AVL.pdf

ED Práctica 5 - Arboles

2º Estructuras de Datos

Grado en Ingeniería Informática

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación
Universidad de Granada

KEEP CALM AND ESTUDIA UN POQUITO

WUOLAH + #QuédateEnCasa

ARBOLES EQUILIBRADOS (AVL)

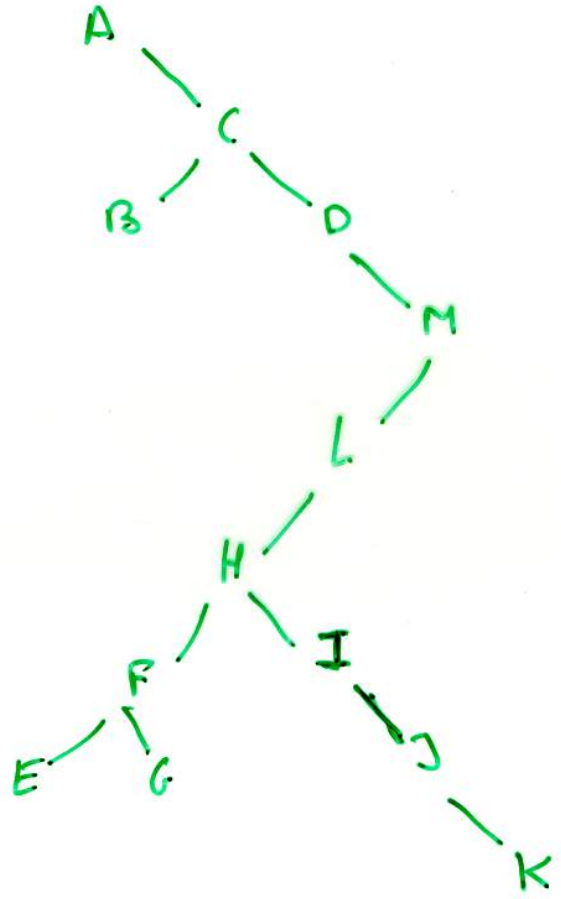
MOTIVACION

A VECES EN LA CONSTRUCCION DE LOS ABB SE LLEGA A ARBOLES CON CARACTERISTICAS POBRES PARA LA BUSQUEDA

P. EJ. SI CONSTRUIMOS UN ABB PARA

{A, C, D, M, L, H, I, B, F, G, J, K, E}

RESULTA:



QUE CONSTITUYE UN ARBOL MUY POCO BALANCEADO

IDEA:

CONSTRUIR ABB EQUILIBRADOS, NO PERMITIENDO QUE EN NINGUN NODO LAS ALTURAS DE SUS SUBARBOLES IZQDO Y DRCHO, DIFIERAN EN MAS DE UNA UNIDAD

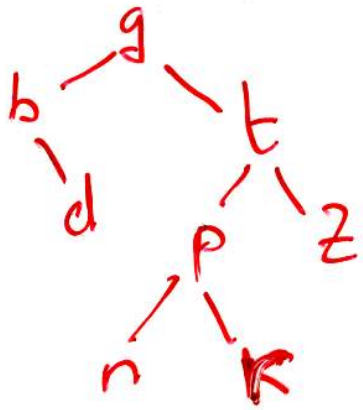
↓

ARBOLES AVL

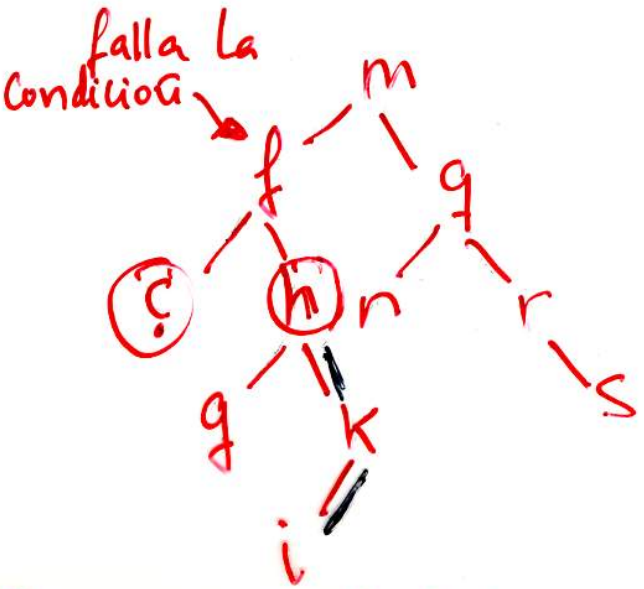
DEFINICION

DIREMOS QUE UN ARBOL BINARIO ABB ES UN AVL (O QUE ESTÁ EQUILIBRADO EN EL SENTIDO DE ADDELSON-VELSKI-LANDIS SI PARA CADA UNO DE SUS NODOS OCURRE QUE LAS ALTURAS DE SUS DOS SUBARBOLES DIFIEREN COMO MUCHO EN 1.

