# Árboles Binarios de Búsqueda y AVL

Algoritmos y Estructuras de Datos





## TAD Conjunto

- Ya hablamos un montón del TAD Conjunto
- Y lo diseñamos de diferentes formas
  - TAD Conjunto Acotado: Array de elementos, Bit Array
  - TAD Conjunto: Lista Enlazada

```
TAD ConjAcotado<ℕ> {
  obs set: conj<ℕ>
  obs cota: ℕ

  proc conjVacío(in cota: ℕ): ConjAcotado<ℕ>
    asegura{res.set = {} ∧ res.cota = cota}

  proc pertenece(in c: ConjAcotado<ℕ>, in e: ℕ): bool
    asegura{res = true ↔ e ∈ c.set}

  proc agregar(inout c: ConjAcotado<ℕ>, in e: ℕ)
    requiere{c = C0 ∧ e ≤ c.cota}
    asegura{c.set = C0.set ∪ ⟨e⟩}
```



## TAD Diccionario

- También podemos usar un TAD un poco diferente (no mucho)
- Diccionario: como conjuntos, pero cada elemento tiene Clave y Valor
- Se puede especificar con $seq < KxV >$, $conj < KxV >$, $dict < K, V >$

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

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

  proc está(in d: Diccionario<K, V>, in k: K): bool
    asegura{res = true ↔ k ∈ d.data}

  proc definir(inout d: Diccionario<K, V>, in k: K, in v: V)
    requiere{d = D0}
    asegura{d.data = setKey(D0.data, k, v)}

  proc borrar(inout d: Diccionario<K, V>, in k: K)
    requiere{d = D0 ∧ k ∈ d.data}
    asegura{d.data = delKey(D0.data, k)}

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

## Diseño de Conjuntos y Diccionarios

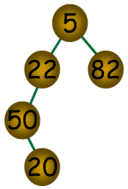
### Diseño sobre Arrays

- Conjuntos y diccionarios pueden representarse a través de arrays (con o sin repetidos, ordenados o desordenados).
- Ya vimos varias de esas soluciones.
- Intenten hacer ustedes diseñar todas estas alternativas, con sus INV, ABS, y los algoritmos
- Complejidad de las operaciones: depende de la implementación, pero
  - Tiempo: alguna de las operaciones requieren O(n) en el peor caso
  - Espacio: O(n).
  - ¿se podrá hacer mejor?



## Árboles/Árboles Binarios

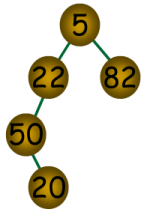
- Podemos definir el tipo conceptual (matemático) árbol$< T >$.
- Así como con las secuencias, podemos definir árboles de cualquier tipo T
- Se puede definir recursivamente como
  - Nil es un árbol$< T >$
  - Una secuencia que contiene un elemento de T y una secuencia de árboles$< T >$, es un árbol$< T >$.
- ¡Y se pueden dibujar!
- Ejemplos
  - Nil
  - $< 5, < Nil, Nil >>$
  - $< 5, << 22, < 50, < Nil, < 20, < Nil, Nil >>> , Nil >, < 82, < Nil, Nil >>$



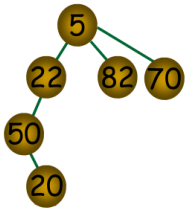



## Árboles/Árboles Binarios

- Sobre árboles, usamos terminología variada:
  - botánica: raíz, hoja
  - genealógica: padre, hijo, nieto, abuelo, hermano
  - física: arriba, abajo
  - topológica: nodo interno, externo
- Hay un tipo particular de árboles, que son los Árboles Binarios:la secuencia de árboles tiene como máximo dos elementos




Árbol Binario




Árbol No Binario



## Árboles/Árboles Binarios

- El concepto matemático árbol tiene muchos usos, propiedades y funciones muy conocidas.
- Por ejemplo, dado un árbol a, podemos hablar de nil?(a), raiz(a), altura(a), elementos(a), está(e,a) y muchas más.
- Y para árboles binarios, también izq(a) y der(a)
- Esas funciones se puede definir recursivamente, por ejemplo:

  Altura(nil) $= 0$

  Altura$(< a, s >) = 1 + max_i(\text{altura}(s_i))$

## Árboles/Árboles Binarios

Especificación del TAD

- Podemos definir el TAD Arbol y el TAD Arbol Binario
- El TAD Arbol Binario podría tener versiones de todas las operaciones conceptuales y los constructores nil y bin(c,I,D).

Diseño del TAD

- Podemos implementar Árboles binarios con punteros:
  `Nodo = Struct <dato: N, izq: Nodo, der: Nodo >`

  ```
  Módulo AB implementa Árbol Binario {
       raíz: Nodo
  }
  ```
- Y podemos escribir un invariante de representación, y una función de abstracción (pero no lo vamos a hacer acá).



## Representación de conjuntos y diccionarios a través de AB

- ¿Podríamos representar conjuntos o diccionarios a través de árboles binarios?
- Claro que podríamos
- ¿Ganaríamos algo? En principio, no demasiado
- Sin importar cómo diseñemos un AB, las operaciones seguirán siendo lineales (O(n))
- Pero...





