Estructuras de Datos y Algoritmos 1

Tema 3 - Árboles

TreeSet<T>
TreeMap<K, V>
} Árbol { ÁRBOLES GENERALES (N-ario)
ÁRBOLES BINARIOS
ÁRBOLES BINARIOS CADENADOS { EQUILIBRADOS → AVL<T>
NO EQUILIBRADOS → RN<T>

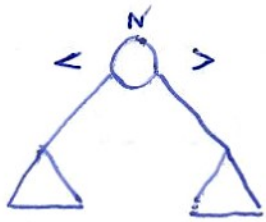
T = K + V

Idea Base

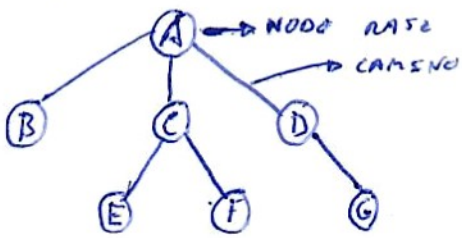
ED Lineal (O(N)) → ~~"Orden" O(n log n) Sort()~~ NO ES VÁLIDO

Solución: Árbol Binario + Orden O(log n)

ABB - Árbol Binario de Búsqueda

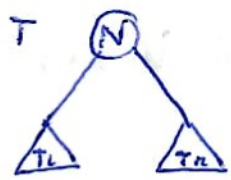


Un mapa es una colección de pares

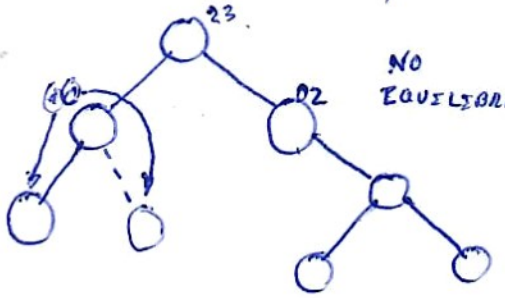


Definición de árbol

T: { ∅
N { Tl (TreeLeft)
Tr (TreeRight) } > Tipo T



AB balanceado o equilibrado: Árbol que difieren en 1 nivel (RESTA DE NODOS 1-0=+1 | 0-2=-2)



Implementación AB

```
Class Node <T> {
  T nodeValue;
  Node <T> izq;
  Node <T> der;
}
```

CÓMODO

```
Class arbolBinario <T> {
  Node <T> raiz;
}
```

Clase Par <K, V> (Paquete auxiliar)

T = K + V

```
public class Par <K, V> implements Comparable <Par <K, V>> {
  private K Key;
  private V Value;
  public Par <K, V> (K Key, V Value) {
    this.Key = Key;
    this.value = Value;
  }
  public K getKey () { }
  public V getValue () { }
  public V setValue (V Value) { }
```

COMPARABLE <K>

¿Pares son iguales si sus K lo son, donde igual al valor

Ejemplo:

```
Par<Integer, ArrayList<Integer>> par = new Par<>(3, null);
```

```
Syso(Par) -> 3, <null>
*{ par.setValue(new ArrayList<>());
   par.getValue().add(3);
```

```
* ArrayList<Integer> aux;
  par.getValue(aux = new ArrayList<>());
  aux.add(3);
  syso(par) -> 3 <[3]>
  par.setValue(null);
  syso(par) -> 3, <null>
  syso(aux) -> [3]
```

① new
② aux - dir
③ setValue(aux)

```
ArrayList<Par<String, ArrayList<Integer>>> arrPares = new ArrayList<>();
```
↳ T = Par<String, ArrayList<Integer>>

```
// 100. M
  int pos ← indexOf(T);
// Buscamos a "Pepe"
  int pos = arrPares.indexOf(new Par<>("Pepe", null);
// Utilizamos .get + .find ("Pepe")
  arrPares.get(pos).getValue().add(0);
```
pos+1

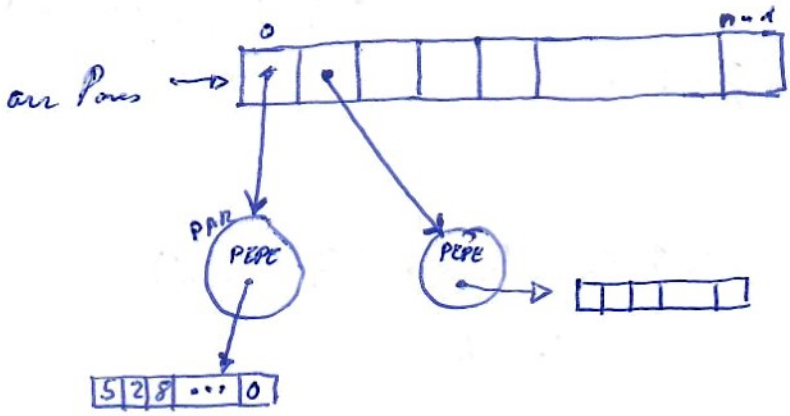
ArrayList<Integer>

Hacer método add:

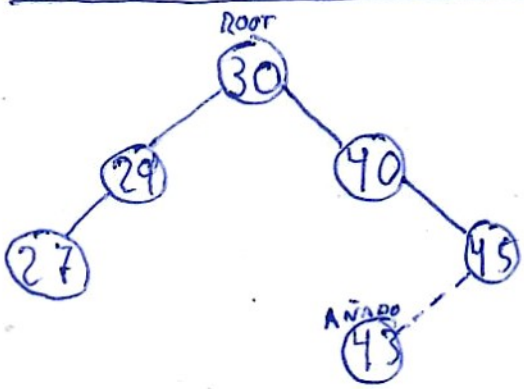
```
public void add (String nombre, Integer nota) {
  int pos = arrPares.indexOf (new Par <> (nombre, null));
  if (pos == -1) {
    arrPares.add (new Par <> (nombre, new ArrayList <> ()));
    pos = arrPares.size () - 1;
  }
  arrPares.get (pos).getValue().add (nota);
}
```

Dibujo de la estructura

ArrayList <Par <String, ArrayList <Integer>>> arrPares = new ArrayList <> ();



Recorrido sobre arboles binarios



27, 28, 29, 30, 40, 45

Añado 43

Mirar hoja adjunta ⚠

Árboles Binarios

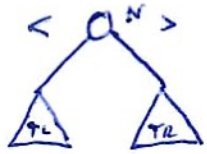
¿Cómo inserto? En orden → Coste $O(\log(n))$

¿Como elimino?

¿Como busco? $O(n)$

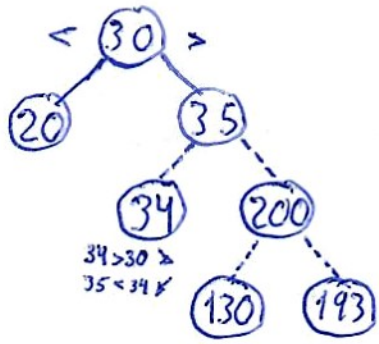
AB + Orden Total

ABB



→ No se existen elementos.

¿Cómo inserto? +34, +200, +130, +193 Se insertan como hoja.



¿Es de altura logarítmica?

$\log(8) = 3$

No es logarítmica, hay ni de 3 niveles.

¿Cómo busco?

Seguimos mismo criterio que para insertar

¿Cómo elimino?

Eliminamos 193.

Caso 1: Tengo hijo → El abuelo se une con el nieto 

Caso 2: 2 hijos → Sustituyo por el menor de los mayores y después lo elimino.

Caso 3: No tengo hijos → Me lo cargo

BSTree <T>

boolean add (T item) // NO contains ()

T remove (T item) int ← indexOf (T)

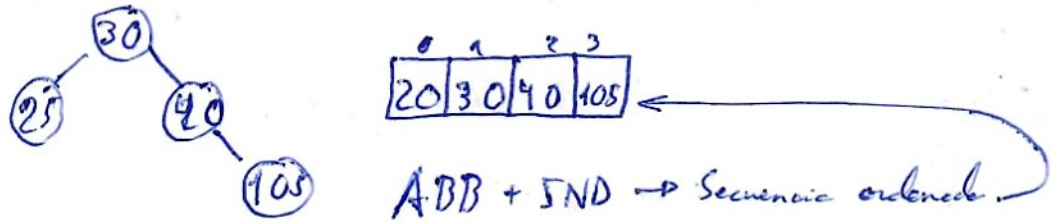
T find (T aux)

CURRENT → REFERENCIA AL OBJETO

→ CLAVE

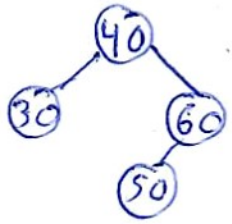
Un add degenerado se obtiene cuando el conjunto de entrada ya viene ordenado a la hora de insertar.