EJEMPLO



AVL (ABB + Equilibrio)



NO AVL
(es ABB pero no está equilibrado)

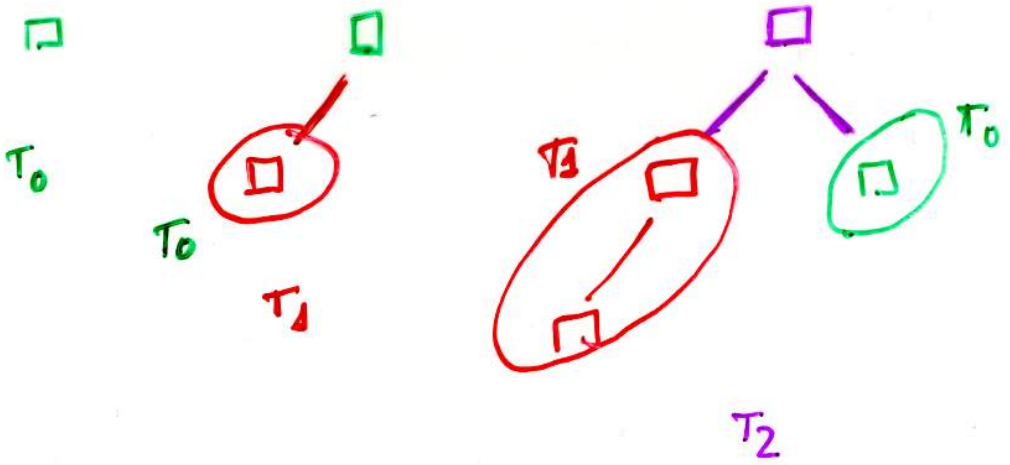
Nos interesan funciones para:

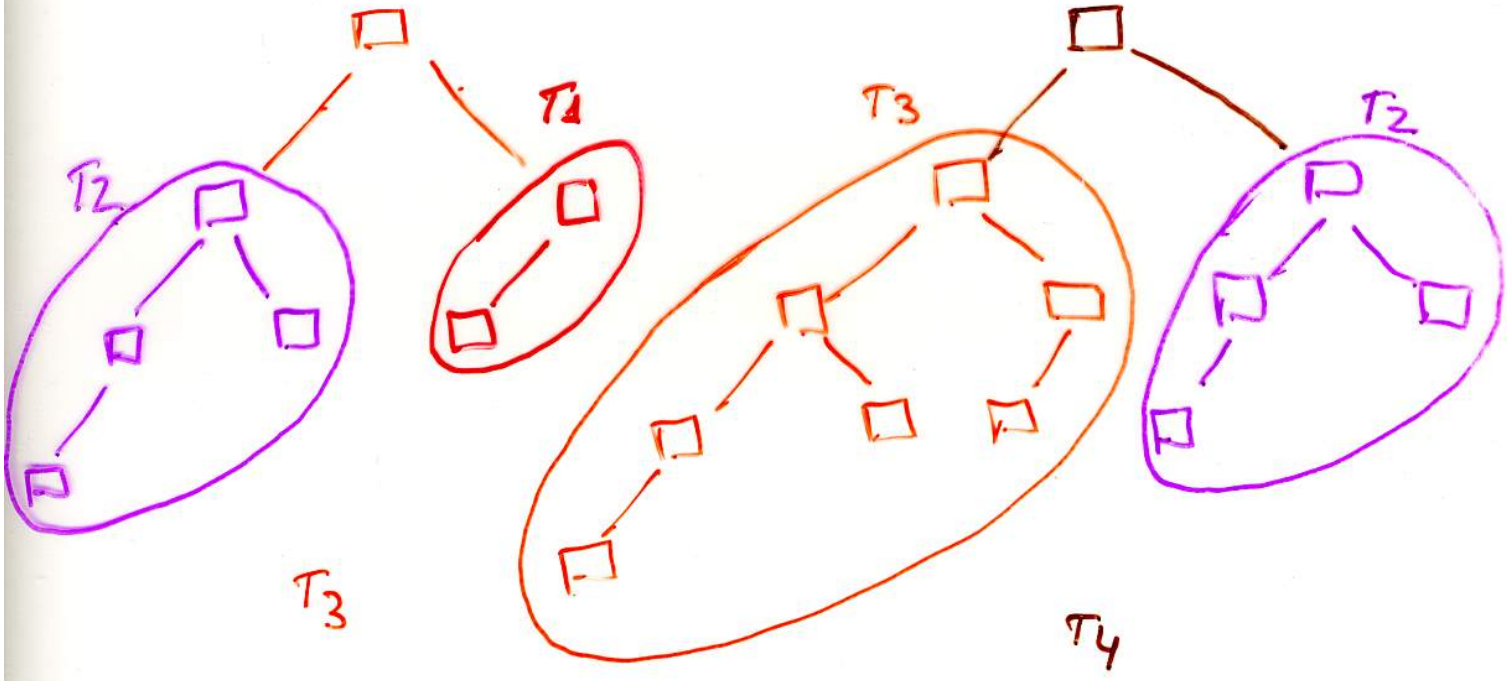
- pertenencia
- insertar
- borrar

teniendo en cuenta que habremos de construir funciones auxiliares que permitan realizar las operaciones manteniendo el árbol equilibrado

