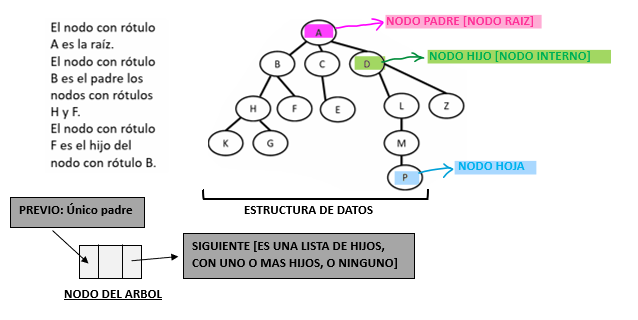


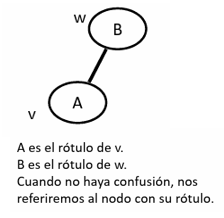
RESUMEN DE ARBOLES GENERALES Y ARBOLES BINARIOS:

**//Las estructuras de arboles, son estructuras jerarquicas**

**TDA ARBOL GENERAL:**

* Un árbol es un TDA que almacena los elementos llamados nodos jerárquicamente
* Con la excepción del nodo raíz, cada nodo en un árbol tiene un nodo padre y cero o más nodos hijos [La raiz se dibuja arriba y es siempre el primer nodo de cualquier recorrido]
* Cada nodo conoce su rotulo [Elemento], a su padre y su lista de nodos hijos

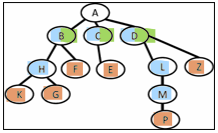




* **DEFINICION FORMAL DE ARBOL:**

Un árbol T se define como un conjunto de nodos almacenando elementos tales que los nodos tienen una relación padre-hijo, que satisface:

1. Si T es no vacío, tiene un nodo especial, llamado la raíz de T, que no tiene padre
2. Cada nodo v de T diferente de la raíz tiene un único nodo padre w
3. Cada nodo v con padre w es un hijo de w



* **RELACIONES ENTRE NODOS:**
* **NODOS HERMANOS:** Dos nodos con el mismo padre se llaman Hermanos

**EJEMPLO:** B y D son hermanos, L y Z lo son, H y E no son hermanos (son primos)

* **NODOS EXTERNOS U HOJAS:** Un nodo v es externo si no tiene hijos

**EJEMPLO:** F, K, G, E, P y Z son hojas

* **FRONTERA:** Conjunto de nodos hoja de un arbol
* **NODO INTERNO:** Un nodo v es interno si tiene uno o más hijos

**RELACIONES CON EL NODO “A”:**

1. Es ancestro propio de sus hijos B,C,D
2. Es ancestro propio de sus nietos H,F,E,L,Z
3. Es ancestro propio de su tataranieto P
4. Es ancestro propio de sus bisnietos K,G,M
5. No es ancestro propio de sí mismo

**RELACIONES CON EL NODO “B”:**

1. Es ancestro de sí mismo, pero no su ancestro propio
2. Es ancestro de sus hijos H y F
3. Es ancestro de sus nietos K y G

**EJEMPLO:** A, B, C, D, H, L y M son nodos internos

* **ANCESTRO(U,V):** Un nodo u es ancestro de un nodo v si u=v o u es un ancestro del padre de v

**EJEMPLO:** ancestro(C,C), ancestro(A,E), ancestro(D,P)

* **ANCESTRO PROPIO (U,V):** u es ancestro propio de v si u es ancestro de v y u ≠ v

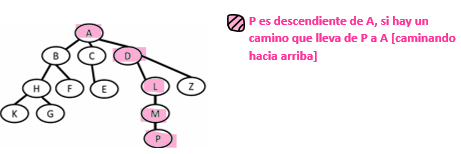
**EJEMPLO:** acentropropio(A,E), ancestropropio(D, P), ancestropropio(D,L)

**EXPLICACION: Todo nodo es ancestro de si mismo y de los que están mas abajo y ancestro propio solo de los que están abajo de si mismo**

* **DESCENDIENTE(U,V):** Un nodo u es descendiente de un nodo v si v es un ancestro de u

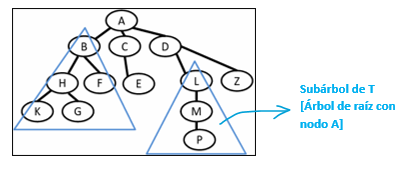
**EJEMPLO:** descendiente(C,C), descendiente(E,A), descendiente(P,D)

* **DESCENDIENTE PROPIO (U,V):** u es descendiente propio de v si u es descendiente de v y u ≠ v

**EJEMPLO:** descendientepropio(E,A)

* **SUBARBOL:** El subárbol de T con raíz en el nodo v es el árbol consistiendo de todos los descendientes de v

**EJEMPLO:** Mostramos dos subárboles con raíces B y L respectivamente



* **ARCOS Y CAMINOS EN ARBOLES:**
* **ARCO DIRIGIDO:** Un arco de un árbol T es un par de nodos (u,v) tales que u es el padre de v

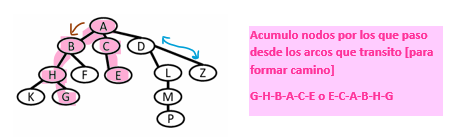
**EJEMPLO:** (D,L) es un arco y (D,Z) es otro arco

* **ARCO NO DIRIGIDO:** Un arco es un par de nodos (u,v) o (v,u) tales que u es padre de v [En ambas direcciones]

**EJEMPLO:** (D,Z) y (Z,D) son arcos no dirigidos

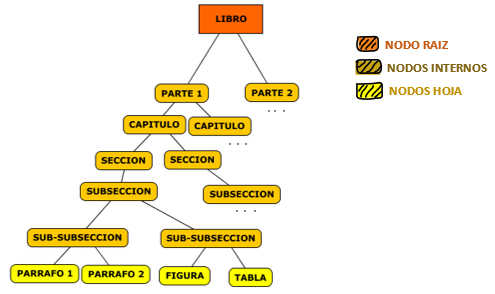
* **CAMINO:** Un camino de T es una secuencia de nodos tales que cualquier par de nodos consecutivos en la secuencia forman un arco

**EJEMPLOS:**

1. A, B, F es un camino
2. F, B, A, D, L, M es otro camino

**EJEMPLO GENERAL:**

* La relación de herencia de clases en Java forma un árbol
* La raíz, java.lang.Object es un ancestro de todas las clases
* Cada clase C es un descendiente de Object y C es la raíz del subárbol formado por todas las clases descendientes de C
* **ARBOLES ORDENADOS:**
* **Un árbol se dice ordenado si existe un orden lineal para los hijos de cada nodo 🡪** Es decir, se puede identificar el primer hijo, el segundo hijo y así sucesivamente
* Tal orden se visualiza de izquierda a derecha de acuerdo a tal ordenamiento

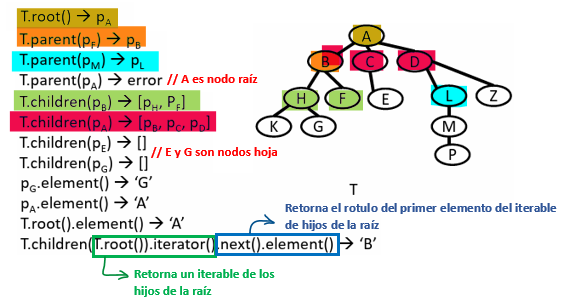
**EJEMPLO:** Las componentes de un libro:

* **TDA ARBOL GENERAL:**

El TDAArbol se modelara por medio de una interfaz que se encuentra modelada por las siguientes operaciones y sus respectivas restricciones [contrato]:

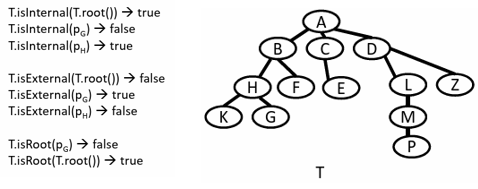
* **METODOS DE ACCESO [Reciben y retornan posiciones]:**

1. **Root()** 🡪 Retorna la raíz del árbol, error si el árbol está vacío **[EmptyTreeException]**
2. **Parent(v)** 🡪 Retorna el padre de v, error si v es la raíz **[InvalidPositionException, BoundaryViolationException]**
3. **Children(v)** 🡪 Retorna una colección iterable conteniendo los hijos del nodo v **[InvalidPositionException]** [Si el árbol es ordenado, children los mantiene en orden; Si v es una hoja children(v) está vacía]



* **METODOS DE CONSULTA:**

1. **isInternal(v)** 🡪 Testea si v es un nodo interno **[InvalidPositionException]**
2. **isExternal(v)** 🡪 Testea si v es una hoja **[InvalidPositionException]**
3. **isRoot(v)** 🡪 Testea si v es la raiz **[InvalidPositionException]**



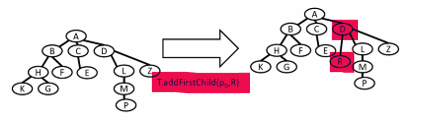
* **METODOS GENERICOS: E**

1. **Size()** 🡪 Retorna el número de nodos del árbol
2. **isEmpty()** 🡪 Testea si el árbol tiene o no nodos
3. **Iterator()** 🡪 Retorna un iterador con los elementos ubicados en los nodos del árbol
4. **Positions()** 🡪 Retorna una colección iterable de los nodos del árbol
5. **Replace(v,e)** 🡪 Reemplaza con e y retorna el elemento ubicado en v **[InvalidPositionException]**