Ejemplo: Pasar add a Array




```
public ArrayList<T> toArray() {
  private ArrayList<T> aux = new ArrayList<>();
  toArray (this.root, aux) // Con if (this.root != null) → if (this.root != null) toArray (this.root, aux);
}

private void toArray (Node<T> current, ArrayList<T> data) {
  if (current == null) return;
  toArray (current.left, data);
  data.add (current.nodeValue);
  toArray (current.right, data);
}
```

① if (current.left != null) toArray (current.left, data);
② if (current.right != null) toArray (current.right, data);
③ }



CURRENT	DATA	LEVEL
H(40)	H3CFC	2
~~H(30)~~	H3CFC	~~3~~
H(60)	H3CFC	1
~~H(50)~~	H3CFC	~~3~~

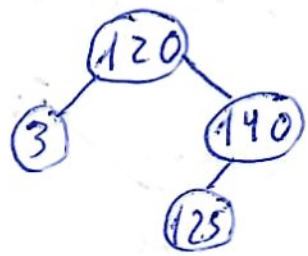
data → 30 | 40 | 50 | 60

Hacer ejemplos, pueden caer este parte.

DNI recursive

```
public String displayTree () {
  if (this.root == null) return " ";
  return displayTree (this.root, 0);
}

private String displayTree (Node<T> current, int level) {
  String result = " ";
① if (current.right != null) return result += displayTree (current.right, level (+1));
  for (int i = 0; i < level; i++) { result += "-"; }
② if (current.left != null) return result += displayTree (current.left, level (+1));
                                                  level++ → NO SE CARGA VARIABLE
  return result;
③ }
```

current	level	result	línea
# 120	0	" "	1
# 140	1	" 140 "	1
# 125	2	" 125 "	3
# 120	0	"140, 125"	2
# 3	1	" 3 "	3
# 120	0	"140, 125, 120"	3
		∅	



Colecciones de pares

Libro → { (palabra, Frecuencia) }

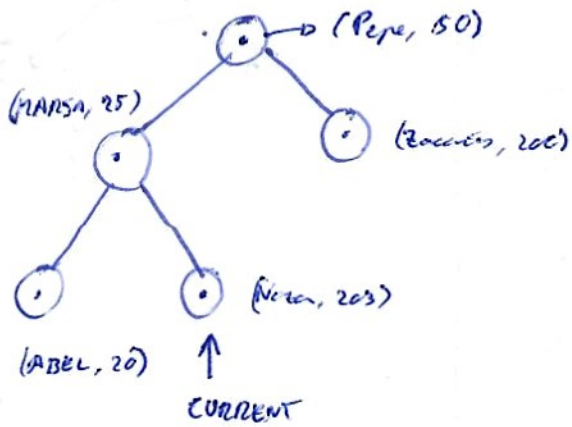
↓ ↓ ↓

Identificador Conjunto Par

AHORA UNA CLASE

```
Class Libro {
  private String libroId;
  private BSTree<Par<String, Integer>> palabrasFreq;
}

public void add (String palabra) {
  Par<String, Integer> cuent = palabrasFreq.find (new Par<>(palabra, null));
  if (cuent == null) { palabrasFreq.add (new Par<>(palabra, 1)); }
  else { cuent.setValue (cuent.getValue () + 1); }
}
```

BSTree<Par<String, Integer>> palabrasFreq;



aux

(Nazar, null)

STRING	INTEGER
ABEL	20
MARÍA	25
NAZAR	203
PEPE	50
ZAMORAS	200

ORDENADO POR PROG

BSTree<Par<Integer, ArrayList<String>>>

Análisis de ABBB ÁRBOL BINARIO DE BÚSQUEDA BÁSICO

Aleatoriedad E

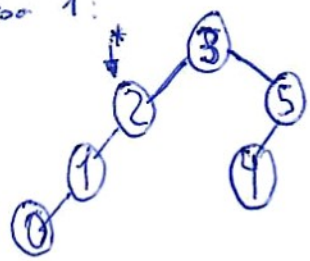
E con 5 elementos

E = {3, 5, 2, 1, 0, 4} Caso 1

E = {5, 4, 3, 2, 1, 0} Caso 2

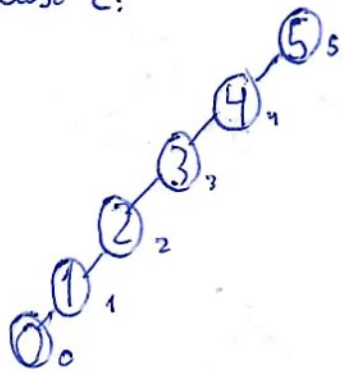
E = {0, 1, 2, 3, 4, 5} Caso 3

Caso 1:



NO, es ABBEq (2)*

Caso 2:



Caso 3:

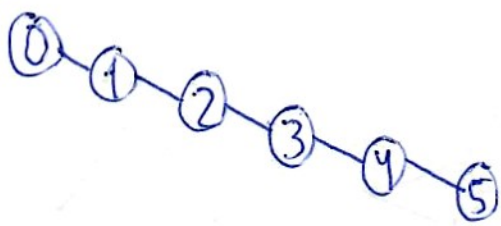


ABB Equilibrado

Factor de equilibrio (F_e)

$F_e(N) = h(N_=) - h(N_D)$

$\forall N \in T,\ F_e(N) \in \{0, -1, 1\}$ → Si lo cumple → ABBE equilibrado o bien equilibrado.