IMPLEMENTACION

PREGUNTA: ¿ES POSIBLE EFECTIVAMENTE CONSTRUIR AVL?





$$n(T_h) = 1 + n(T_{h-1}) + n(T_{h-2})$$ → Parecida a Fibonacci

Resolviendo: $\log_2(n+1) \leq h < 1.44 \log_2(n+2) - 0.33$

La altura h, en el árbol AVL T_h con mínimo número de nodos, es tal que nunca excede al 44% de la altura del árbol completamente equilibrado correspondiente.

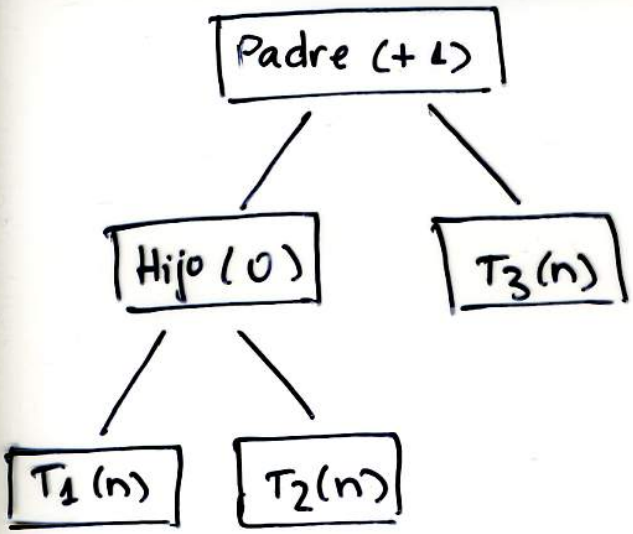
EQUILIBRIO EN LA INSERCION Y BORRADO

Idea:

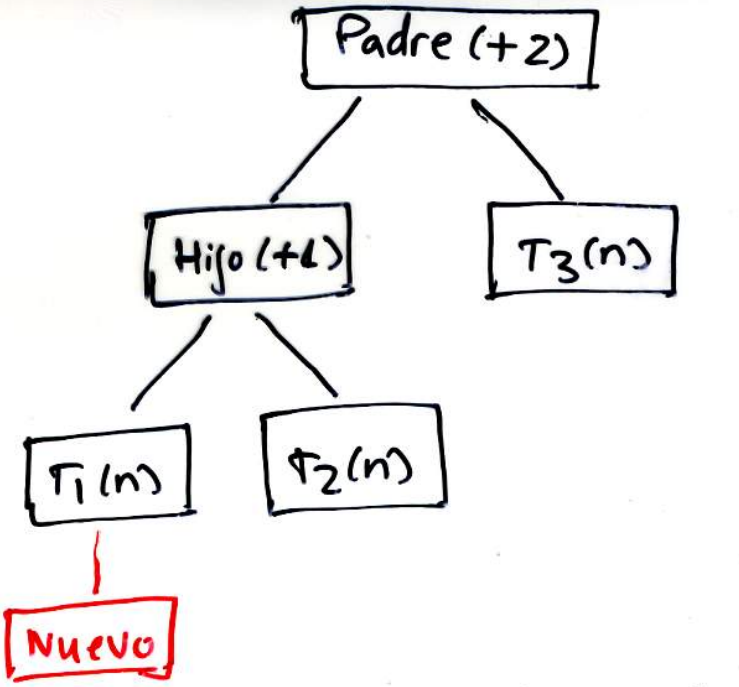
Usar un campo [altura] en el registro que represente a cada uno de los nodos del AVL para ~~determinar~~ el factor de equilibrio (diferencia de altura entre los subárboles rgdo y drcho), de forma que cuando esa diferencia sea >1 se hagan los reajustes necesarios de los punteros para que tenga diferencia de alturas ≤ 1

Veamoslo con una serie de ejemplos en que mostramos todos los casos posibles:

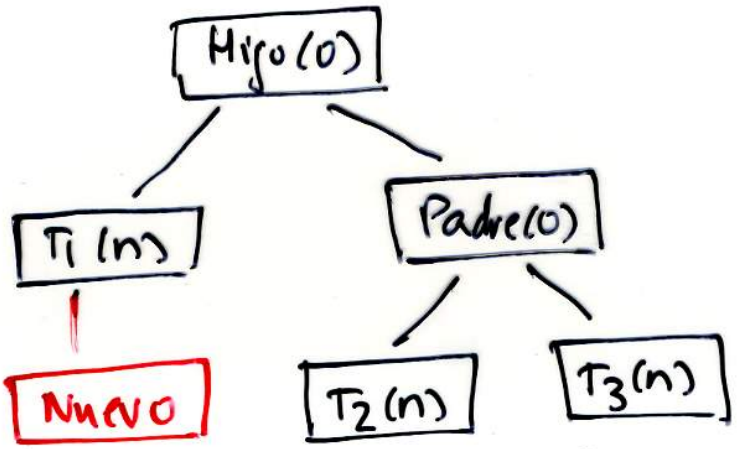
- Notaremos a los subárboles como T_k anotando entre paréntesis su altura
- Notaremos al factor de equilibrio como un valor con signo situado entre paréntesis al lado de cada nodo padre o hijo
- Las dos situaciones posibles que pueden presentarse son:
  - ROTACIONES SIMPLES
  - ROTACIONES DOBLES



