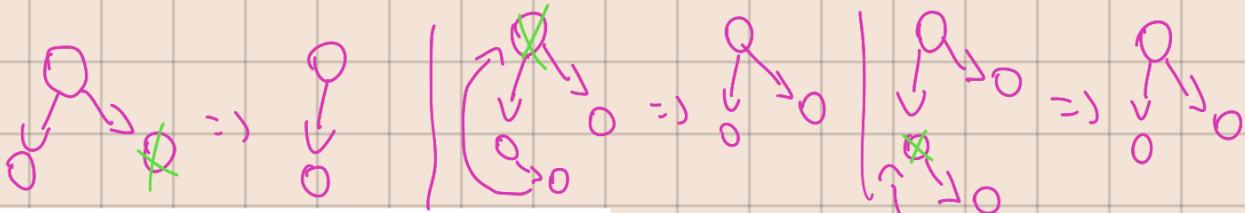
• En hoja 1 HIJO: CAMBIO HIJO X PADRE.

En herc PADRE=HIJO y BONITO HIJO

En herc 2 HIJOS: BUSCO MAS GRANDE DE LOS CHICOS.

CAMBIO MAS GRANDE X NODO A ELIM PERO APUNTANDO AIZQ Y DER

DE AL ELIMINAR



Ejercicio 8. Un Árbol Binario de Búsqueda (ABB) es un árbol binario que cumple que para cualquier nodo N , todos los elementos del árbol a la izquierda son menores o iguales al valor del nodo y todos los elementos del árbol a la derecha son mayores al valor del nodo, es decir

$$(Vi : \mathbb{Z})((estáPred(N.izq, i) \rightarrow i \leq N.val) \wedge (estáPred(N.der, i) \rightarrow i > N.val))$$

Implemente los algoritmos para los siguientes procs y calcule su complejidad en mejor y peor caso

a) `está(in ab: ABB<T>, int t: T): bool` // devuelve true si el elemento está en el árbol

b) `cantidadApariciones(in ab: ABB<T>, int t: T): int`

c) insertar(inout ab: ABB<T>, int t: T)
d) eliminar(inout ab: ABB<T>, int t: T) \rightarrow Doloroso. ¿Seja q Recorrido en ABB.
e) inOrder(in ab: ABB<T>) : Array<T> // devuelve todos los elementos del árbol en una secuencia ordenada

MÓDULO ABB IMPLEMENTA ABB<T>

VAR RAIZ: NODOABB<T>

~ PUBLIC

PROC ESTA(IN AB: ABB<T>, IN T:T): bool{

VAR NODO: NODOABB<T>:= AB.RAIZ;

MEJOR CASO $O(1)$

RETURN ESTAAUX(NODO,t);

PEOR CASO $O(M) \Rightarrow Q_{S_0}$

}

~ PRIVATE

PROC ESTAAUX(IN NODO: NODOABB<T>, IN E:T): bool {

IF(NODO==NULL) THEN

RETURN FALSE;

ENDIF

IF(NODO.VAL.(COMPARABLETO(e))>0) THEN

RETURN TRUE;

ENDIF

IF(e.COMPARABLETO(NODO.VAL)>0):

RETURN ESTAAUX(NODO.DERECHA,e);

ENDIF

RETURN ESTAAUX(NODO.IZQUIERDA,e);

}

~ PUBLIC

PROC CANTIDADAPARICIONES (IN AB: ABB<T>, IN E:T): INT{

VAR NODO: NODOABB<T>:= AB.RAIZ;

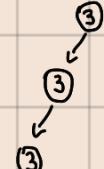
VAR CANTAP: INT:= 0;

MEJOR CASO = $O(n)$

IF(!NODO) THEN RETURN CANTAP ENDIF

PEOR CASO = $O(n) \Rightarrow$

CANTAP:= CANTAPAU(X(NODO,e,0));



RETURN CANTAP;

}

↑ PUBLIC

```
PROC CANTAPAU(X;IN NODO:NODOABB<T>,IN e:T,IN CANTAP:INT):INT{  
    IF(NODO==NULL){  
        RETURN CANTAP;  
    }  
    ENDIF  
    IF(e.COMPARABLETO(NODO.VAL)==0):  
        CANTAP:=CANTAP+1;  
    }  
    ENDIF  
  
    CANTAP:=CANTAPAU(X,NODO.DERECHA,e,CANTAP);  
    CANTAP:=CANTAPAU(X,NODO.IZQUIERDA,e,CANTAP);  
}  
}
```

↑ PUBLIC

```
PROC INSERTAR(OUT ABB:ABB<T>,IN e:INT){  
    VAR NODO:NODOABB<T>:=NEW NODOABB(e);  
    VAR RAIZ:NODOABB<T>:=ABB.RAIZ;  
    IF(RAIZ==NULL) THEN  
        ABB.RAIZ:=NODO;  
        RETURN;  
    }  
    ENDIF  
    INSERTARAUX(RAIZ,NODO);  
}
```

MEJOR (ASO: O(1))

¿θ seña O(log n) x si esté
BALANCEADO?
↳ YA SERÍA AVL.