NODO

Árboles BB equilibrados

- AVL ← Extensión de ABBB
- Rojinegro (RN) → JCF → TREESET<T> / TREEMAP<K, V>
- AA
- ...

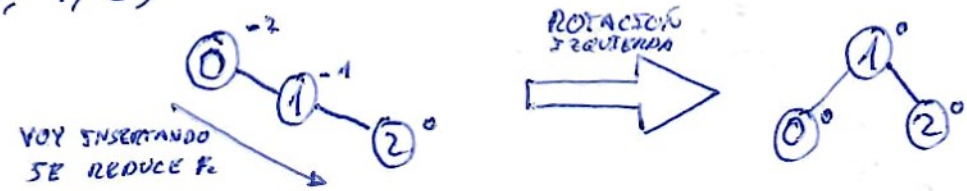
AVL

ABB que añade una operación de Equilibrado (ROTACIÓN) en Inserción y Eliminación

Rotaciones { Simples { I, D ; Dobles { ID, DI

Ejemplos:

E = {0, 1, 2}



E = {2, 1, 0}

E = {0, 2, 1}

E = {2, 0, 1}

Inserción en AVL

Voy recalculando Fe, cuando encuentre error, hago rotación (Sub 1)

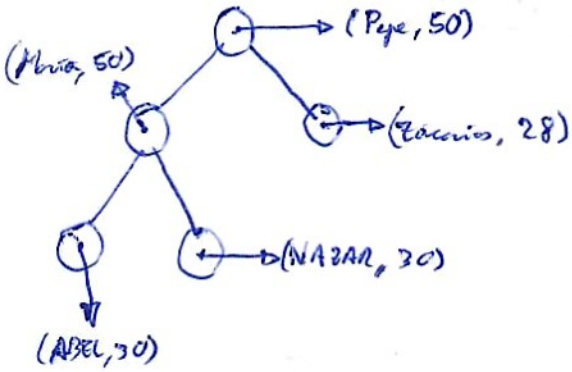
Inserto, vuelvo por el camino de inserción recalculando el Fe y hago como antes

En el código siempre es un intercambio de referencias

AVL<Par<String, Integer>> datos; → Atributo

Ordena x frecuencia

Siempre ordena según la clase



Arraylist Bidimensional

CLAVE	VALOR	
ABEL	30	Par 1
MARÍA	50	Par 2
NAZAR	30	Par 3
PEPE	50	Par 4
ZACARÍAS	28	Par 5

Ordenado por frecuencia

AVLTree<Par<Integer, Arraylist<String>>> result;