AVL Inicial



Arbol desequilibrado tras insertar



AVL final

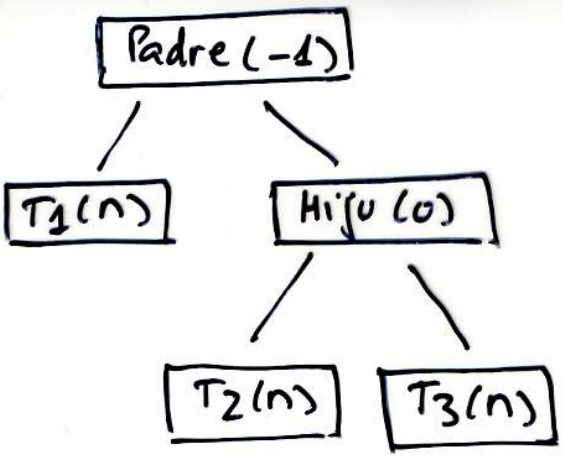
Reajustes necesarios para hacer la rotación:

```
padre → izqda = hijo → drcha;
hijo → drcha = padre;
```

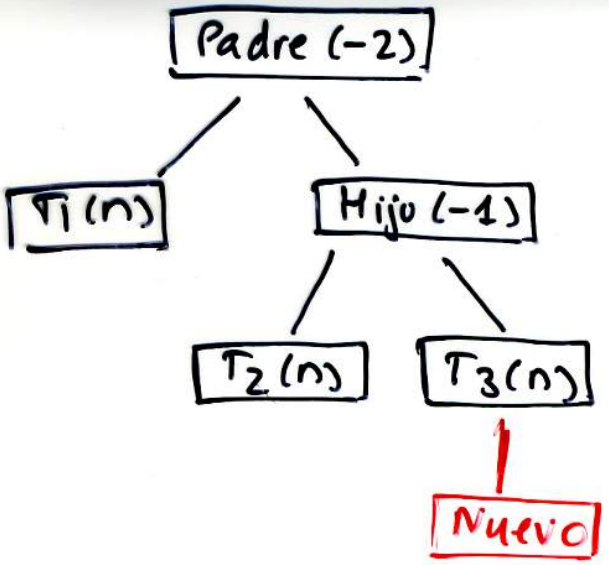
(ROTACION SIMPLE A LA DRCHA)

a) SE PRESERVA EL INORDEN

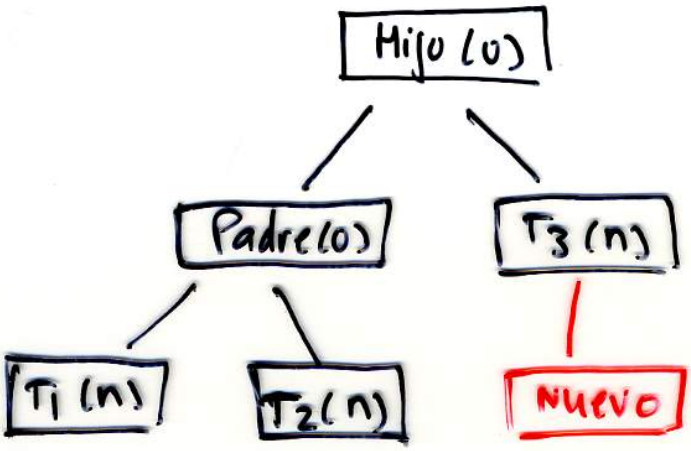
b) ALTURA DEL ARBOL FINAL = ALTURA DEL ARBOL INICIAL



AVL INICIAL



AVL desequilibrado tras insertar



AVL Final

Reajustes necesarios para hacer la rotación:

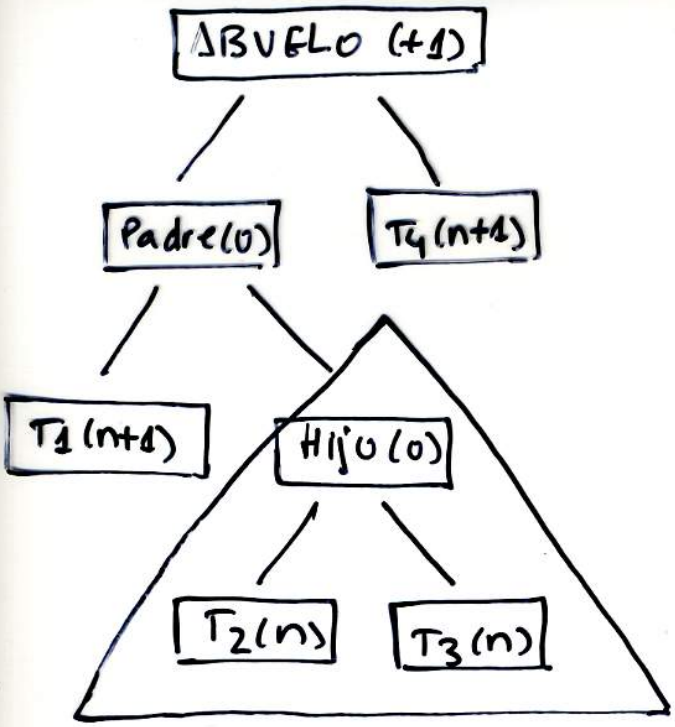
padre → drcha = hijo → Izqda;

hijo → Izqda = padre;

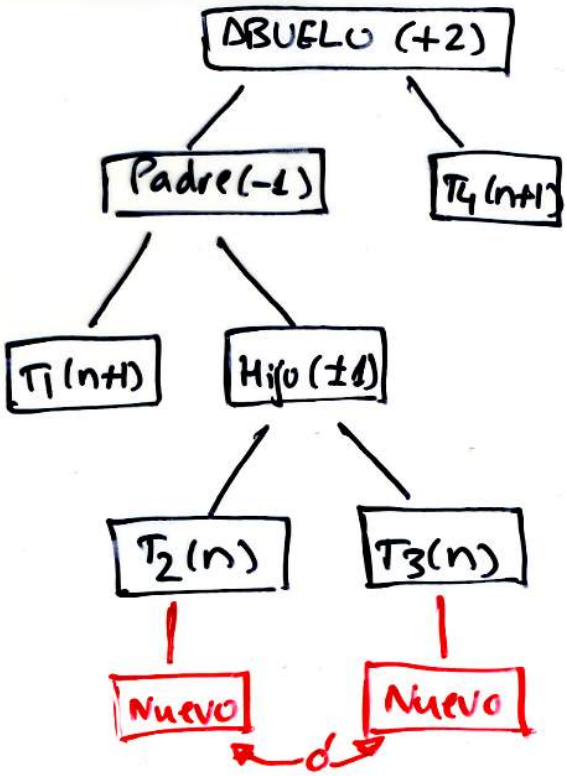
(ROTACION SIMPLE A LA IZQDA)

a) SE PRESERVA EL INORDEN

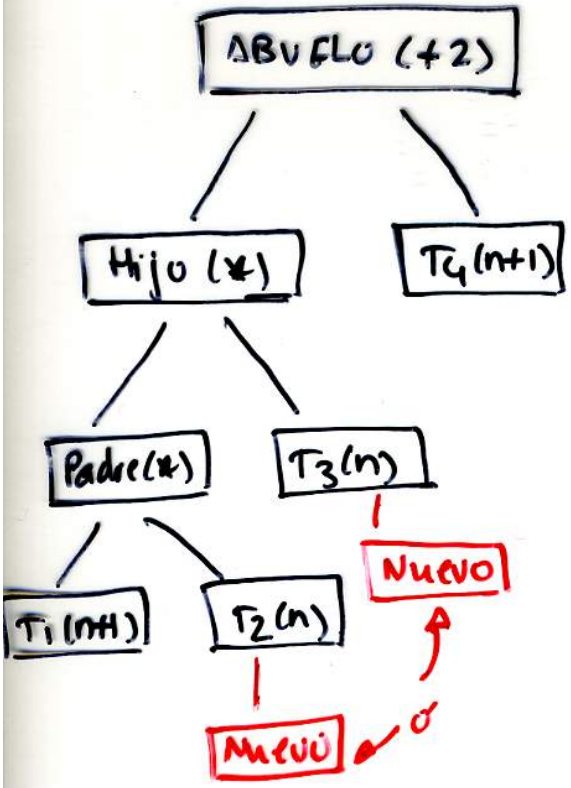
b) ALTURA DEL ARBOL FINAL = ALTURA DEL ARBOL INICIAL