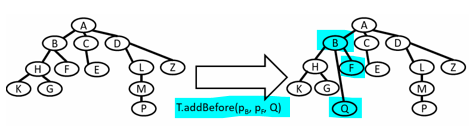
* **METODOS DE MODIFICACION:**

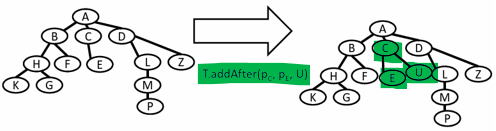
1. **createRoot(e)** 🡪 crea un nodo raíz con rótulo e **[InvalidOperationException: si ya existe una raiz]**
2. **addFirstChild(p, e)** 🡪 Agrega un primer hijo al nodo p con rótulo e **[InvalidPositionException]**
3. **addLastChild(p, e)** 🡪 Agrega un último hijo al nodo p con rótulo e **[InvalidPositionException]**
4. **addBefore(p, rb, e)** 🡪 Agrega un nodo con rótulo e como hijo de un nodo padre p dado 🡪 El nuevo nodo se agregará delante de otro nodo hermano rb también dado **[InvalidPositionException]**
5. **addAfter(p, lb, e)** 🡪 Agrega un nodo con rótulo e como hijo de un nodo padre p dado 🡪 El nuevo nodo se agregará detrás de otro hermano lb también dado **[InvalidPositionException]**
6. **removeExternalNode (p)** 🡪 Elimina la hoja p **[InvalidPositionException]**
7. **removeInternalNode (p)** 🡪 Elimina el nodo interno p 🡪 Los hijos del nodo eliminado lo reemplazan en el mismo orden en el que aparecen🡪 La raíz se puede eliminar si tiene un único hijo **[InvalidPositionException]**
8. **removeNode (p)** 🡪 Elimina el nodo p **[InvalidPositionException]**

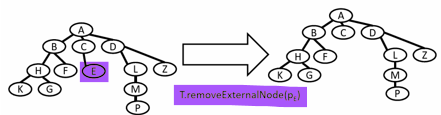
**EJEMPLOS GRAFICOS DE LOS METODOS:**











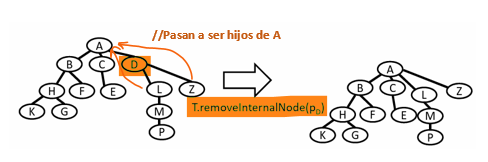
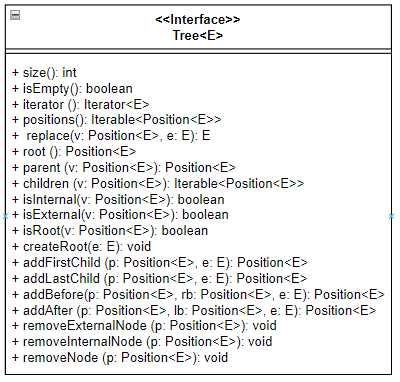
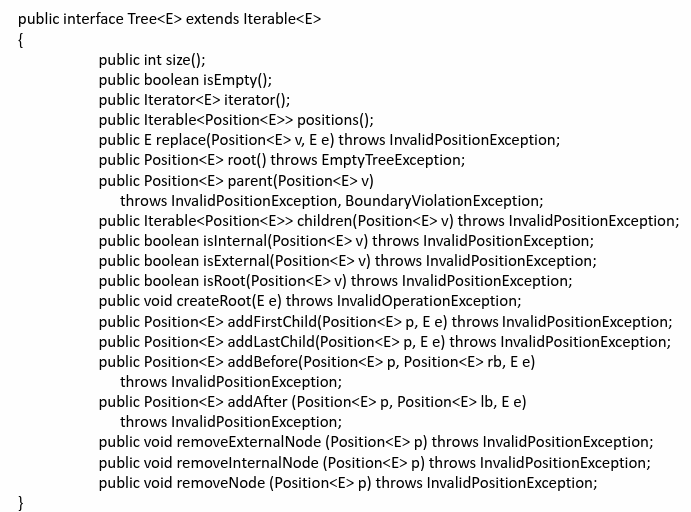
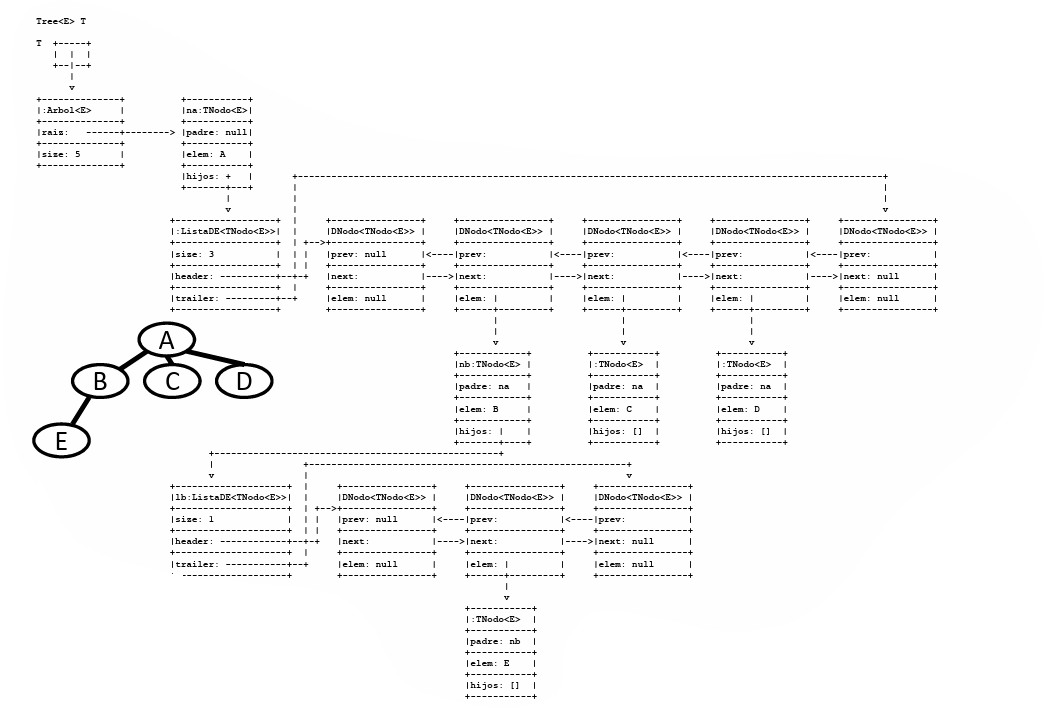
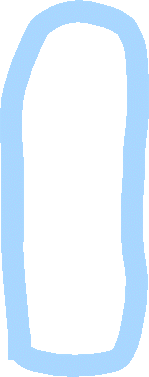
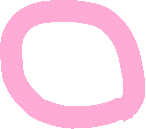


DIAGRAMA UML DE LA INTERFAZ TREE<E>

INTERFAZ TREE<E> EN JAVA:



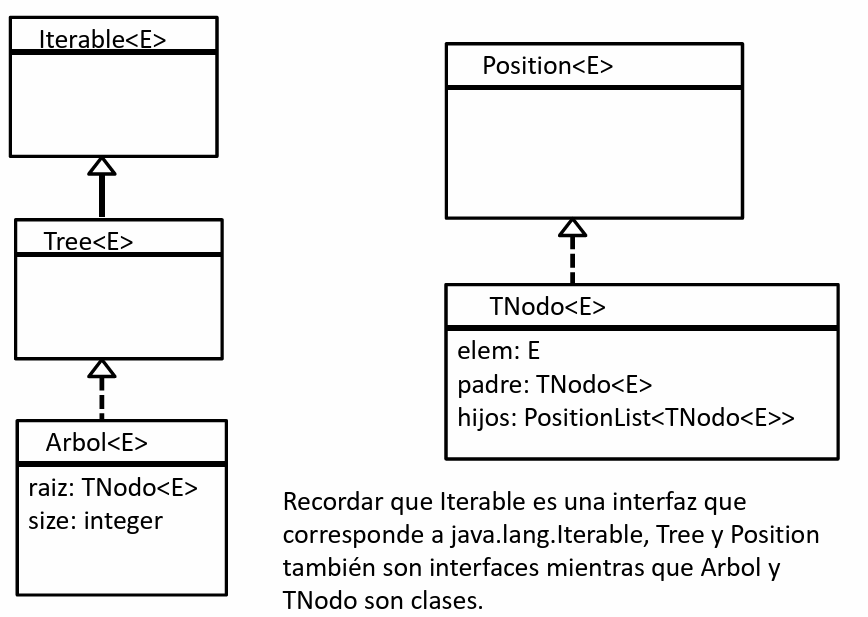
**REPRESENTACION GRAFICA DE LA IMPLEMENTACION:**



**Dnodo<E> de lista doblemente enlazada que contienen elementos del tipo TNodo<E> que representan los hijos de un nodo en la lista doblemente enlazada [Esto porque son las posiciones de los árboles]**



* **Lista de hijos + padre**
* **El árbol conoce la raíz**
* **Cada nodo conoce el rotulo(elemento), la lista de nodos hijos y su padre**



**ACLARACIONES:**

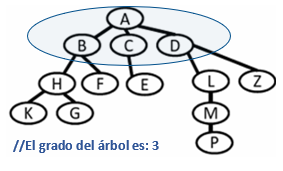
* MODELO: Árbol con nodos rotulados y ordenado de izquierda a derecha
* Las operaciones de TDAArbol se definen en función de posiciones
* No se definen en función de sus nodos para:

1. No exponer la implementación
2. No permitir cambios en la estructura [Solo se puede modificar el estado interno a través de las operaciones del TDA]

REPRESENTACION DE ARBOLES EN DIAGRAMA UML REDUCIDO

|  |  |
| --- | --- |
| **METODO** | **ORDEN DEL TIEMPO DE EJECUCION** |
| **size()** | O(1) |
| **isEmpty()** | O(1) |
| **iterator ()** | O(2n) |
| **positions()** | O(n) |
| **replace(v,e)** | O(1) |
| **root ()** | O(1) |
| **parent(v)** | O(1) |
| **children(v)** | O(g) [g🡪 Grado del arbol] |
| **isInternal(v)** | O(1) |
| **isExternal(v)** | O(1) |
| **isRoot(v)** | O(1) |
| **createRoot(e)** | O(1) |
| **addFirstChild(p,e)** | O(1) |
| **addLastChild(p,e)** | O(1) |
| **addBefore(p,rb,e)** | O(g) [g🡪 Grado del arbol] |
| **addAfter(p,lb,e)** | O(g) [g🡪 Grado del arbol] |
| **removeExternalNode(p)** | O(g) [g 🡪 Grado del arbol] |
| **removeInternalNode(p)** | O(g) [g 🡪 Grado del arbol] |
| **removeNode(p)** | O(g) [g 🡪 Grado del arbol] |
| **checkPosition(v)** | O(1) |