CLAVE	VALOR
28	ZACARÍAS
30	ABEL, NAZAR
50	MARÍA, PEPE

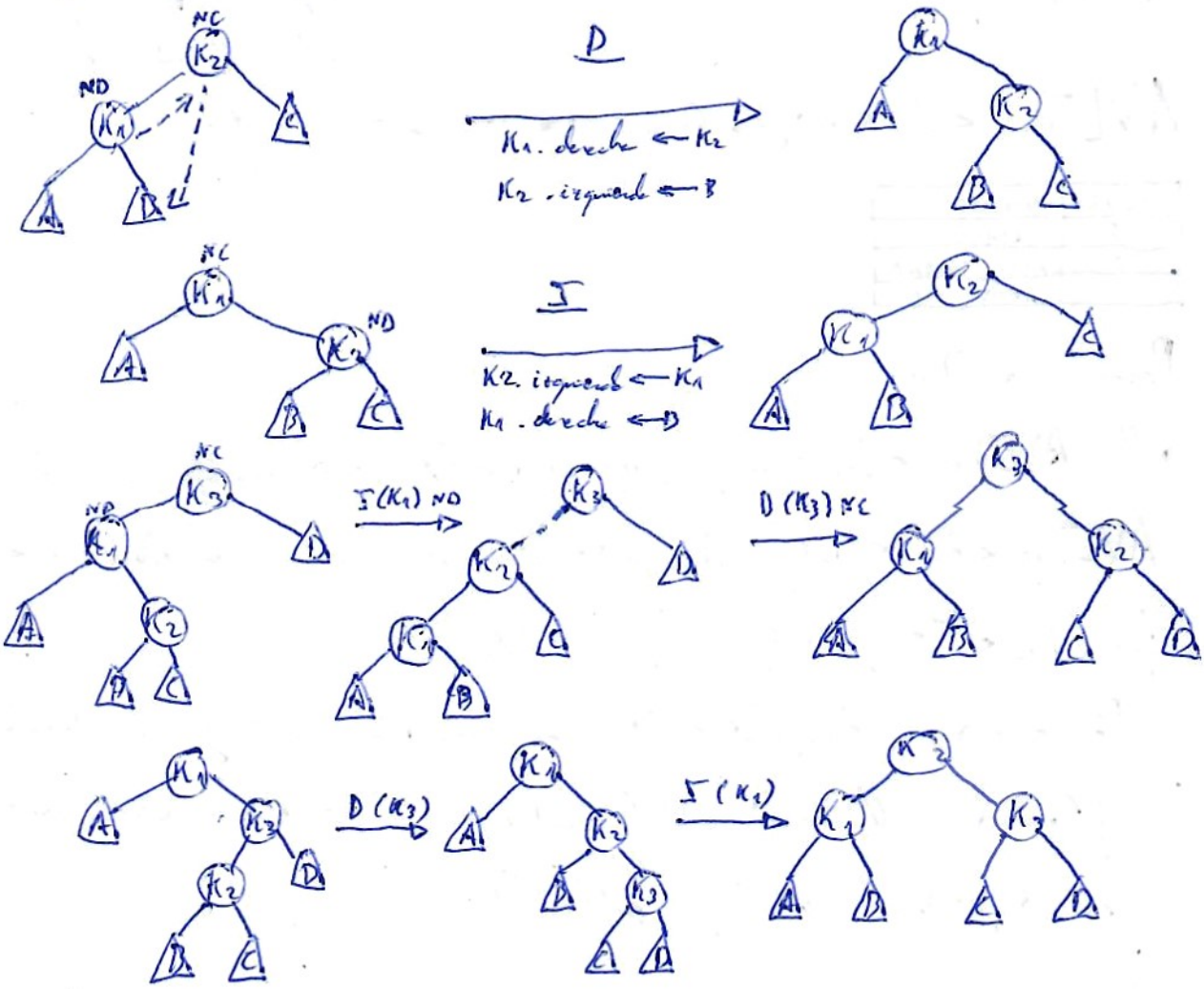
Pasamos a Java:

```
public AVLTree<Par<Integer, Arraylist<String>>> transforma (){
 AVLTree<Par<Integer, Arraylist<String>>> result = new AVLTree<>();
 for (Par<String, Integer> par, datos).{
   Par<Integer, Arraylist<String>> current = result.find(new Par<>(par.getValue(),null));
   if (current == null) result.add (current = new Par<> (Par.getValue(), new Arraylist<>());
   current.getValue().add(par.getKey());
 }
return result;
}
```

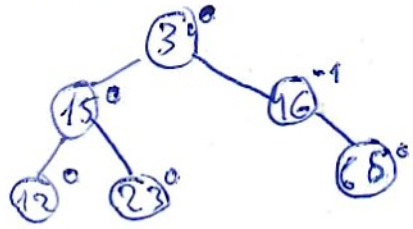
Si lo hacemos con ArrayList en vez de AVL Tree sería:

```
public ArrayList<Par<Integer, ArrayList<String>>> transform() {
  ArrayList<Par<Integer, ArrayList<String>>> result = new ArrayList<>();
  for (Par<String, Integer> par, datos) {
    int pos = result.indexOf(new Par<>(par.getValue(), null));
    if (pos == -1) { result.add(pos);}
    result.get(pos).getValue(par.getValue());
    result.add();
  }
```

Rotación (Operación atómica)



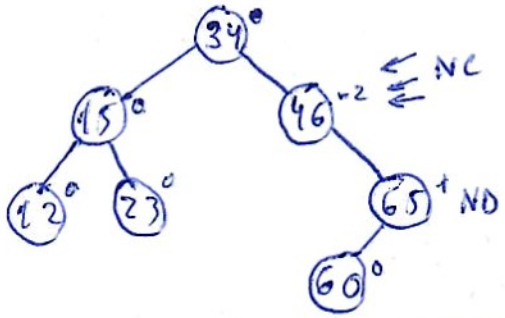
Ejemplo de Inserción: +60, +62, +66, +50