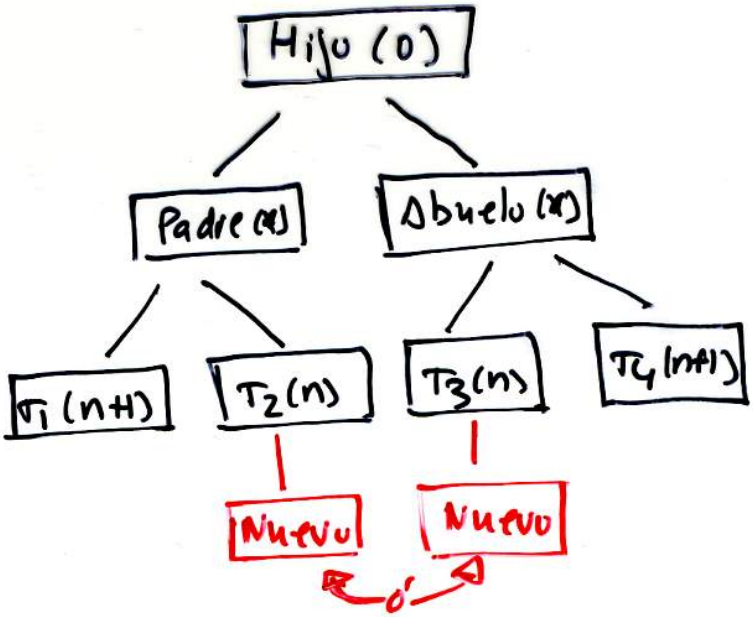
AVL inicial



Arbol desequilibrado tras insertar un nuevo elemento

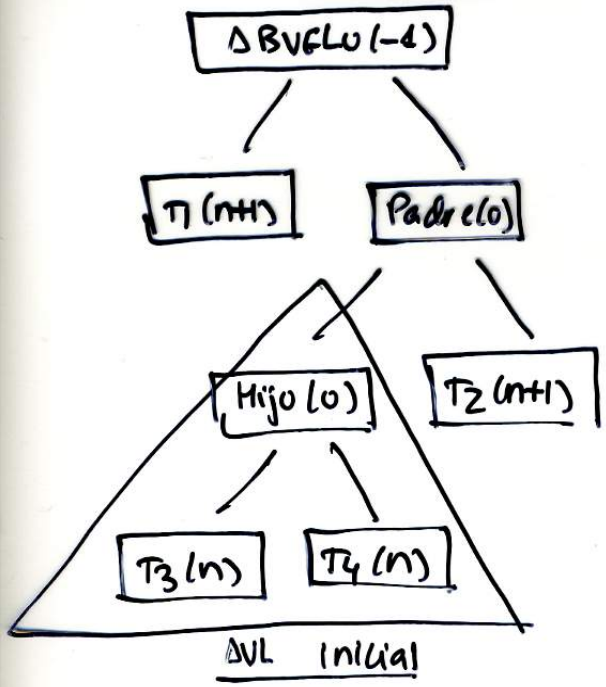


Rotación simple a la izqda en padre

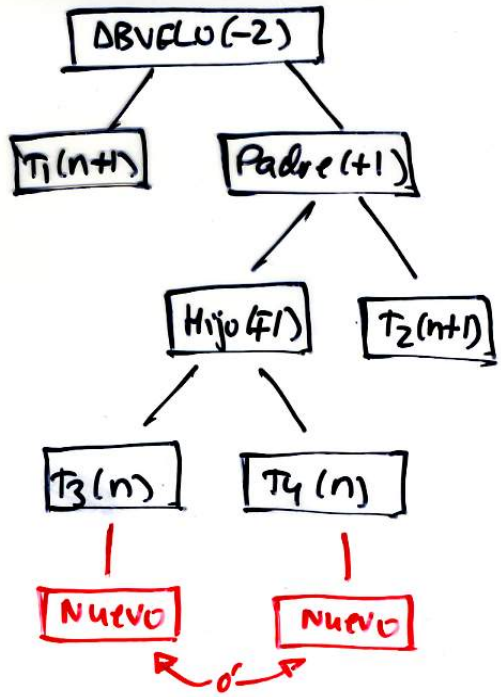


rotación simple a la drcha en abuelo

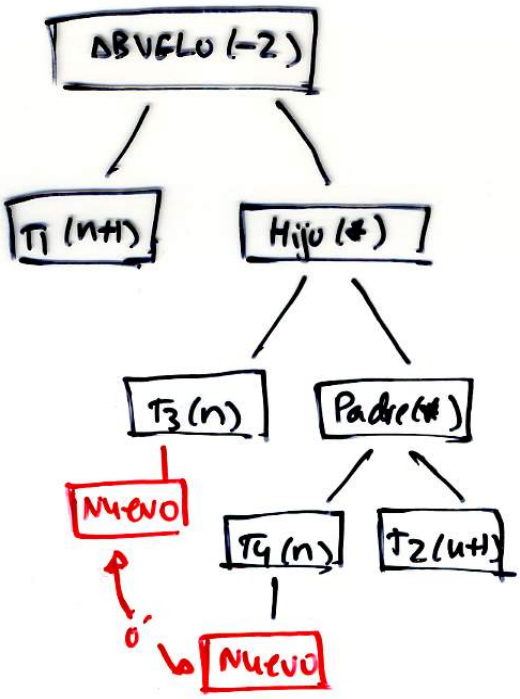
(ROTACIÓN DOBLE A LA DRCHA)



