# ABS y AVL (Árboles busqueda binaria)

---

## Conceptos clave:

1. Estructura de datos
2. Diccionario/Conjunto
3. Complejidad/Eficiencia
4. Árboles binarios
5. ABB
6. AVL

Playlist ABB y AVL: https://www.youtube.com/watch?v=62HzsyiQufI&list=PLgKwkU8blA_NxMbxyeHq4l3Zkb4lg51_y

## Repaso

- Planteamos un nuevo capítulo en la materia.
- Dejamos de lado el mundo de la especificación de problemas y nos concentramos en resolver problemas concretos.
- **Problema principal de la algoritmia:** buscar elementos con características particulares dentro de un conjunto de datos (grande o pequeño).

### Diccionario (estructura de dato)

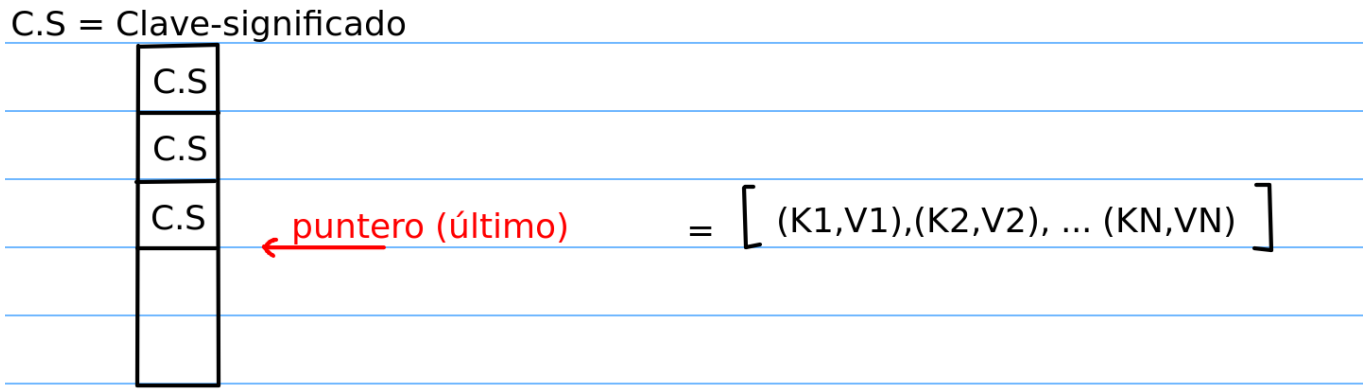
Recordamos la nueva estructura para el TAD diccionario.

**TAD Diccionario**

```
TAD Diccionario<K, V> {
    obs data: dict<K, V>

    proc diccionarioVacio(): Diccionario<K, V>
        asegura res.data == {}

    proc esta(in d: Diccionario<K, V>, in k: K): bool
        asegura res == true <==> k in d.data

    proc definir(inout d: Diccionario<K, V>, in k: K k, in v: V)
        asegura d.data == setKey(old(d).data, k, v)

    proc obtener(in d: Diccionario<K, V>, in k: K): V
        requiere k in d.data
        asegura res == d.data[k]

    proc borrar(inout d: Diccionario<K, V>, in k: K)
        requiere k in d.data
        asegura d.data == delKey(old(d).data, k)
}
```

- Recordamos que los diccionarios los utilizamos cuando queremos asociar un determinado valor K a un valor V (ej: K: manzana -> V: cantidad).
- Otros ejemplos: El diccionario del corrector ortográfico (a cada horror ortográfico nuestro se le asocia la palabra "correcta" que quisimos escribir para poder reemplazarla).
- El padron electoral, con clave el DNI, nombre, domicilio.
- Las claves de usuarios en una base de datos se encuentra en una estructura de tipo diccionario (de forma encriptada la password para prevenir ataques).

Podemos encontrar formas de representar la estructura de Diccionario o Conjuntos, se nos puede ocurrir de manera secuencial.




Podríamos representar la misma idea de diccionario a partir de un arreglo (seq<Clave x Significado>)

- A partir de esta estructura, podemos ver el orden de complejidad de realizar ciertas operaciones.

## Opción 1: Seq <Clave, Significado>, Ult (puntero)

**CrearDiccionario:** O(1) simplemente inicializamos un arreglo vacío y un puntero vacío.

**Definir:** O(1) movemos el puntero una posición para adelante de donde está "guardado" e insertamos el elemento en la posición anterior a donde apunta el puntero.

**Pertenece:** O(n) tendríamos que hacer una busqueda secuencial por todo el arreglo (n en el peor de los casos) a ver si el elemento está en el arreglo.