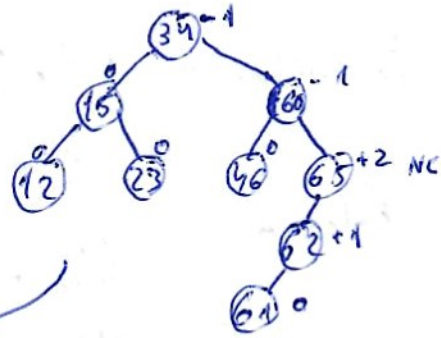


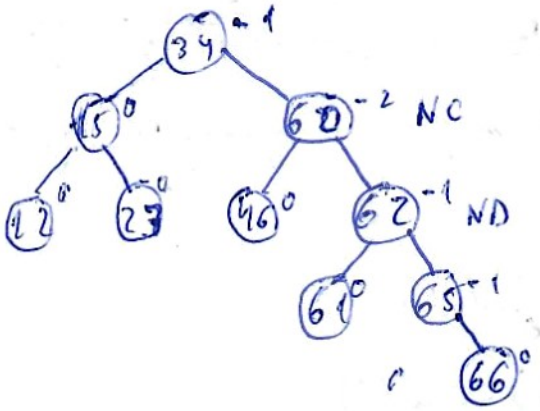
¿Es un AVL? INO → Da secuencia ordenada

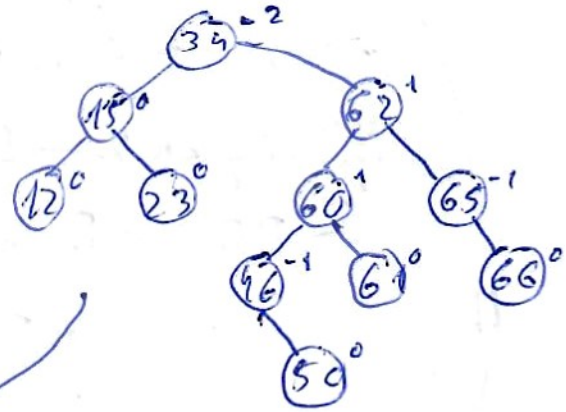
$F_e(N) = h(N_I) - h(N_D)$

Empezamos a insertar:

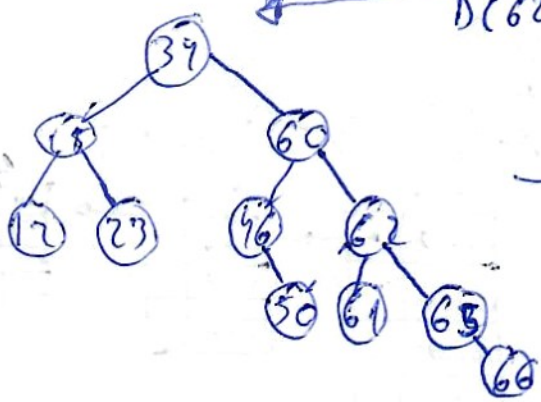


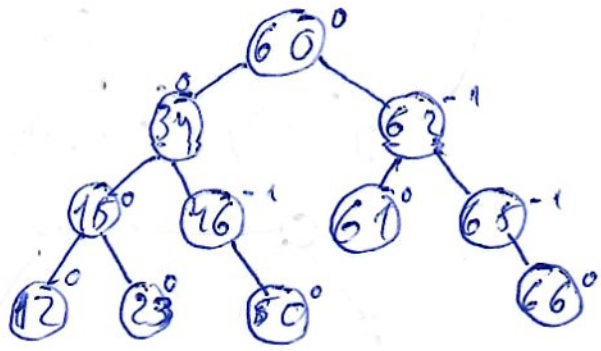






D(62)





Par <K, V>

- > facilidad
- < nº clases
- > forma más natural de representación de relaciones

1 libro → N palabras
1 libro → N freq
} Nº palabras Freq

¿Inconveniente? → Legibilidad

Par <Integer, ArrayList <String>> par = estructura.find (new Par <> (clave, null));

Colección
- 1 autor → N° libros
- 1 libro → M palabras (M freq)

AVLTree <Par <String autor, AVLTree <Par <String libro, AVLTree <Par <String palabra, Integer>>>>>>

datos; → Representación compacta pero (ineficiente) → MyTreeMap<K,V> → MyEntry → K-V