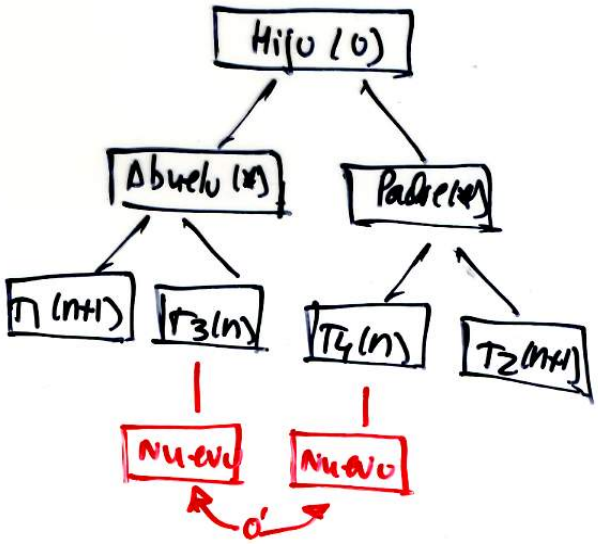
ΔVL inicial



Árbol desequilibrado tras insertar un nuevo elemento

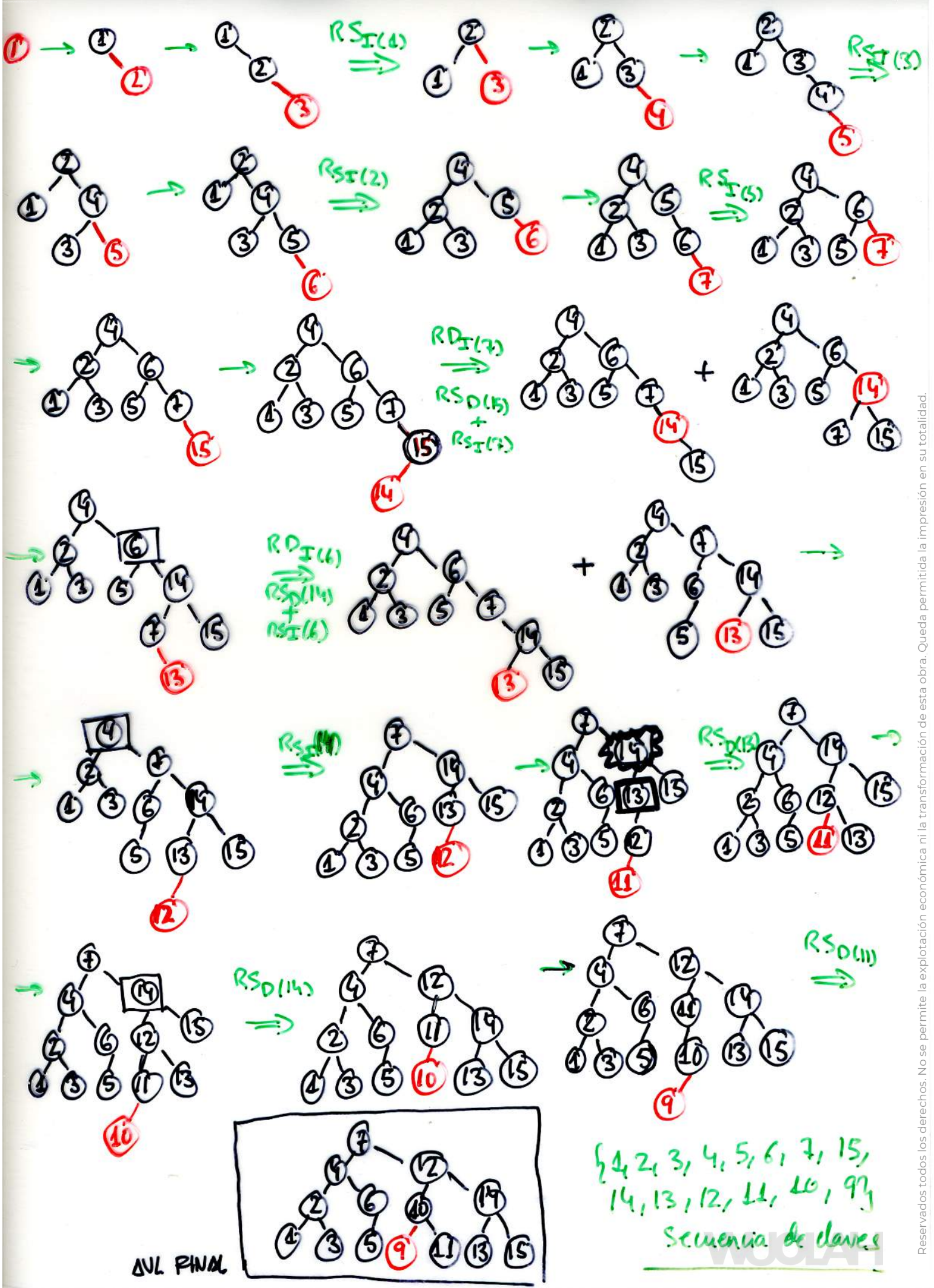


Rotación simple a la drcha en padre



Rotación simple a la izda en hijo

(ROTACION DOBLE A LA IZDA)



Árboles equilibrados AVL

- Son árboles binarios de búsqueda equilibrados. Las operaciones de inserción y borrado tienen un orden de eficiencia logarítmico.
- Se caracterizan porque para cada nodo se cumple que la diferencia de la altura de sus dos hijos es como mucho de una unidad.
- La especificación coincide con la del Árbol binario de búsqueda.
- La implementación varía en las operaciones que modifican la altura de un nodo: insertar y borrar.