**ObtenerElemento:** O(n) al igual que pertenece, como peor caso el elemento que queremos obtener puede estar al final del arreglo que planteamos.

- Podríamos pensar también una alternativa a nuestro diccionario **ORDENANDOLO**. Simplificaría la complejidad de buscar/obtener elementos en la secuencia.

## Opción 2: Seq <Clave, Significado>, Ult (puntero) -> Ordenado

**CrearDiccionario:** O(1) simplemente inicializamos un arreglo vacío y un puntero vacío.

**Definir:** O(1) movemos el puntero una posición para adelante de donde está "guardado" e insertamos el elemento en la posición anterior a donde apunta el puntero.
**Pertenece:** O(log n) tendríamos que hacer una busqueda binaria en el arreglo.

**Pertenece:** O(log n) tendríamos que hacer una busqueda secuencial por todo el arreglo (n en el peor de los casos) a ver si el elemento está en el arreglo.

**ObtenerElemento:** O(log n) al igual que pertenece, realizamos busqueda binaria para verificar si está el elemento.

- Tendríamos el problema que cada vez que insertamos un elemento, tenemos que ordenar nuevamente todo el diccionario (es costoso) para que las busquedas se mantengan en tiempo logaritmico.

- *¿Cuál es mejor?* depende del problema.
  -> **Prioridad es buscar**: Opción 2
  -> **Prioridad es manipular la estructura** (insertar, agregar): Opción 1

---

## Árboles (estructura de dato)

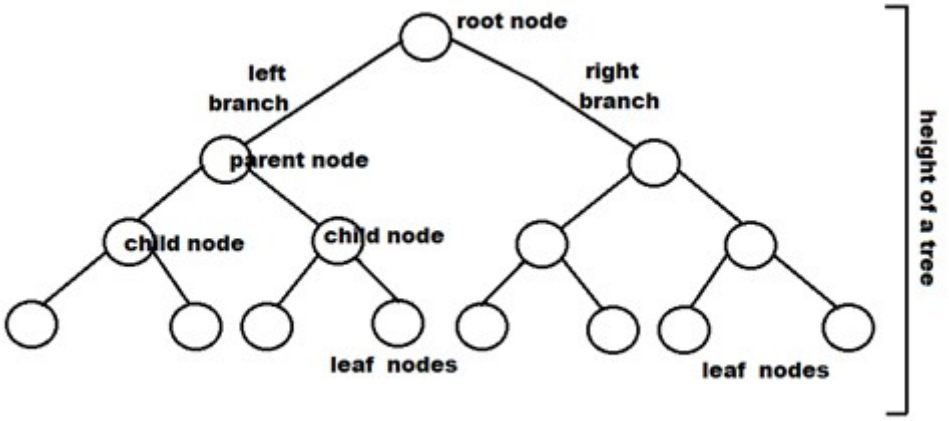
*¿Podríamos encontrar una estructura para los Diccionarios y Conjuntos con una complejidad menor a lineal O(n) para las operaciones de búsqueda?*

-> Vimos que un arreglo ordenado la búsqueda y obtener era log(n).

- Entremos a ver la idea de **Árbol Binario**.

> **Arbol binario**
>
> Un árbol binario es una estructura de datos de tipo árbol la cual, en ningún caso puede ocurrir que un nodo tenga más de dos subarboles. (no pueden salir más de 2 circulitos de cada nodo).

**Ejemplo visual**




- Queremos ver que ventajas tiene utilizar esta estructura de datos para representar la idea de diccionario.
- Veamos las operaciones y su complejidad para el árbol.

## Árbol desordenado

**CrearArbol:** O(1) simplemente inicializamos un arreglo vacío

**Definir:** Dependiendo de como planteamos nuestros arbol el costo de definir. Si queremos un árbol completo (cubro siempre todos los niveles) => O(n) si no tengo puntero al hueco vacío.

**Pertenece:** O(n) tendríamos que hacer una busqueda secuencial por todo el árbol

**ObtenerElemento:** O(n) al igual que pertenece, como peor caso el elemento que queremos obtener puede estar al final del árbol.