**//PEOR CASO: El arbol es una listA 🡪 Por lo tanto la altura de un arbol depende de su cantidad de nodos, osea “n”**

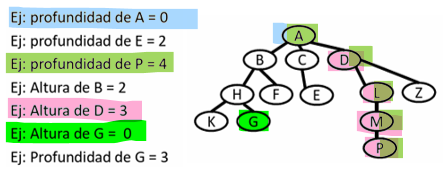


**GRADO DEL ARBOL: Cantidad de hijos del nodo con mayor cantidad de hijos**

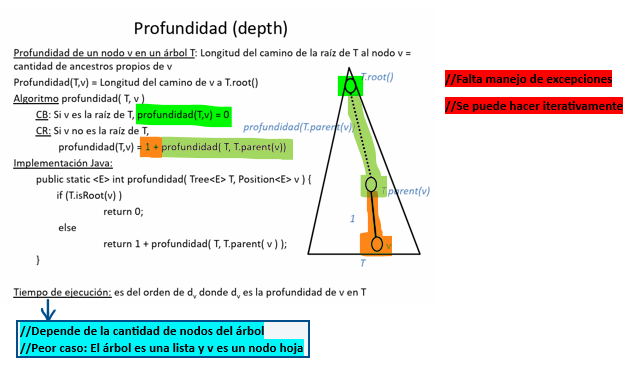


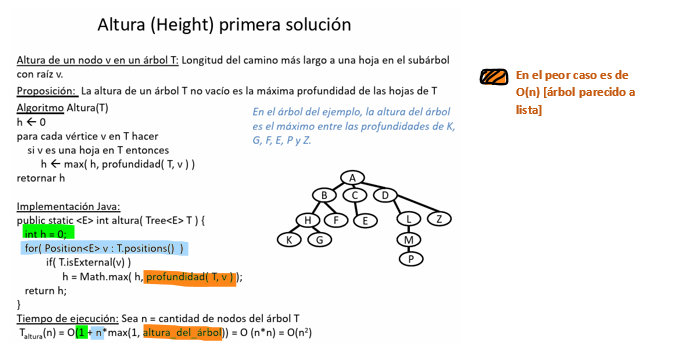
**//Ver implementaciones de Arbol General [Arbol<E>, TNodo<E>, Position<E>, y demas clases relacionada**

* **PROFUNDIDAD Y ALTURA EN ARBOLES:**
* **PROFUNDIDAD DE UN NODO V EN UN ARBOL T:** Longitud del camino de la raíz de T al nodo v = cantidad de ancestros propios de v
* **LONGITUD DE UN CAMINO:** Cantidad de arcos del camino
* **ALTURA DE UN NODO V EN UN ARBOL T:** Longitud del camino más largo a una hoja en el subárbol con raíz v
* **ALTURA DE UN ARBOL T:** Altura del nodo raíz de T [Distancia (+) larga, en cantidad de arcos; hoja mas lejana]

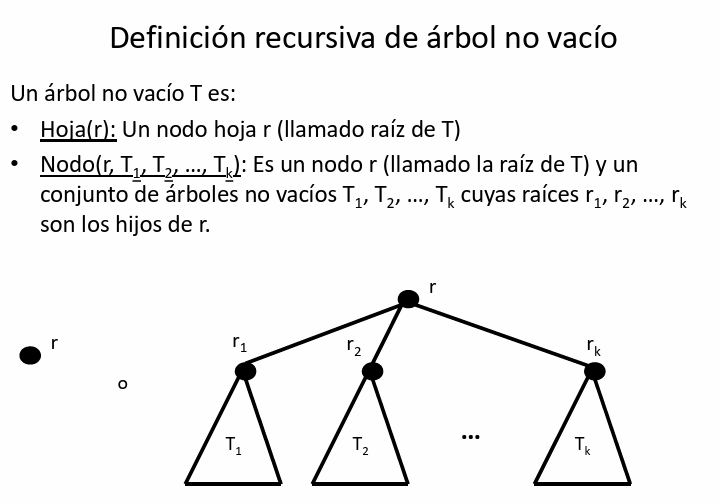


ALGORITMO RECURSIVO DE CALCULO DE PROFUNDIDAD DE UN NODO:





ALGORITMO ITERATIVA DE CALCULO DE ALTURA DE UN NODO (1):

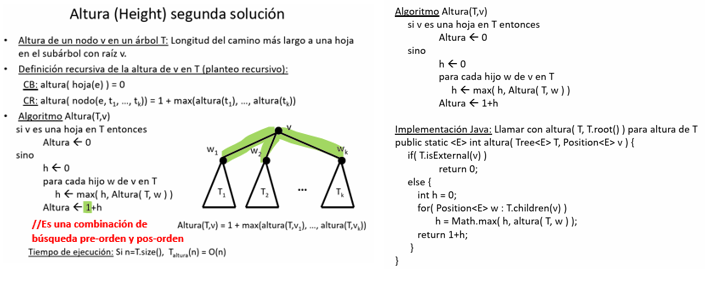


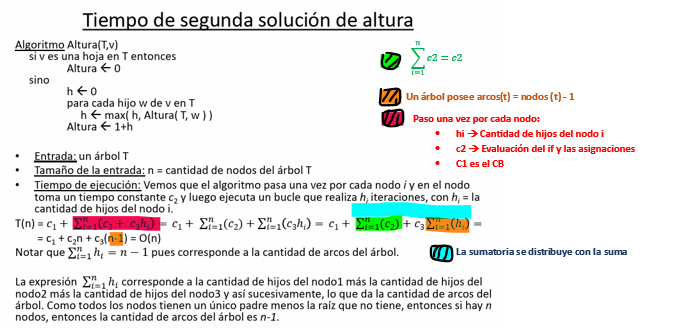


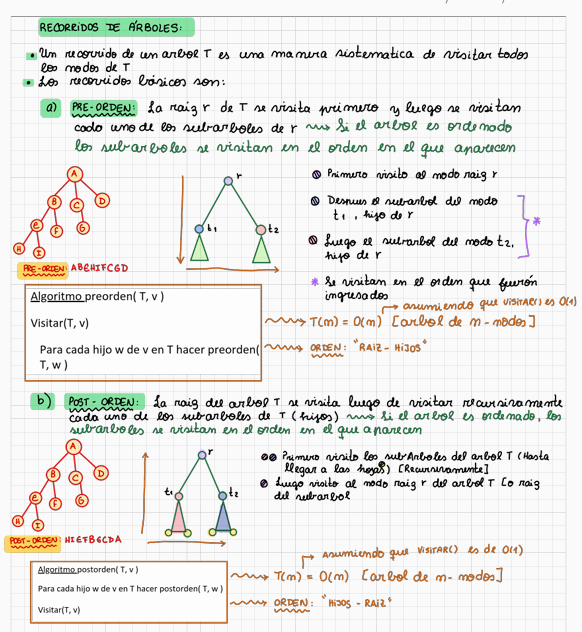
**ALGORITMOS RECURSIVOS:**

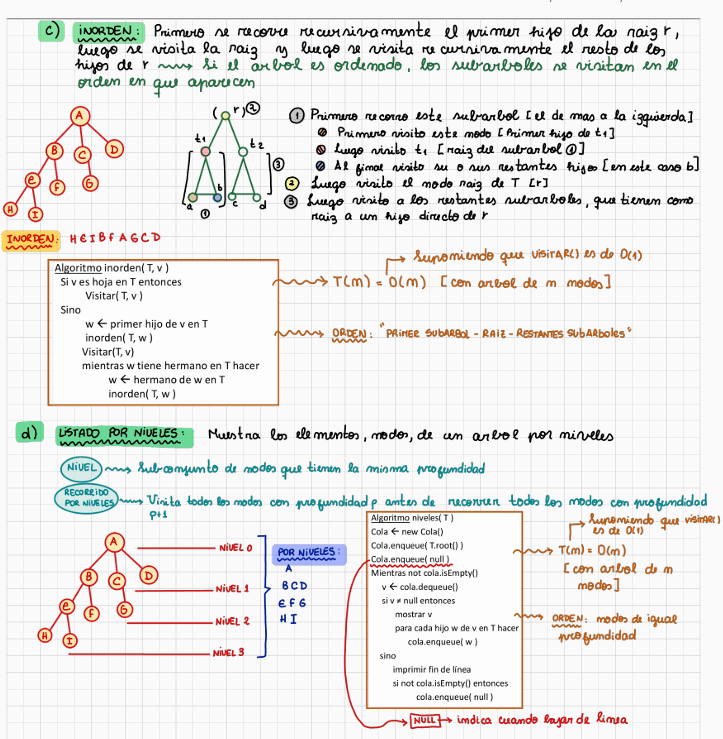
1. Plantear CB y CR
2. El CR debe reducir el tamaño del problema (n) 🡪 En este caso la cantidad de nodos
3. HI 🡪 Supongo que el método del CR funciona bien 🡪 En este caso Altura(T,w)

ALGORITMO RECURSIVA DE CALCULO DE ALTURA DE UN NODO (2):