Implementación

```
template <class Tbase>
void AVL<Tbase>::ajustarArbol
  (ArbolBinario<Tbase>::Nodo &n)
{
  int aIzda;
  int aDcha;
  ArbolBinario<Tbase>::Nodo hIzda, hDcha;

  // Ajustamos desde n hasta la raíz del árbol
  while (n!=ArbolBinario<Tbase>::NODO_NULO) {
    aIzda = altura(arbolb.HijoIzqda(n));
    aDcha = altura(arbolb.HijoDrcha(n));

    if (abs(aIzda-aDcha)>1) // Hay que ajustar
      if (aIzda>aDcha) {
        hIzda = arbolb.HijoIzqda(n);
        if (altura(arbolb.HijoIzqda(hIzda)) >
    altura(arbolb.HijoDrcha(hIzda)))
          rotarHijoIzqda(n);
        else {
          rotarHijoDrcha(hIzda);
```


```
            rotarHijoIzqda(n);
          }
        }
        else { // Exceso de altura por la dcha
          hDcha = arbolb.HijoDrcha(n);
          if (altura(arbolb.HijoIzqda(hDcha)) >
      altura(arbolb.HijoDrcha(hDcha))) {
            rotarHijoIzqda(hDcha);
            rotarHijoDrcha(n);
          }
          else
            rotarHijoDrcha(n);
        }

      n = arbolb.Padre(n);
    }
  }

  template <class Tbase>
  void AVL<Tbase>::rotarHijoIzqda
    (ArbolBinario<Tbase>::Nodo &n)
  {
    assert(n!=ArbolBinario<Tbase>::NODO_NULO);

    char que_hijo;
```

simétrico rotarHijoDrcha

```
ArbolBinario<Tbase>::Nodo elPadre =
  arbolb.Padre(n);

ArbolBinario<Tbase> A;
arbolb.PodarHijoIzqda(n, A);

ArbolBinario<Tbase> Aux;
if (elPadre!=ArbolBinario<Tbase>::NODO_NULO)
  if (arbolb.HijoIzqda(elPadre)==n) {
    arbolb.PodarHijoIzqda(elPadre, Aux);
    que_hijo = IZDA;
  }
  else {
    arbolb.PodarHijoDrcha(elPadre, Aux);
    que_hijo = DCHA;
  }
else
  Aux = arbolb;

ArbolBinario<Tbase> B;
A.PodarHijoDrcha(A.Raiz(), B);

Aux.InsertarHijoIzqda(Aux.Raiz(), B);
A.InsertarHijoDrcha(A.Raiz(), Aux);
```

doble o simple ?

```
  if (elPadre!=ArbolBinario<Tbase>::NODO_NULO) {
    if (que_hijo==IZDA) {
      arbolb.InsertarHijoIzqda(elPadre, A);
      n = arbolb.HijoIzqda(elPadre);
    }
    else {
      arbolb.InsertarHijoDrcha(elPadre, A);
      n = arbolb.HijoDrcha(elPadre);
    }
  }
  else {
    arbolb = A;
    n = arbolb.Raiz();
  }
}

template <class Tbase>
void AVL<Tbase>::rotarHijoDrcha
  (ArbolBinario<Tbase>::Nodo &n)
{
  assert(n!=ArbolBinario<Tbase>::NODO_NULO);

  char que_hijo;
```

```
ArbolBinario<Tbase>::Nodo elPadre =
  arbolb.Padre(n);

ArbolBinario<Tbase> A;
arbolb.PodarHijoDrcha(n, A);

ArbolBinario<Tbase> Aux;
if (elPadre!=ArbolBinario<Tbase>::NODO_NULO)
  if (arbolb.HijoIzqda(elPadre)==n) {
    que_hijo = IZDA;
    arbolb.PodarHijoIzqda(elPadre, Aux);
  }
  else {
    que_hijo = DCHA;
    arbolb.PodarHijoDrcha(elPadre, Aux);
  }
else
  Aux = arbolb;

ArbolBinario<Tbase> B;
A.PodarHijoIzqda(A.Raiz(), B);

Aux.InsertarHijoDrcha(Aux.Raiz(), B);
A.InsertarHijoIzqda(A.Raiz(), Aux);
```


```
    if (elPadre!=ArbolBinario<Tbase>::NODO_NULO) {
      if (que_hijo==IZDA) {
        arbolb.InsertarHijoIzqda(elPadre, A);
        n = arbolb.HijoIzqda(elPadre);
      }
      else {
        arbolb.InsertarHijoDrcha(elPadre, A);
        n = arbolb.HijoDrcha(elPadre);
      }
    }
    else {
      arbolb = A;
      n = arbolb.Raiz();
    }
}
```