*¿Para qué planteamos esta estructura si no es más eficiente que lo planteado antes*
-> Esto da paso a la siguiente estructura

## Árbol Binario de Busqueda (ABB - Binary Search Tree/BST)

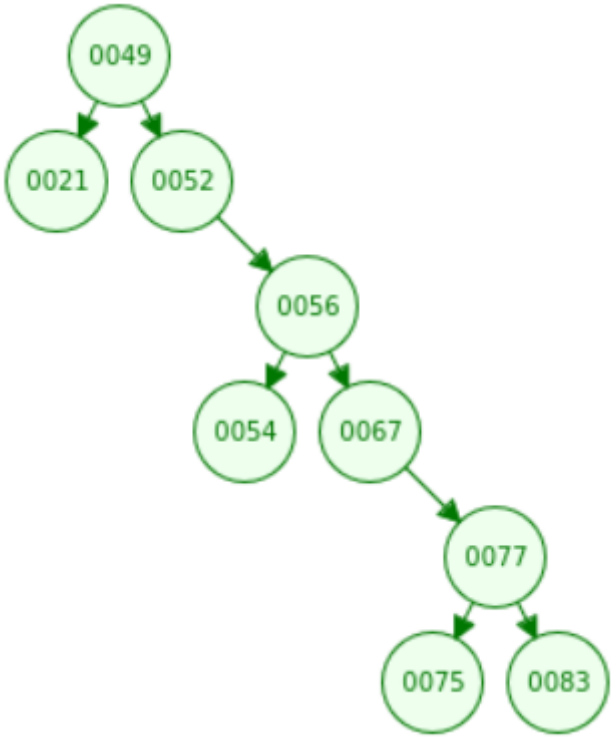
** Insertar diapositiva de ¿Qué es un arbol binario de busqueda?

** Insertar ejemplo de un ABB.

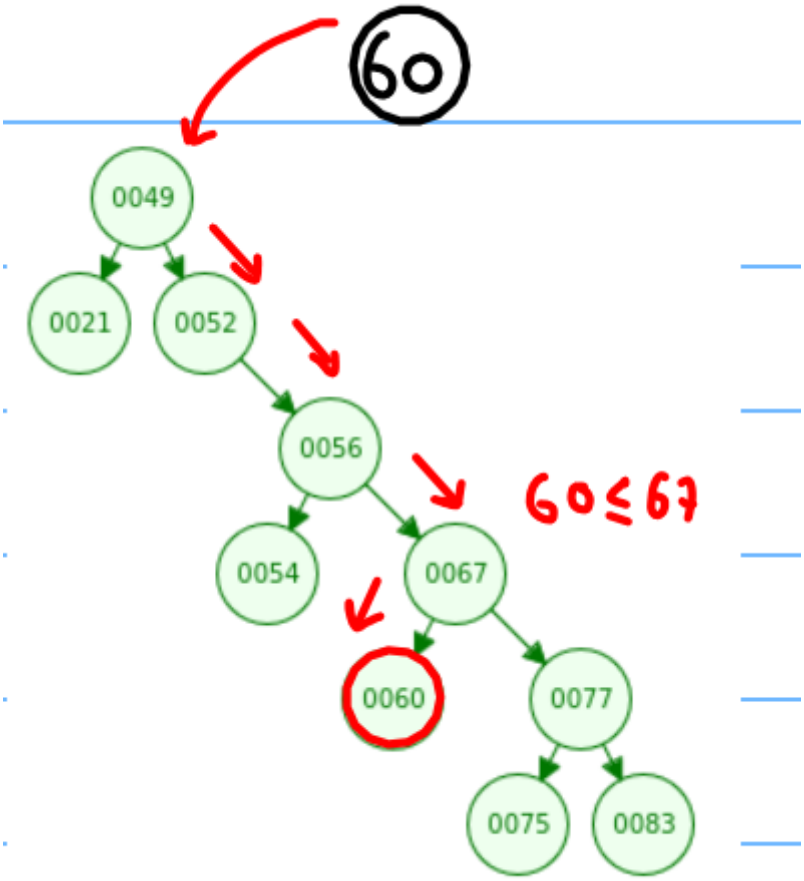
- El invariante de representación para nuestro árbol deberá verificar que:
  -> Todas las claves a la **izquierda** del nodo serán **menores** al valor del nodo donde nos paremos(raiz).
  -> Todas las claves a la **derecha** del nodo serán **mayores** al valor del nodo donde nos paremos(raiz).
  -> Dado una raiz, si agarramos el subarbol izquierdo también tiene que ser un ABB
  -> Dado una raiz, si agarramos el subarbol derecho también tiene que ser un ABB.

**Ejemplo:** Sea D = {49,21,52,56,67,54,77,83,75}

-> La siguiente es la representación de D a partir de un Árbol Binario de Búsqueda




- Si tuviera que insertar un elemento en mi ABB (0060), sería visualmente así.