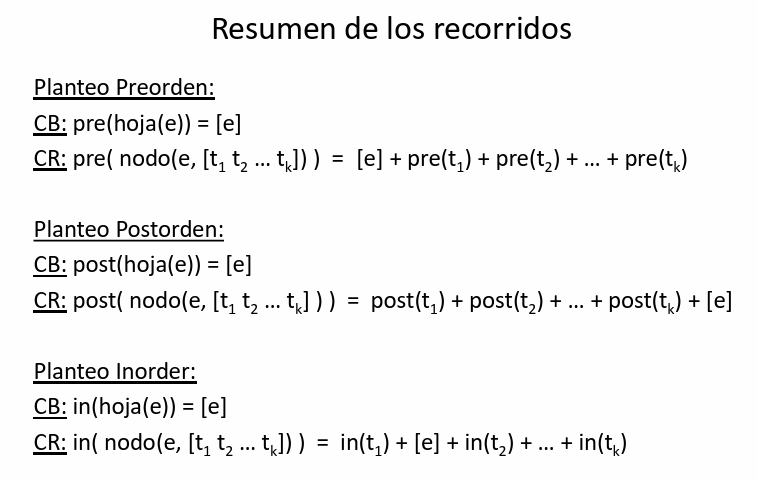




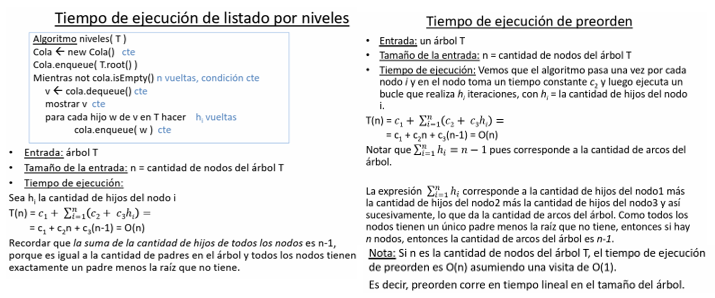
* **RECORRIDOS DE ARBOLES:**



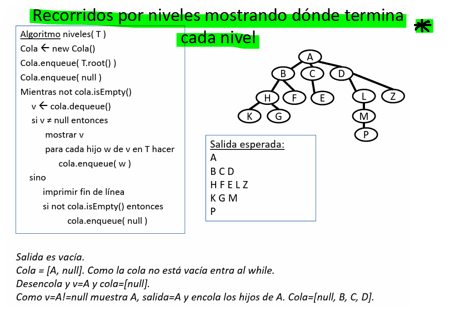








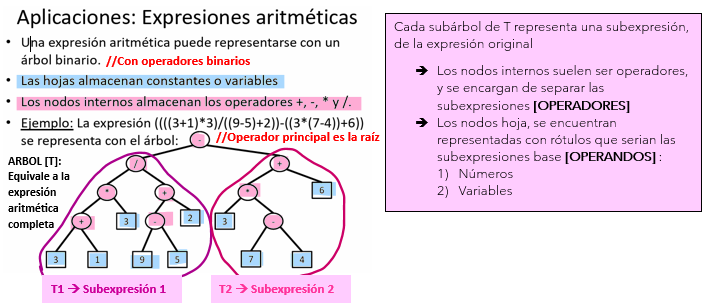




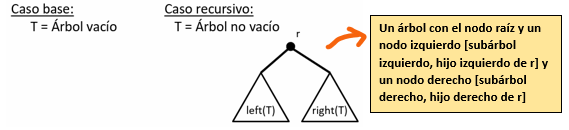
**TDA ARBOL BINARIO:**

* **DEFINICION:** **Un arbol binario es un árbol ordenado que cumple:**

1. **Cada nodo tiene a lo sumo dos hijos**
2. **Cada nodo hijo es o bien hijo izquierdo o hijo derecho**
3. **El hijo izquierdo precede al hijo derecho en el orden de los hijos de un nodo**

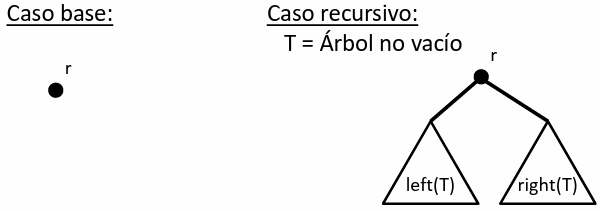
* El subárbol que tiene como raíz al hijo izquierdo se llama **SUBARBOL IZQUIERDO**
* El subárbol que tiene como raíz al hijo derecho se llama **SUBARBOL DERECHO**
* **ARBOL PROPIO** 🡪 En un árbol binario propio, cada nodo tiene 0 o dos hijos [FULL BT]
* Si un árbol binario no es propio, entonces es **ARBOL IMPROPIO**
* **DEFINICION RECURSIVA DE ARBOL BINARIO [CON MODELADO DE ARBOL VACIO]:** Un árbol binario T es o bien vacío, o consiste de:

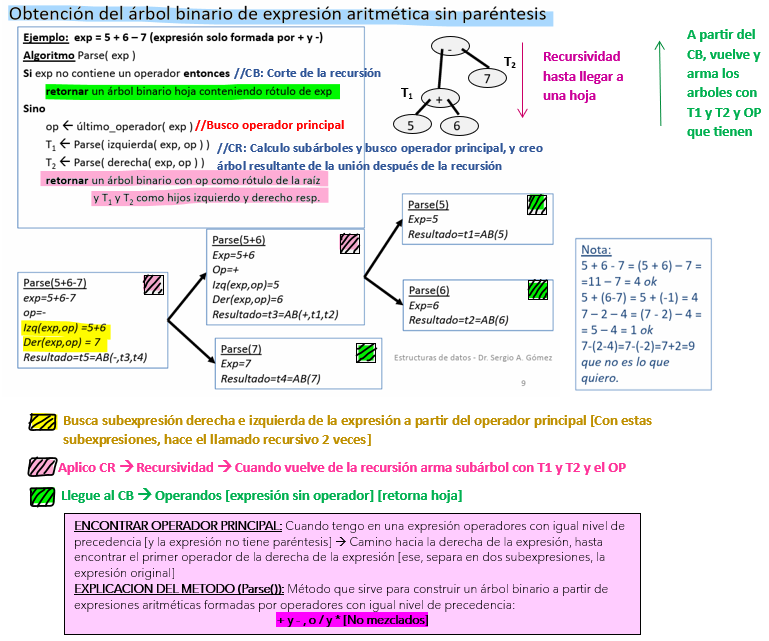
1. Un nodo r, llamado la raíz de T, que contiene un rótulo (o elemento)
2. Un árbol binario, llamado el subárbol izquierdo de T
3. Un árbol binario, llamado el subárbol derecho de T

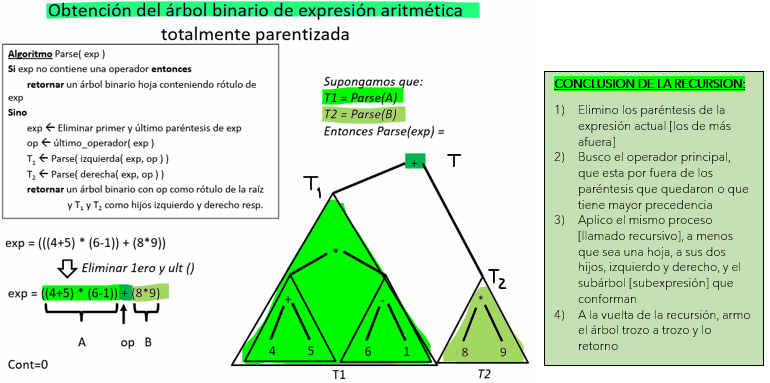


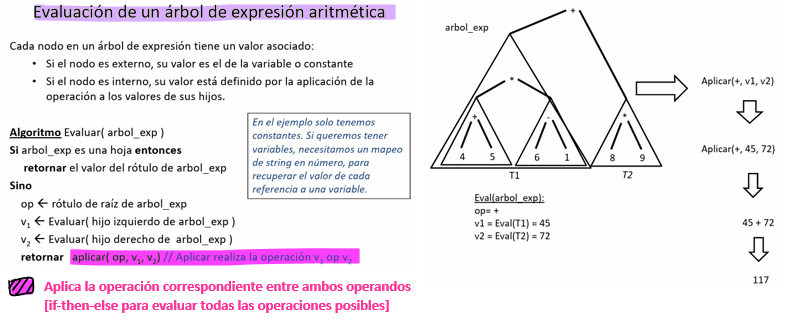
* **DEFINICION RECURSIVA DE ARBOL BINARIO [NO VACIO, NO MODELA ARBOL AVACIO]:** Un árbol binario T es :

1. Hoja(k): Un nodo r, llamado la raíz de T, que contiene un rótulo (o elemento k)
2. Nodo( k, TI, TD ): k es el rótulo y TI y TD son árboles binarios llamados hijo izquierdo y derecho, respectivamente









* **DIFERENCIA ENTRE ARBOL BINARIO Y ARBOL GENERAL:**

La diferencia entre un árbol binario y un árbol general, radica en la forma en que los nodos se ramifican y en la cantidad de hijos que puede tener cada nodo → Cada nodo en un árbol binario puede tener como máximo 2 hijos (hizo izquierdo y derecho) y un solo padre, mientras que en los árboles generales la cantidad de hijos por nodo es ilimitada

* **TDA ARBOL BINARIO:**

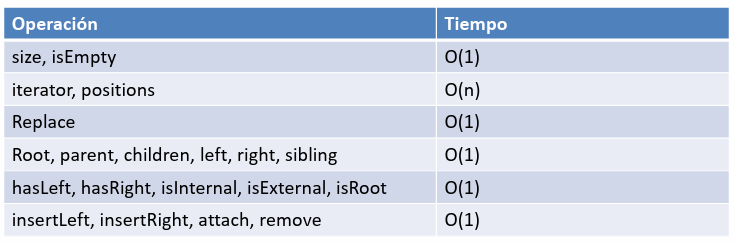
El TDAArbolBinario es una especializacion de Tree<E> que ademas soporta los siguientes metodos adicionales:

* **METODOS DE ACCESO:**

1. **left(v)** 🡪 Retorna el hijo izquierdo de v, ocurre error si v no tiene hijo Izquierdo **[InvalidPositionException, BoundaryViolationException]**
2. **right(v)** 🡪 Retorna el hijo derecho de v, ocurre error si v no tiene hijo derecho**[InvalidPositionException, BoundaryViolationException]**
3. **hasLeft(v)** 🡪 Testea si v tiene hijo Izquierdo **[InvalidPositionException]**
4. **hasRight(v)** 🡪 Testea si v tiene hijo derecho **[InvalidPositionException]**

* **METODOS DE MODIFICACION:**

1. **addRoot(e) (o createRoot(e))** 🡪 Agrega un nodo raíz con rótulo e, error si ya hay raíz **[InvalidOperationException]**
2. **insertLeft(v, e)** 🡪 Crea un nodo w hijo izquierdo de v con rótulo e, error si v ya tiene hijo Izquierdo **[InvalidOperationException,InvalidPositionException]**
3. **insertRight(v,e)** 🡪 Crea un nodo w hijo derecho de v con rótulo e, error si v ya tiene hijo derecho **[InvalidOperationException,InvalidPositionException]**
4. **remove(v)** 🡪 Elimina el nodo v (si v tiene un hijo, reemplaza a v por su hijo, si v tiene dos hijos entonces error) **[InvalidOperationException, InvalidPositionException]**
5. **attach( v, T1, T2 )** 🡪 Setea T1 como hijo izq de v y T2 como hijo derecho de v (error si v no es hoja) **[InvalidPositionException]**
6. **sibling(p)** 🡪 Retorna el hermano de la posicion p pasada por parametro, sino tiene hermano, retorna nulo **[InvalidOperationException, InvalidPositionException]**

****

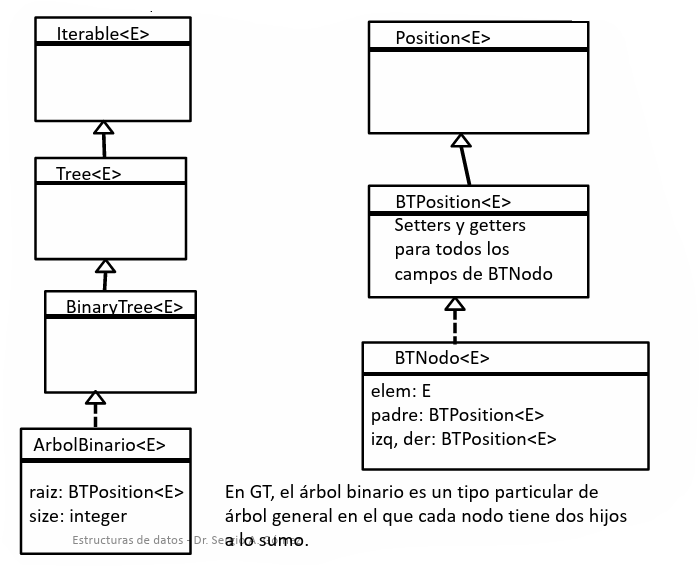
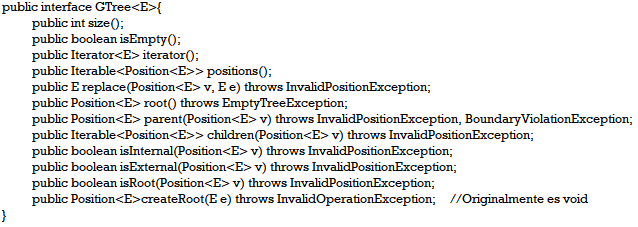
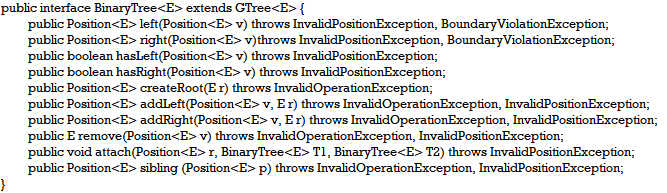


DIAGRAMA UML REDUCIDO DE LA IMPLEMENTACION DEL ARBOL BINARIO

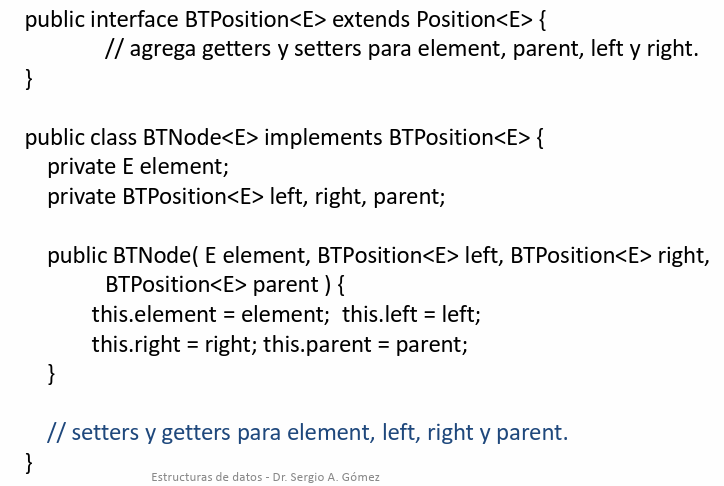


Interfaz GTree<E> EN JAVA





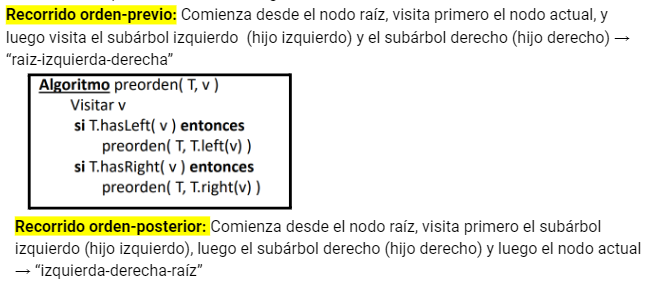
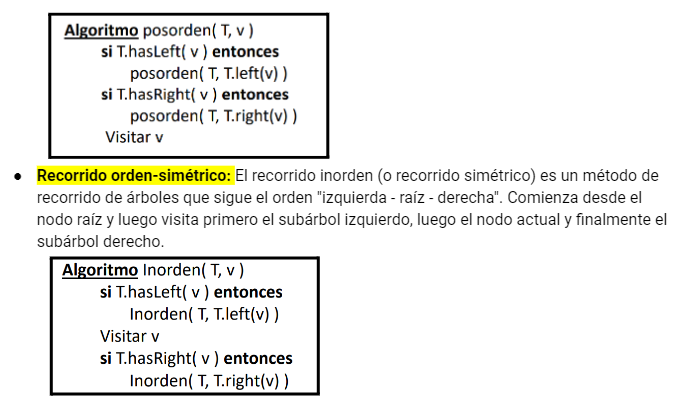
Interfaz BinaryTree<E> EN JAVA

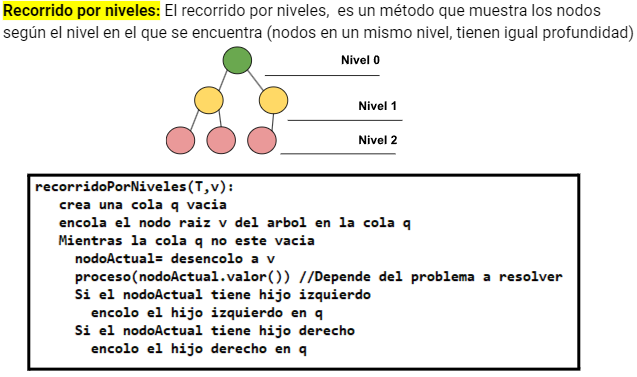


Implementación de la clase de nodos que formaran parte de la estructura de un ARBOL BINARIO

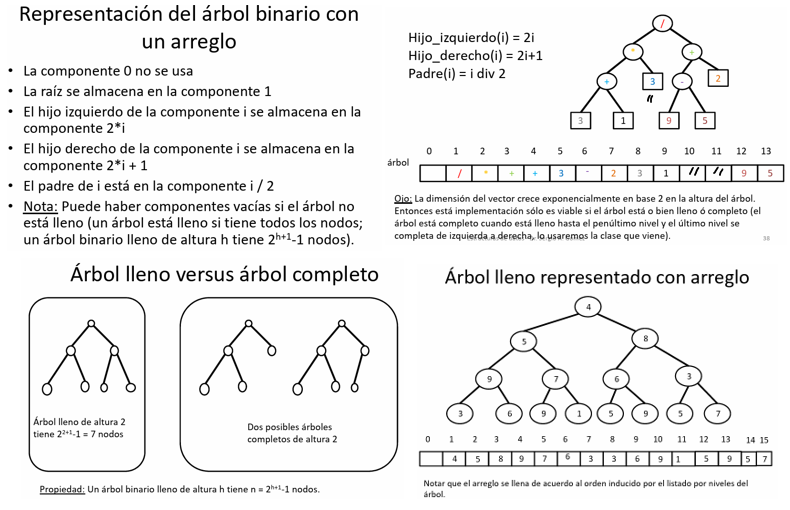
**//Ver implementación del Árbol Binario [ArbolBinarioEnlazado<E> y clases relacionadas]**

* **RECORRIDOS EN ARBOLES BINARIOS:**

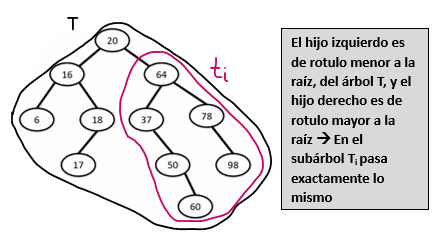




* **Un arbol binario puede representarse a traves de otras estructuras, un ejemplo de ello es la representacion del mismo con un ARREGLO:**