MyTreeMap <K, V> → Encapsula árbol de pares e implementa conjunto de pares K V

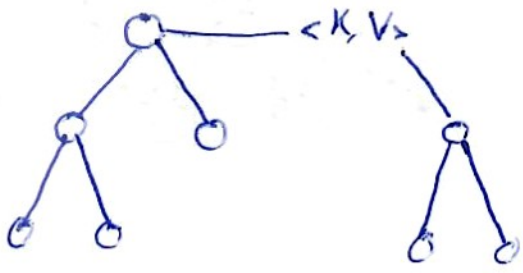
- BSTree <Par, <K, V>>

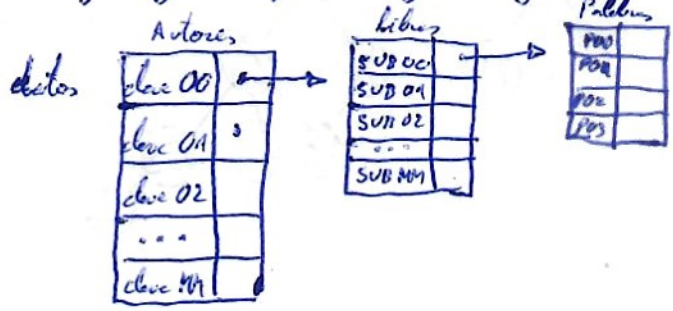
V ← get (K) = find ()

Vant ← put (K, V) = add ()

Boolean ← containsKey ()

MyTreeMap <String autor, MyTreeMap <String libro, MyTreeMap <String palabra, Integer>>> datos;



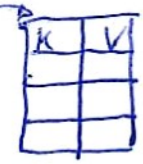


¿Problema de AVLTree? → Recursión (No válido para grandes volúmenes de datos)

RojiNegro ← ABB Equilibrado Estática

TreeMap<K, V>

├ Par<K, V> → Entry<K, V>



getKey()

getValue()

setValue()

V ← get(K)

Void ← put(K, V)

Boolean ← containsKey()

{Ki} ← KeySet()

{(K, V)} ← entrySet()

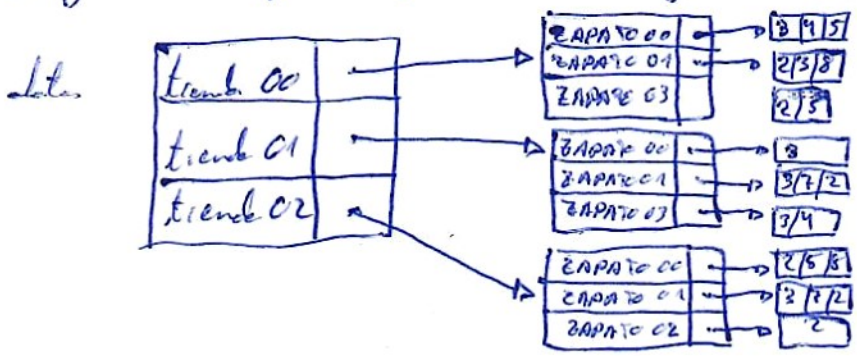
[Vi] ← Values()

TreeMap<String, TreeMap<String, TreeSet<Integer>>> dates;

① for (String friendId : dates.keySet()) {...}

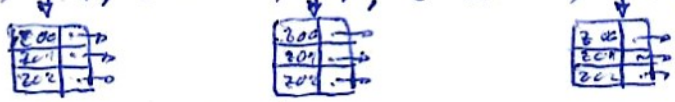
② for (Entry<String, TreeMap<String, TreeSet<Integer>>> par : dates.entrySet()){...}

③ for (TreeMap<String, TreeSet<Integer>> mapa : dates.values() {...}

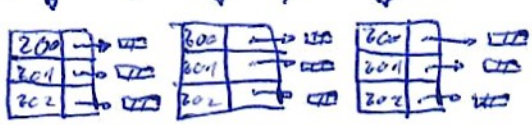


dates.KeySet() → {friend 00, friend 01, friend 02}

dates.entrySet() → {(friend 00, ↓), (friend 01, ↓), (friend 02, ↓)}



dates.values() → { ↓ , ↓ , ↓ }



TreeMap<String, TreeMap<String, TreeSet<Integer>>> datos = new TreeMap<>();

TiendaId ModeloId Tallas

Comparator.reverseOrder()

Si lo quiere al revés

datos → Referencia a un mapa

Método add

```
public void add (String tiendaId, String modeloId, Integer talles) {
  TreeMap<String, TreeSet<Integer>> modelosCun = this.datos.get(tiendaId);
  if (modelosCun == null) {
    this.datos.put(tiendaId, modelosCun = new TreeMap<>());
  }
```

⚠ Los mapas no tienen add, tienen put

```
  TreeSet<Integer> tallesModelo = modelosCun.get(modeloId);
  if (tallesModelo == null) {
    modelosCun.put(modeloId, tallesModelo = new TreeSet<>());
  }
  for (Integer talla : talles) {
    tallesModelo.add(talla);
  }
}
```