## Complejidades sobre ABB's

**Obtener**

*Depende de la distancia del nodo a la raíz.*

*-> Peor caso (arreglo ordenado)*

O(n)

-> Caso promedio (Demostración al final)

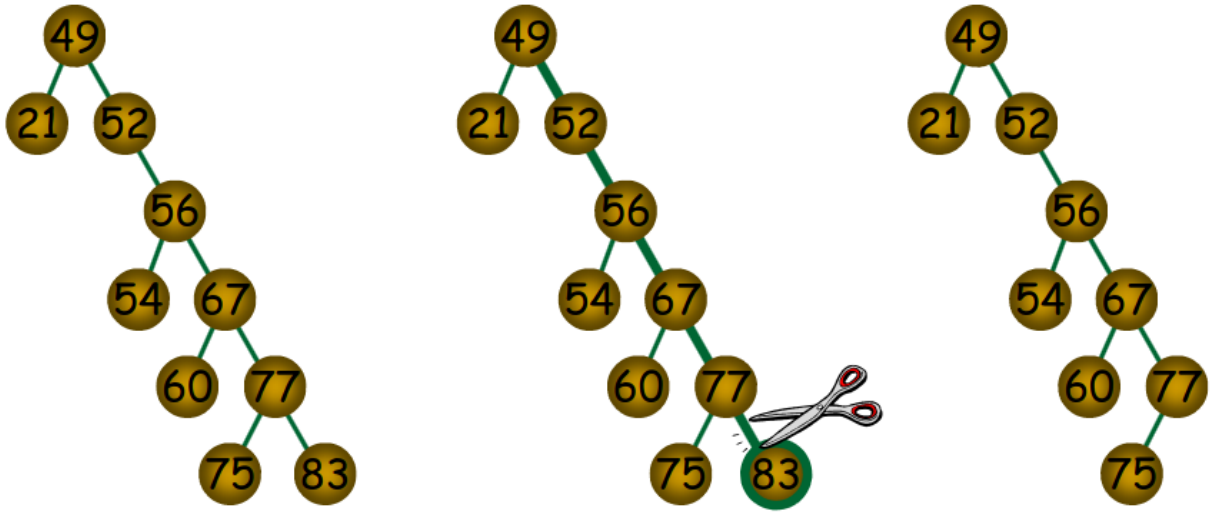
* O(log n)

**Eliminar**

**(caso 1)**

**Borrar nodo sin hijos (hoja)**

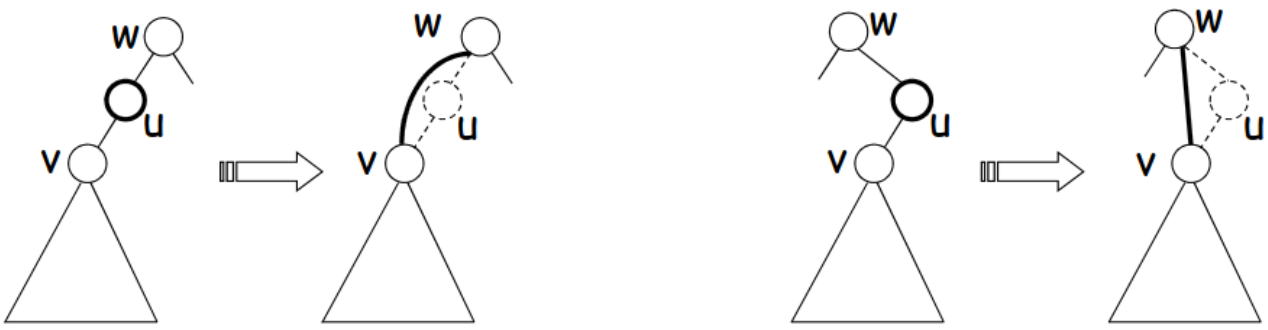
1. **Borrar una hoja**
   - Buscar al padre
   - Eliminar la hoja



