3. Pila

Pvacia: → pila

Apilar: elemento pila → pila

Parcial desapilar: pila → pila

Parcial cima: pila → elemento

Vacia?: pila → bool

4. cola

Cvacia: cola

añadir: elemento cola → cola

Parcial eliminar: cola → cola

Parcial primero: cola → elemento

vacia?: cola → bool

7. Listas

[]: → lista

_:_: elemento lista → lista {Añade por la izquierda}

parcial resto: lista → lista {Elimina primero}

Parcial eult: lista → lista {Elimina último}

Parcial prim: lista → lista

Parcial ult: lista → lista

Vacia?: lista → bool

4. Listas2

[_]: elemento → lista

_#_: elemento lista → lista {Añadir por la derecha}

_++_: lista lista → lista {concatenar 2 listas}

long: lista → natural

6. Lista+

Parcial _[_]: lista natural → elemento {devuelve el elemento de la posición n}

Parcial insertar: elemento lista natural → lista

Parcial modificar: elemento lista natural → lista

Parcial borrar: lista natural → lista

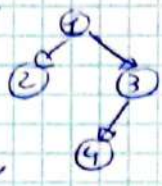
esta?: elemento lista → bool

buscar: elemento lista → natural

ARBOLES_binarios

- Padre
- Hijo
- Altura

Altura 0 

Altura 2 

- Profundidad de un sub arbol

- Δ : → a_bin
- _ • _ • _ : a_bin elemento a_bin → a_bin
- Parcial raiz: a_bin → elemento
- Parcial der : a_bin → a_bin
- Parcial izq : a_bin → a_bin
- Vacio? : a_bin → bool
- Parcial altura : a_bin → natural

Recorrido

Preorden : Se visita la raiz, los subarboles izquierdos y luego derechos

Postorden: Izquierdo , derecho , raiz

Inorden: Izquierdo, raiz , derecho

Arboles_binarios +

Preorden: a_bin → lista

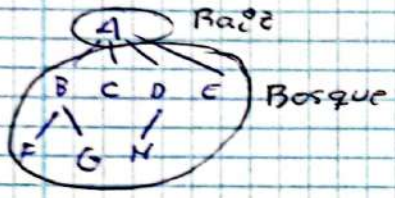
Postorden: a_bin → lista

Inorden: a_bin → lista

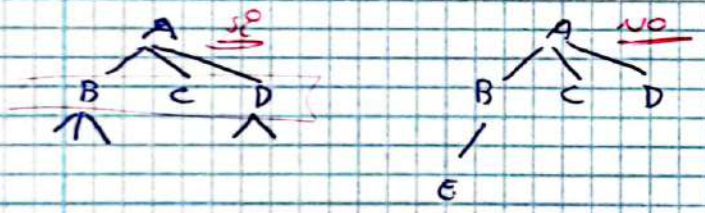
ejm

```
func vacio? (ab: a_bin): bool
  si (ab = null) entonces
     Devolver False
  sino
     Devolver True
  fsi
ffunc
```

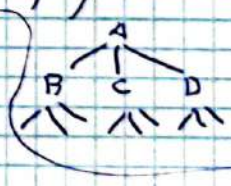
Árboles generales



- Nivel: Conjunto de nodos que están en la misma profundidad
- Grado de un árbol: Nº máximo de hijos que puede tener un árbol
- Árbol homogénea: Los subárboles tienen todos N hijos



- Árbol completo: Si todas sus hojas tienen la misma profundidad
- semicompleto:



- _ • _ : elemento bosque → árbol
- [ ] : → bosque
- _ : _ : árbol bosque → bosque {con bosques de arboles}
- raíz : árbol → elemento
- hijos : árbol → bosque
- vacío? : bosque → bool
- long : bosque → Natural {tamaño del bosque}
- num_hijos: árbol → Natural
- Parcial primero : bosque → árbol
- Parcial resto : bosque → bosque
- Parcial subárbol : árbol Natural → árbol
  {acceso al i-ésimo hijo de un arbol}

Árboles +

- preorden : árbol → lista
- prebosque : bosque → lista
- postorden : árbol → lista
- postbosque : bosque → lista
- hoja? : árbol → bool
- altura_arbol : árbol → Natural {altura de un árbol}
- altura_bosque : bosque → Natural {altura máxima del bosque}

```
func preorden (a: árbol): lista
  Devolver raiz(a) . prebosque (hijos(a))
ffun
func prebosque (b: bosque): lista
  si esvacio?(b) entonces
    Devolver [ ]
  sino
    Devolver preorden( primero(b)) ++ prebosque (resto(b))
  fsi
ffun
```

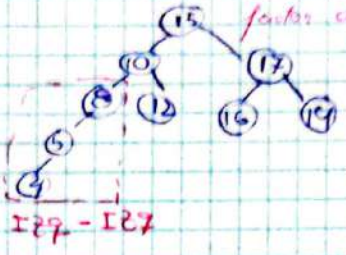
AVL

- Permiten operaciones de árboles de búsqueda
  estar insertar borrar

- Al insertar un elemento, miramos el factor de desequilibrio

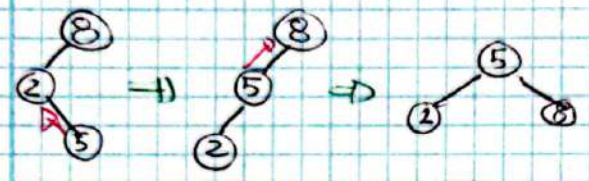
-2

- Izq - Izq
- Izq - Der



+2

- Der - Der
- Der - Izq



Izq - Der Izq - Izq

- Borrar
  - Si { es hoja / o / tiene 1 sob hijo } → se elimina directamente
  - Si 2 hijos, el nodo se reemplaza por uno de estos
    - Mayor de la rama izq
    - (puede ser necesario balancear)