Método getTienda

```
public TreeSet<String> getTienda(int tabb) {
  TreeSet<String> result = new TreeSet<>();
  for (Entry<String, TreeMap<String, TreeSet<Integer>>> parCun : this.datos.EntrySet()){
    for (TreeSet<Integer> tablesCun : parCun.getValue.Values()) {
      if (!tablesCun.contains(tabb)) continue;
      result.add(parCun.getKey());
      break;
    }
  }
  return result;
}
```

Método getModelo

```
public TreeSet<String> getModelo(int tabb){
 TreeSet<String> result = new TreeSet<>();
 for (TreeMap<String, TreeSet<Integer>> submapa: this.datos.values()) {
   for (Entry<String, TreeSet<Integer>> parCun: submapa.entrySet() {
     if (!parCun.getValue().contains(tabb)) continue;
     result.add(parCun.getKey());
   }
 }
 return result;
}
```

Método getModelos

```
public TreeSet<String> getModelos(String tiendaId) {
  TreeMap<String, TreeSet<Integer>> modelosCun = this.datos.get(tiendaId);
  return modelosCun == null ? null : new TreeSet<>(modelosCun.keySet());
}
```

Clase Prueba

```
private AVLTree<Par<String, ArrayList<Integer>>> estructuraPrincipal;
private AVLTree<Par<String, ArrayList<String>>> estructuraSecundaria;
public void dataDump() {
  ... estructuraPrincipal -> estructuraSecundaria
}
public ArrayList<Par<String, Integer>> transform() {
  // estructuraPrincipal
}
```

TreeMap<K, V> -> new()
TreeSet<T> -> new()
-> comparator

TreeSet<~~ArrayList<Integer>~~> datos = new TreeSet<>(Comparator NPS)
-> NO es posible

TreeSet<T> -> Colección (ArrayList<Integer>)

```
public class Grupo implements Comparable<Grupo> {
  private ArrayList<Integer> datos;
  ;
  compare(d);
}
```

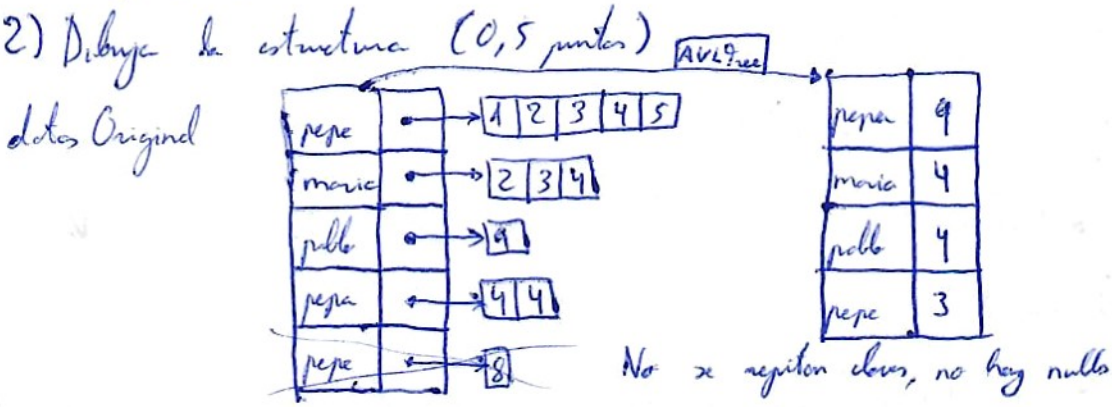
¿Cuanto tiempo tarde en implementar el método transform()?

```
public class prueba {

  private ArrayList<Par<String, TreeSet<Integer>>> datosOriginal;

  private AVLTree<Par<String, Double>> datosDispersivo;
```

1) Comentar la implementación (0,5 puntos)

En lenguaje formal, sin código. Mostrar la idea de lo que queremos hacer.

2) Dibuja la estructura (0,5 puntos)



No se repiten claves, no hay nulls

3) Rellena el método (1 punto) → Resta eficiencia y cosas de más

```
  for (Par<String, TreeSet<Integer>> par : datosOriginal) {
    int suma = 0;
    for (Integer valor : par.getValue()){
      suma += valor;
    }
    datosDispersivo.add(new Par<>(par.getKey(), suma/par.getValue().size()));
}
```

4) Suponemos que se repite la clase Pepe

```
for (Par<String, TreeSet<Integer>> par : datosOriginal) {
  int suma = 0;
  for (Integer valor : par.getValue()) {
    suma += valor;
  }
  Par<String, Double> parAcum = datosDispersion.find(new Par<>(par.getKey(), null));
  if (parAcum == null) {
    datosDispersion.add(new Par<>(par.getKey(), suma / par.getValue().size()));
  } else {
    parAcum.setValue(parAcum.getValue() + suma / par.getValue().size() / 2.0);
  }
}
```