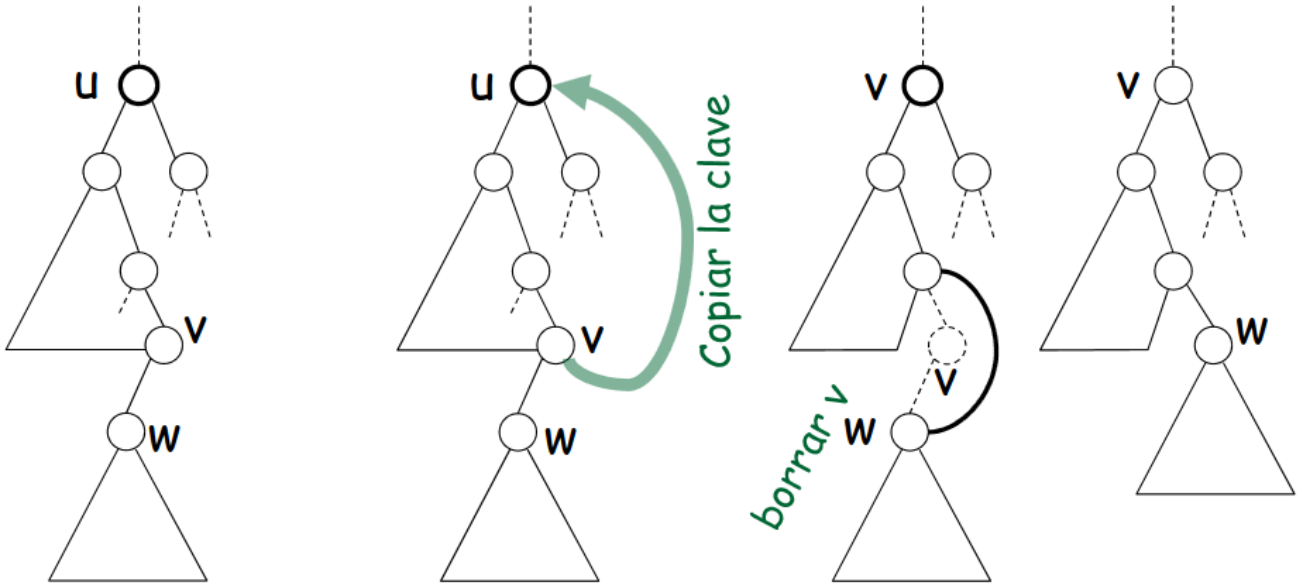

- Solamente busco la raiz que tenga la hoja que queremos eliminar.
  -> Peor caso (arreglo ordenado)
  *O(n)*
  *-> Caso promedio (Demostración al final)*
  O(log n)

**(caso 2)**

2. **Borrar un nodo u con un solo hijo v**
   - Buscar al padre w de u
   - Si existe w, reemplazar la conexión (w,u) con la conexión (w,v)



**(Caso 3)**
**Borrar nodo con dos hijos**




**Conclusión**

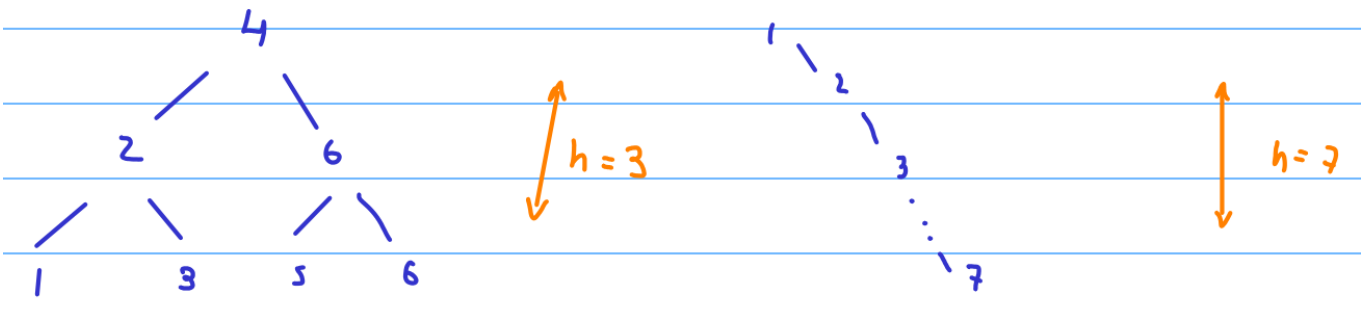
- En el peor caso, ambos costos de eliminar son lineales:
  O(n) + O(n) = O(n)

---

## Notas extra Complejidad

- En los ABB, podemos ver que la complejidad tiene mucho que ver con la "profundida" de nuestro arbol.
- El caso de la derecha es casi igual a una lista enlazada.

- Si tenemos los siguientes árboles, vemos que el segundo es más costoso que el primero porque tenemos que comparar linea por linea, en el primero solamente comparamos dependiendo la altura del árbol.



- Si bien n = 7 para ambos árboles, el costo de insertar en el árbol izquierdo es menor al árbol derecho.

-> Decimos que el árbol izquierdo está **Balanceado** y el derecho **Desbalanceado** (porque los elementos están ordenados).