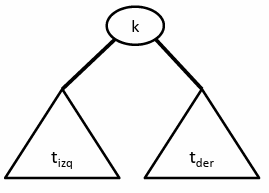


**ESTRUCTURA AUXILIAR ARBOL BINARIO DE BUSQUEDA (ABB):**

* El árbol binario de búsqueda (ABB) es una estructura de datos útil para implementar conjuntos, mapeos y diccionarios
* En un ABB las claves en los nodos se hallan ordenadas de una manera particular
* El tiempo de insertar, recuperar y eliminar es proporcional a la altura del ABB 🡪 Si el árbol tiene n claves, la altura del ABB se halla entre:

1. **h = log2(n) [Mejor Caso]**
2. **h = n [Peor Caso: El arbol es una lista]**

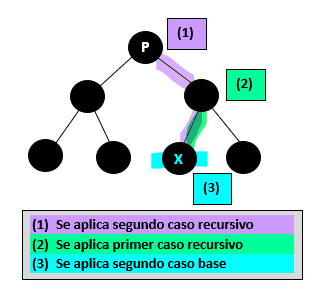
* **No se considera un TDA, mas bien una ESTRUCTURA AUXILIAR que permite modelar otros tipos de datos abstractos**
* **DEFINICION RECURSIBA DE ABB:**

Un ABB es un arbol binario tal que:

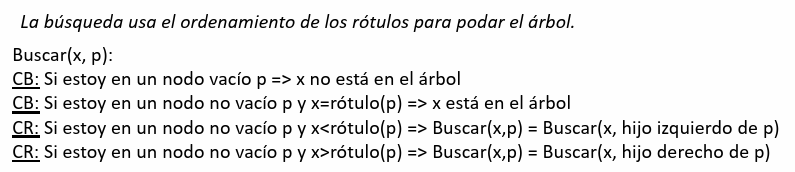
**IMPORTANTE:**

**La organización de las claves va a determinar la forma de inserción, búsqueda y eliminación**

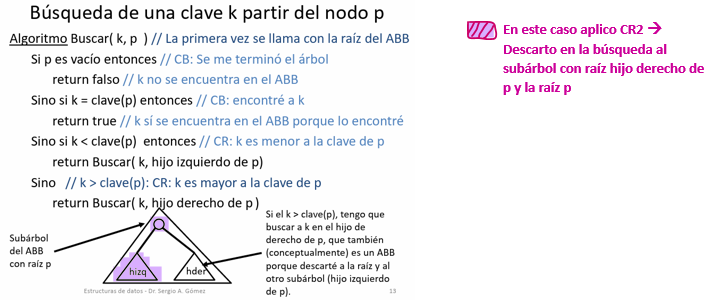
1. Es vacio
2. Es un nodo con rótulo k e hijos tizq y tder tales:
3. k > claves de tizq
4. k < claves de tder
5. tizq y tder son ABB

* **OBJETIVO:** El objetivo del arbol binario de busqueda (ABB) es proporcionar una estructura de datos ordenada y eficiente, para la busqueda, insercion y eliminacion de elementos, aprovechando su propiedad de ordenamiento y su estructura jerarquica
* **BUSQUEDA DE ROTULOS:**

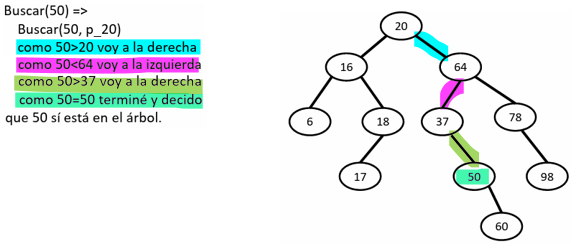
PLANTEO RECURSIVO DEL METODO DE BUSQUEDA:



ALGORITMO RECURSIVO DE BUSQUEDA DE UN NODO EN ABB:



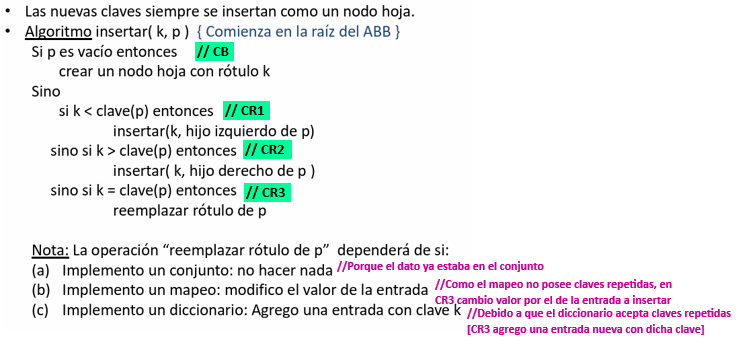
**EJEMPLO: Busco nodo de rotulo 50**



**DATO:**

En el ABB, a medida que busco un elemento, voy descartando partes del árbol en las que se que no voy a encontrarlo [Facilita la búsqueda de un elemento en la ED]

* **INSERCION EN UN ABB:**



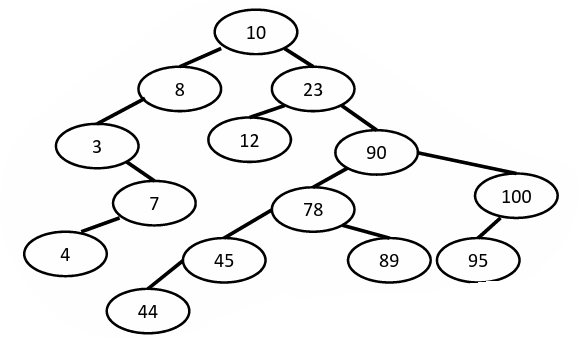
**IMPORTANTE:**

No hay un único árbol posible para un CONJUNTO de claves

El árbol resultante depende del orden en que fueron ingresadas las claves [Da una mejor o peor estructura del árbol, afecta TIEMPO DE EJECUCION]

**VENTAJA A LA HORA DEI NSERATR ELEMENTOS EN UN ABB:**

A diferencia de un arreglo, cuando inserto no necesito reorganizar la estructura

**EJEMPLOS: Insertando secuencias en ABB vacios:**

1. **10, 23, 8, 90, 78, 45, 3, 7, 12, 89, 44, 100, 95, 4**

* Comenzamos en el árbol vacío insertando 10
* Queremos insertar 23, como 23>10, vamos hacia la

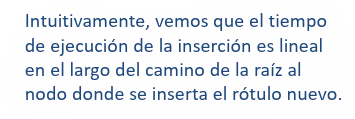
derecha, e insertamos el 23

* Queremos insertar el 8, como 8 < 10 , vamos hacia la

Izquierda, e insertamos 8

* 90 > 10 vamos a la derecha, luego 90 > 23 vamos a

la derecha e insertamos 90

* Como 78 > 10, vamos a la derecha, como 78 > 23,

vamos a la derecha, Como 78 < 90 vamos a la izquierda

e insertamos 78

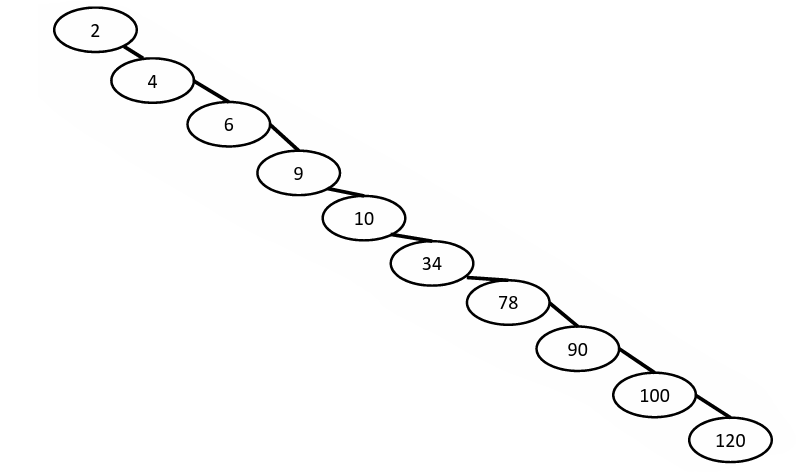
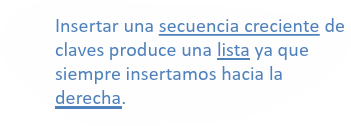
* Como 45 > 10 vamos a la derecha, Como 45 > 23, vamos

a la derecha, como 45 < 90 vamos a la izquierda, como 45 < 78,

vamos a la izquierda de nuevo e insertamos

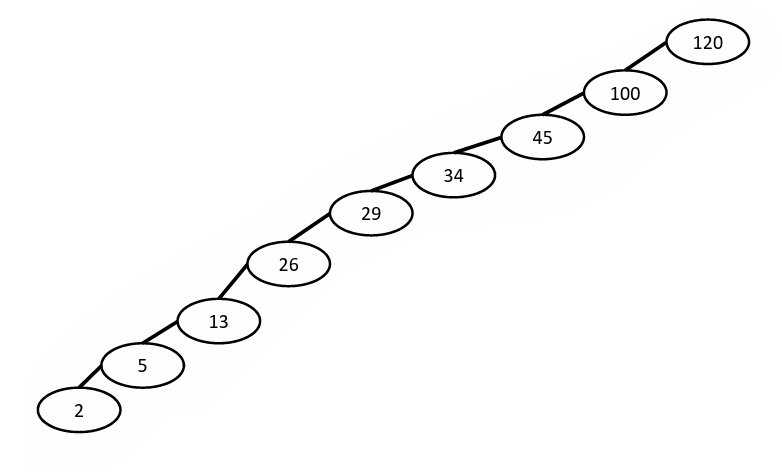
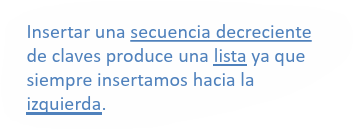
**…. Asi sucesivamente hasta obtener coma resultado el arbol de la imagen, terminando toda la secuencia**

1. **2, 4, 6, 9, 10, 34, 78, 90, 100, 120 🡪 Se repite proceso anterior, pero se obtiene este resultado:**