PEOR (ASO: O(m))

No hay cond de corse.
IGNORAR ABB VACÍO
(ASO EXCEPCIONAL)



↑ PUBLIC

```
PROC INSERTARAUX(AB:NODOABB<T>,IN NODO:NODOABB<T>){  
    IF(NODO.VAL.COMPARABLETO(AB.VAL)<0) THEN
```

```

IF(AB.IZQUIERDA == NULL) THEN
    AB.IZQUIERDA := NODO;
ELSE
    INSERTARAUX(AB.IZQUIERDA, NODO);
ENDIF
ELSE
    IF(AB.DERECHA == NULL) THEN
        AB.DERECHA := NODO;
    ELSE
        INSERTARAUX(AB.DERECHA, NODO);
    ENDIF
ENDIF
}

```

¿ Si hay repetido (name igual); Borro 1 o todos ?

```

PROC BORRAR(inout ABB:ABB<T>, in e:T) {
    VAR RAIZ: NODOABB<T> := ABB.RAIZ;
    IF(RAIZ == NULL) RETURN;
    IF(RAIZ.VAL == e ^ RAIZ.IZQUIERDA == NULL ^ RAIZ.DERECHA == NULL) THEN
        ABB.RAIZ := NULL
        RETURN;
    ENDIF
    ABB.BORRAR_AUX(RAIZ, e);
}

```

MEJOR CASO: O(1)

PEOR CASO: O(m)

MEJOR CASO: O(m) si borro todos

```

PROC BORRAR_AUX (inout RAIZ:ABB<T>, in e:T) {
    VAR PADRE:ABB<T> := RAIZ;
    IF(RAIZ == NULL) THEN RETURN; ENDIF;
    IF(e.COMPARABLETO(RAIZ.VAL) > 0) THEN
        RETURN BORRAR_AUX(RAIZ.DERECHA, e);
    ELSE IF(e.COMPARABLETO(RAIZ.VAL) < 0) THEN
        RETURN BORRAR_AUX(RAIZ.IZQUIERDA, e);
    ELSE

```

```

ELSE
    IF( RAIZ.IZQUIERDA == NULL ^ RAIZ.DERECHA == NULL) THEN
        RAIZ := NULL
    ELSE IF( RAIZ.IZQUIERDA != NULL ^ RAIZ.DERECHA == NULL) THEN
        RAIZ := RAIZ.IZQUIERDA;
    ELSE IF( RAIZ.IZQUIERDA == NULL ^ RAIZ.DERECHA != NULL) THEN
        RAIZ := RAIZ.DERECHA;
        → TIENE 2 HIJOS
    ELSE
        VAR PREDECESORINM : NODOABB<T> := RAIZ.IZQUIERDA;
        VAR ANT : NODOABB<T> := RAIZ;

        WHILE( PREDECESORINM.DERECHA != NULL) Do:
            ANT := PREDECESORINM;
            PREDECESORINM := PREDECESORINM.DERECHA;
        END WHILE

        RAIZ.VAL := PREDECESORINM.VAL;
        IF( ANT != RAIZ) THEN
            ANT.DERECHA := PREDECESORINM.IZQUIERDA;
        ELSE
            ANT.IZQUIERDA := PREDECESORINM.IZQUIERDA;
        ENDIF
    ENDIF
}

```

REQUIERE: min 1 elem

```

PROC MAXIMO (IN ABB:ABB<T> | NODOABB<T> {
    VAR NODO := ABB.RAIZ;
    RETURN maximorecursivo(NODO);
}

```

```

PROC MAXIMORECURSIVO (IN RAIZ: NODOABB<T> | NODOABB<T> {

```



```

IF (RAIZ.DERECHA = NULL) THEN
    RETURN RAIZ
ELSE
    RETURN MAXIMO RECURSIVO(RAIZ.DERECHA);
ENDIF
}
}

```

AUL

Ejercicio 9. Asumiendo que el árbol está balanceado, recalcule, si es necesario, las complejidades en peor caso de los algoritmos del ejercicio 8.

Los Algoritmos quedan iguales por Ejemplo Con Los:



ESTÁ: MEJOR CASO $O(1)$ \Rightarrow En Raíz.

PEOR CASO $O(\log n)$

CANTIDAD APARICIONES: MEJOR = PEOR CASO = $O(\log n)$

dicho recorrer hacia un lado del árbol y
de no encontrar.

INSERTAR:

MEJOR = $O(1)$ PEOR = $O(\log n)$ (ON BALANCE)

ELIMINAR:

MEJOR = PEOR CASO = $O(\log n)$ (ON BALANCE)

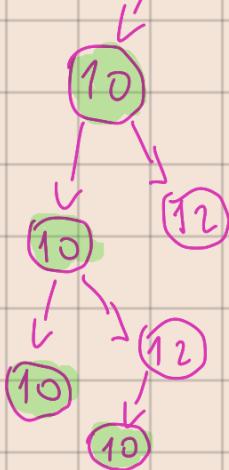
Ejercicio 10. ¿Qué pasa en un ABB cuando se insertan valores repetidos? Proponga una modificación del módulo que resuelva este problema

Uno de los problemas que se me ocurre es que cuando se tiene que eliminar ese valor con conteo, que nos quiere borrar si varios tienen ese valor.

Además, coluden como tantas operaciones se tendrían que borrar para borrar ese valor.

Además, tendríamos un gran desbalanceo en el peor caso.

↳



Un problema sería generar una lista ordenada con los valores (y)

La modificación sería que, a la hora de insertar valor en el elem hubiese un . El coste de un desbalanceo tardaría $O(n)$ en el peor caso de árbol.

PROC INSERTAR(*dato* ABB: ABB<T>, *e*: INT){

VAR NODO: NODOABB<T> := NEW NODOABB(e);

VAR RAIZ: NODOABB<T> := ABB.RAIZ;

IF (RAIZ == NULL) THEN

ABB.RAIZ := NODO;

RETURN;

ENDIF

IF (ABB.ESTA(e)) THEN

RETURN

ENDIF

INSERTARAUX (RAIZ, NODO);

}