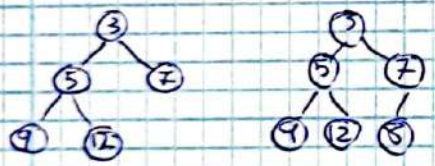
nodo_avl[pos]

altura : natural
Valor : elemento
izq : a_avl
Der : a_avl

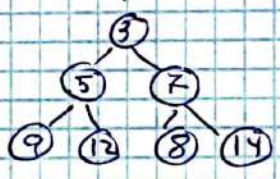
Montículos

- o el arbol es vacío ó la raiz es menor o igual que los hijos

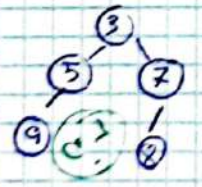
semicompleto



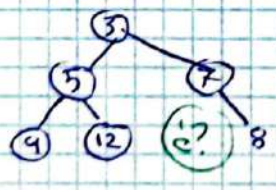
completo



no semicompleto



ó



- usa la operaciones de arboles_binarios

y

_ ≤ _ : elemento elemento → bool

es_completo? a_bin → bool

es_semicompleto? a_bin → bool

es_monticulo? a_bin → bool

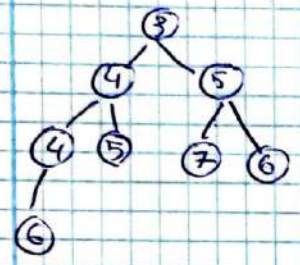
niveles a_bin → Natural {profundidad}

menor_igual? elemento a_bin → bool

es_semicompleto? $(i \cdot x \cdot d) = (es\_completo?(i \cdot x \cdot d)) \vee (es\_completo(i) \wedge es\_semicompleto?(d) \wedge (niveles(i) == niveles(d)) \vee (es\_semicompleto?(i) \wedge es\_semicompleto?(D) \wedge (niveles(I) == niveles(D)+1))$

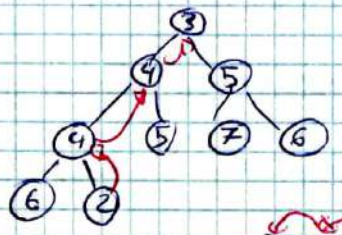
- La raíz ocupa primera posición del vector
- el nodo de la posición i-ésima, tiene su hi en $i \cdot 2$ y el hd en $i \cdot 2 + 1$
- El padre del nodo estáen la posición $\lfloor i/2 \rfloor$ $\lfloor 9/2 \rfloor = 4$

- Al insertar un elemento, hay que flotar el elemento
- Al eliminar " " , " " hundir el último elemento

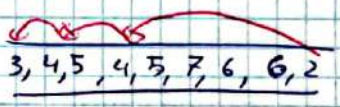


3, 4, 5, 4, 5, 7, 6, 6

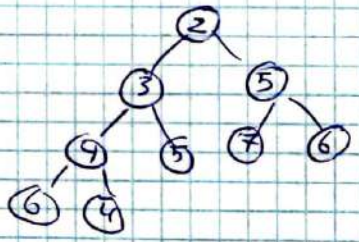
Insertar 2



Flotar

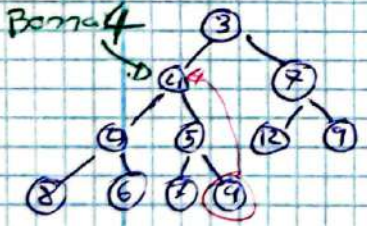


3, 4, 5, 4, 5, 7, 6, 6, 2



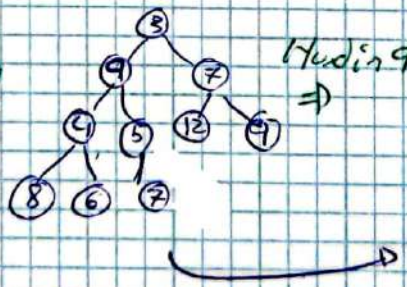
2, 3, 5, 4, 5, 7, 6, 6, 4

Borra 4

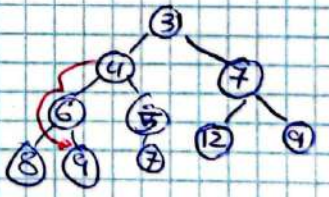


3, 4, 7, 4, 5, 12, 9, 8, 6, 7, 9

Subir 9



Hundir 9



Se intercambia con el hijo menor todas las veces que se necesite

```
Proc flotar( m:monticulo, i: 1,...tamaño)
  mientras (i>1) ∧ (m.datos[i div 2] > m.datos[i]  hacer
     intercambiar (m.datos[ i div 2], m.datos [i])
     i = i div 2
  fmientras
fproc
```

- es_completo?(Δ) = T
- es_completo?(i · X · d) = es_completo?(i) ∧ es_completo?(d) ∧ (
  niveles(i) == niveles(d))

- es_semicompleto(Δ) = T
- es_semicompleto(i · X · d) = (escompleto(i · X · d) ∨ (

(es_completo?(i) ∧ es_semicompleto?(d) ∧ (niveles(i) == niveles(d))

∨

(es_semicompleto?(i) ∧ es_completo?(d) ∧ (niveles(i) == (niveles(d)+1))

- es_monticulo?(Δ) = T
- es_monticulo?(i · X · d) = es_semicompleto(i · X · d) ∧
  menor_igual?(X, i) ∧ menor_igual?(X, d) ∧ es_monticulo?(i) ∧
  es_monticulo(d)

10

9 8 • Montículo de máximos

7 6 5 4