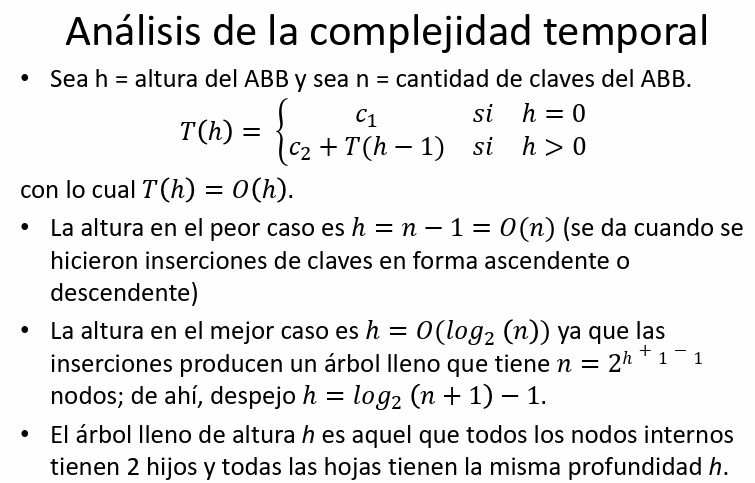


**ABB CON SECUENCIA CRECIENTE: El árbol ABB es una lista en las que a medida que uno avanza por la derecha, las claves son valores cada vez más grandes**

1. **120, 100, 45, 34, 29, 26, 16, 13, 5, 2 🡪** **Se repite proceso anterior, pero se obtiene este resultado:**



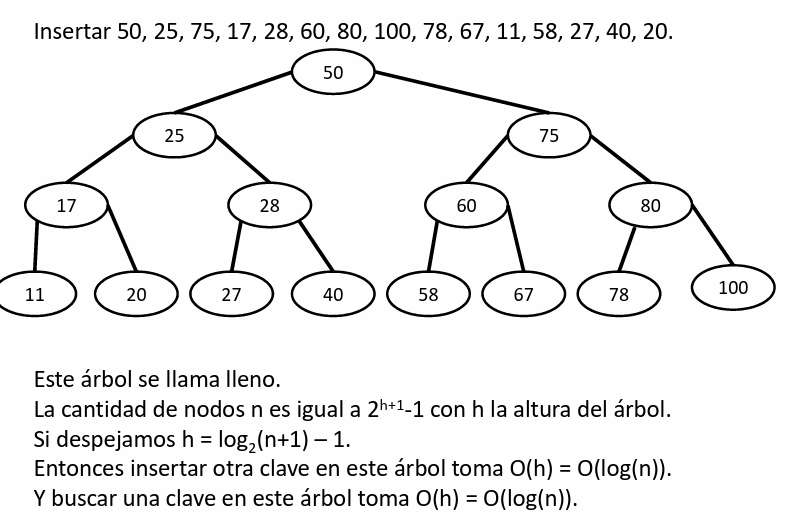
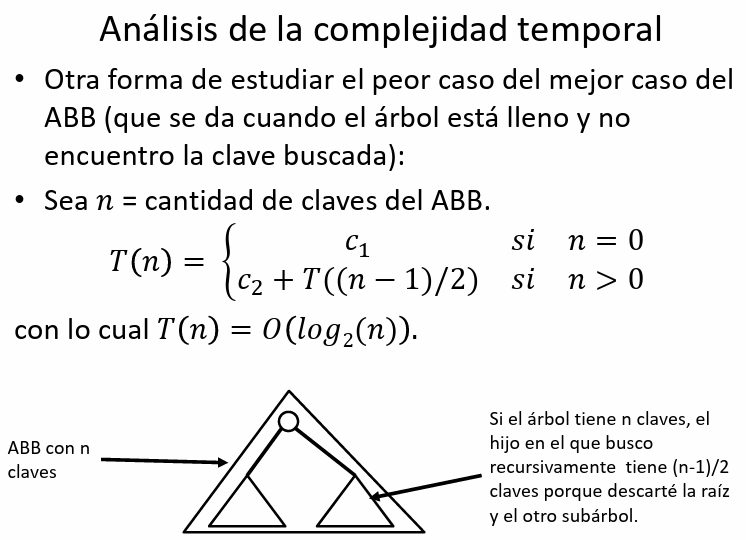
**ABB CON SECUENCIA DECRECIENTE: El árbol ABB es una lista en las que a medida que uno avanza por la izquierda, las claves son valores cada vez más chicos**

* **ANALISIS DE COMPLEJIDAD TEMPORAL EN ABB (Tiempo de ejecucion en function de la altura):**

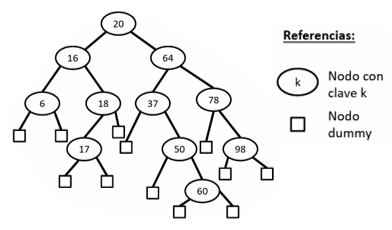
**// Arboles lineales**

**//**

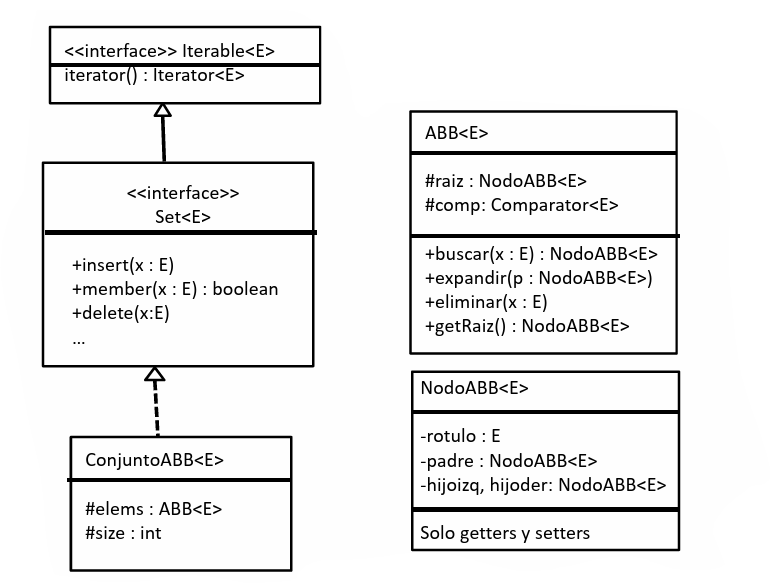
* **MEJOR ARBOL POSIBLE: “ARBOL LLENO DE ALTURA H” [Tiempo de ejecucion en function de la cantidad de nodos del ABB]**



* **IMPLEMENTACION EN JAVA (PRELIMINARES):**
* Para facilitar la programación usaremos un árbol propio donde cada hoja tiene dos hijos DUMMY
* Estos dummies sirven para insertar nuevas claves en su lugar y representan ABB vacíos



* Al insertar un nuevo elemento en la ESTRUCTURA, se intercambia un nodo dummy por uno con el rotulo deseado, y a este ultimo se le crean dos nuevos nodos dummy como hijos, para futuras inserciones
* Ahora un **ARBOL VACIO** se trata nada menos que de un **NODO DUMMY**
* **IMPLEMENTACION DE UN CONJUNTO USANDO UN ABB COMO ESTRUCTURA AUXILIAR:**

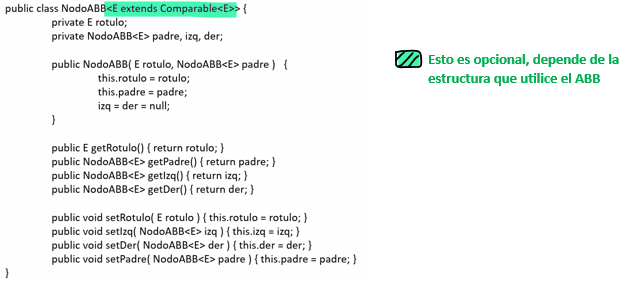


**METODOS DEL ABB:**

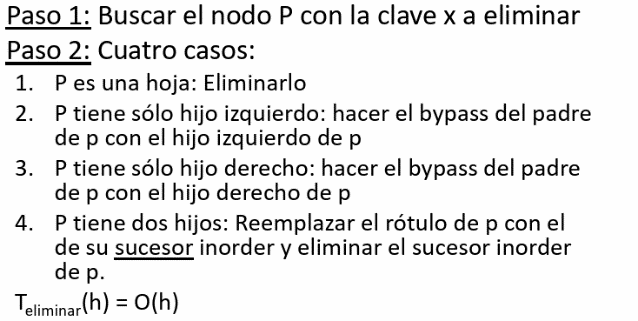
* **Buscar(x)** 🡪 Busca un nodo con rotulo x
* **Expandir(p)** 🡪 Agrega nodos dummy a la hoja p
* **Eliminar(x)** 🡪 Busca y elimina un nodo con rotulo igual a x
* **getRaiz():** Devuelve la raíz del árbol ABB

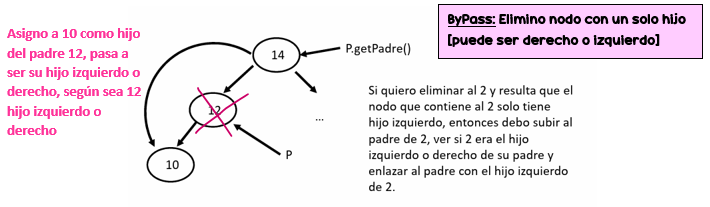
**//Los métodos de un ARBOL BINARIO pueden incluirse**

**//No se agrega el método insert(), ya que depende de la ED que este modelando [depende del TDA que lo utilice]**

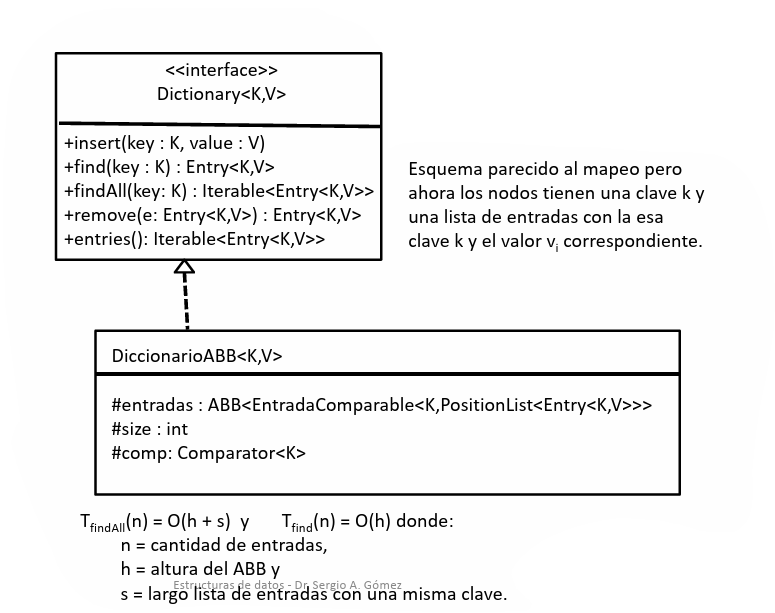
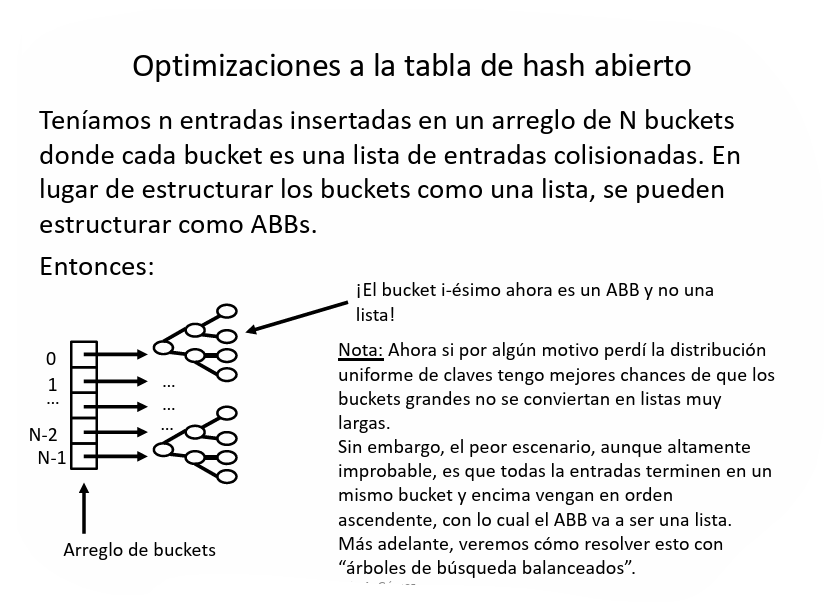
* **CLASE NODOABB : Modela los nodos de un ABB**
* **ELIMINACION EN ARBOL ABB [Dos estrategias]:**

1. **ELIMINACION PEREZOSA [LAZY DELETE]:** Busco el dato a eliminar y marco el nodo como eliminado (sin sacarlo del árbol) 🡪 Para ello necesito un campo más en el NodoABB de tipo Boolean que vale true si el dato ha sido eliminado 🡪 Member e Insert deben ser modificados para tener en cuenta este 🡪 Cuando member encuentra el dato en el árbol debe además ver si el dato no está eliminado 🡪 Cuando insert encuentra el dato debe ver si no es necesario setear eliminado en falso
2. **ELIMINACION FISICA:** Busco el dato y elimino el nodo del árbol (**Ver implementacion en Conjunto<E> y ABB<E>, y en MapeoABB<K,V>**)

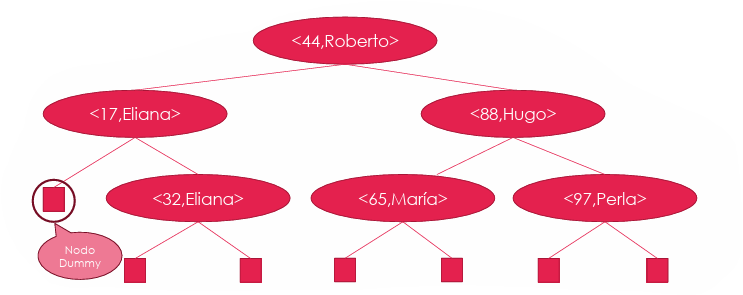


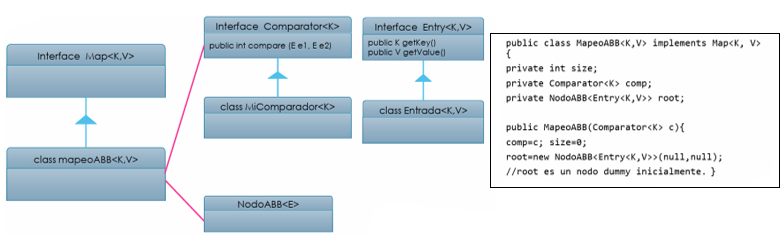


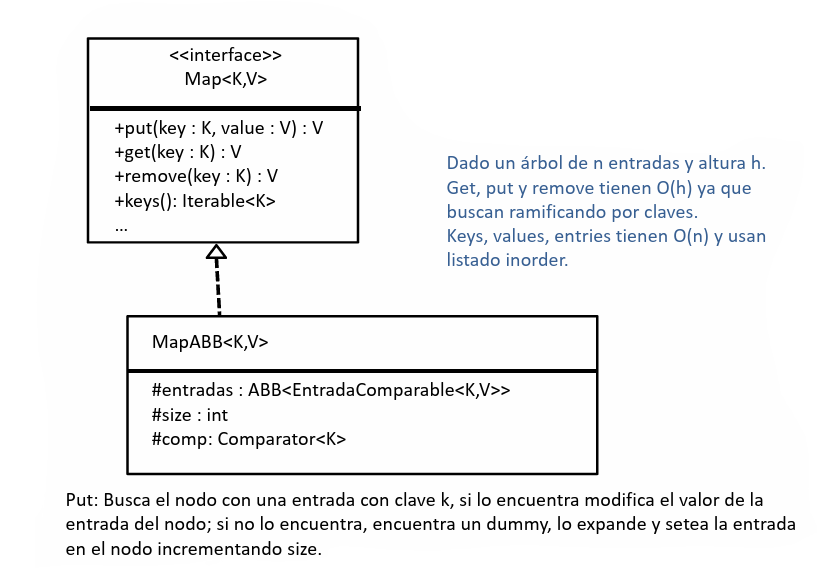
* **IMPLEMENTACION DE UN DICCIONARIO UTILIZANDO UN ABB COMO ESTRUCTURA AUXILIAR:**



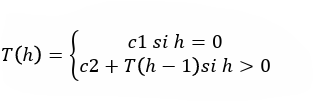
* **IMPLEMENTACION DE UN MAPEO UTILIZANDO UN ABB COMO ESTRUCTURA AUXILIAR:**
* El rótulo del nodo es una entrada (k,v)



* Existen dos posibles implementaciones:

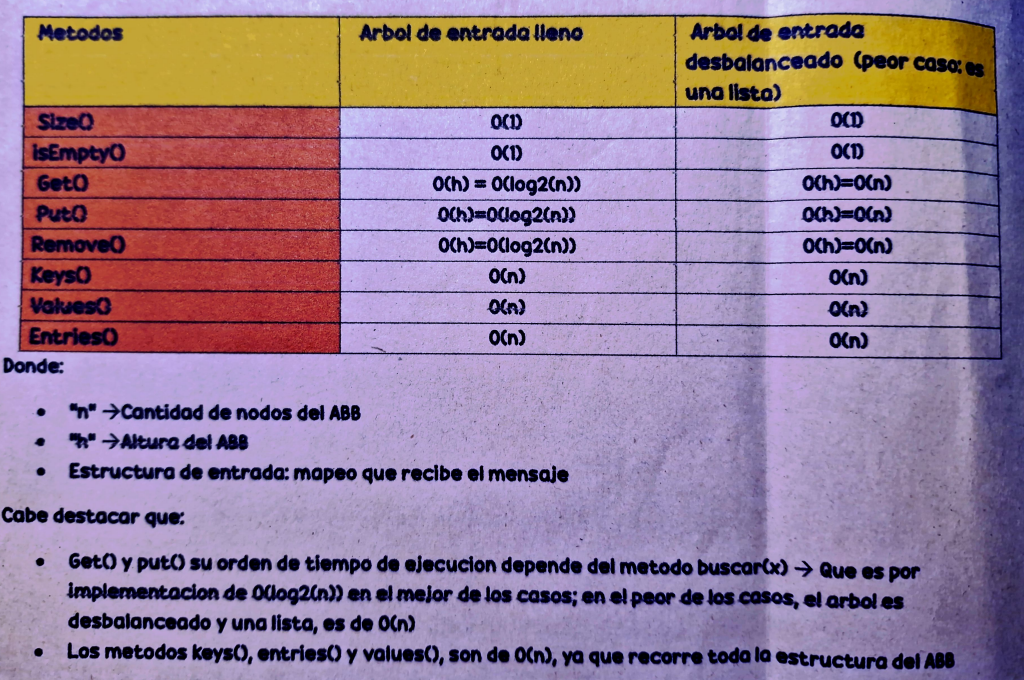


* **ANALISIS DE COMPLEJIDAD TEMPORAL:**

1. Para todas las operaciones que involucran buscar en el ABB, siendo h su altura y n la cantidad de claves:

**T(h) es de O(h)**

1. La altura en el peor caso es ℎ = n (se da cuando se hicieron inserciones de claves en forma ascendente o descendente)
2. La altura en el mejor caso es ℎ =𝑂(log2𝑛) (las inserciones produjeron un árbol balanceado)

**//VER IMPLEMENTACION CORRESPONDIENTE A LA PRIMERA IMPLEMENTACION DE MAPEO CON ABB**