## Árboles Binarios de Búsqueda (ABB)

¿Qué es un Árbol Binario de Búsqueda?

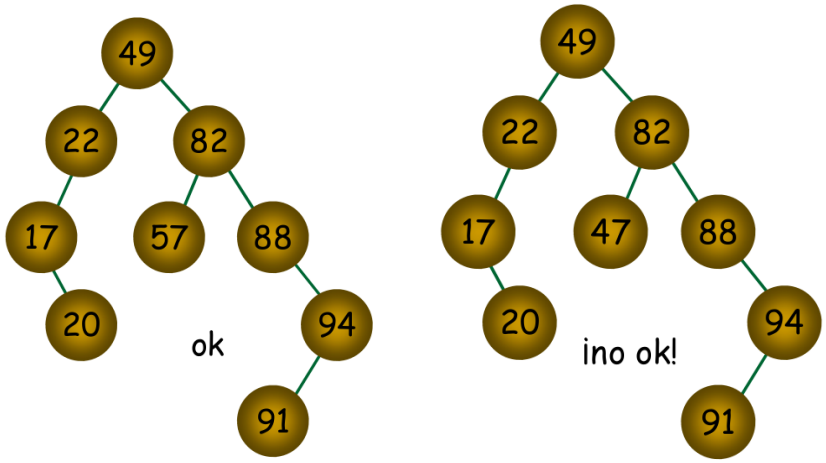
- Es un AB que satisface la siguiente propiedad:
- Para todo nodo, los valores de los elementos en su subárbol izquierdo son menores que el valor del nodo, y los valores de los elementos de su subárbol derecho son mayores que el valor del nodo
- Dicho de otra forma, el valor de todos los elementos del subárbol izquierdo es menor que el valor de la raíz, el valor de todos los elementos del subárbol derecho es mayor que el valor de la raíz, y tanto el subárbol izquierdo como el subárbol derecho son ABB.

Formalmente

`pred esABB(a: ArbolBinario<` $T$ `>)`

$$\{a = Nil \vee ($$
$$(\forall e : T)(e \in elems(a.Izq) \rightarrow e \leq a.dato) \wedge$$
$$(\forall e : T)(e \in elems(a.Der) \rightarrow e > a.dato) \wedge$$
$$esABB(d.Izq) \wedge esABB(d.Der)\ )\}$$

## Ejemplos






## Conjunto sobre ABB

### Rep y Abs

```
Nodo = Struct <dato: N, izq: Nodo, der: Nodo >

Modulo ConjSobreABB implementa Conjunto {
      raíz: Nodo

      pred invRep(c: ConjSobreABB)
            {esArbol(c.raiz)∧_L esABB(c.raiz)}

      pred predAbs(c: ConjSobreABB, c':Conjunto)
            {(∀e : T)(alcanzable(e, c) ↔ e ∈ c')}

      pred esABB(c: ConjSobreABB)
            {c.raiz = Null ∨ (
            (∀e : T)(alcanzable(e, c.raiz.izq) → e ≤ c.raiz.dato) ∧
            (∀e : T)(alcanzable(e, c.raiz.der) → e > c.raiz.dato) ∧
            esABB(d.raiz.izq) ∧ esABB(d.riaz.der) )}

      pred alcanzable(e: T, n:Nodo)
            {Null ≠ n.raiz.dato ∨_L e = n.raiz.dato ∨
            alcanzable(e, n.raiz.izq) ∨ alcanzable(e, n.raiz.der)} }
```



## Conjunto sobre ABB

### Algoritmos - Crear Vacío y Agregar

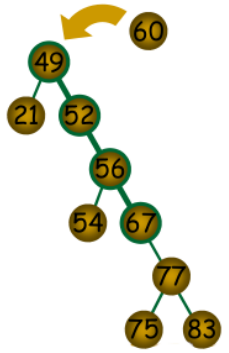
```
Nodo = Struct <dato: N, izq: Nodo, der: Nodo >

Modulo ConjSobreABB implementa Conjunto {
       raíz: Nodo

       proc conjVacio():ConjSobreABB
         res.raiz = Null;
         return res

       proc agregar(inout c:ConjSobreABB, in e: T)
         if c.raiz == Null:
            c.raiz.data = e;
            c.raiz.izq = Null;
            c.raiz.der = Null;
         else:
            if e <= c.raiz.dato:
               agregar(c.raiz.izq, e);
            else:
               agregar(c.raiz.der, e);
         return res

}
```



## Conjunto sobre ABB

### Algoritmos - Agregar

- Lo hicimos de forma recursiva
- Pero lo que estamos haciendo es:
  1. Buscar el padro del nodo a insertar
  2. Insertarlo como hijo de ese padre
- Costo de la inserción: Depende de la distancia del nodo a la raiz
  - En el peor caso:
    - O(n)
  - En el caso promedio (suponiendo una distribución uniforme de las claves):
    - O(log n)



# Conjunto sobre ABB

## Algoritmos - Borrar

A la hora de borrar un elemento $u$ de un ABB podemos tener 3 casos:

1. $u$ es una hoja
2. $u$ tiene un solo hijo
3. $u$ tiene dos hijos

Veamos caso por caso



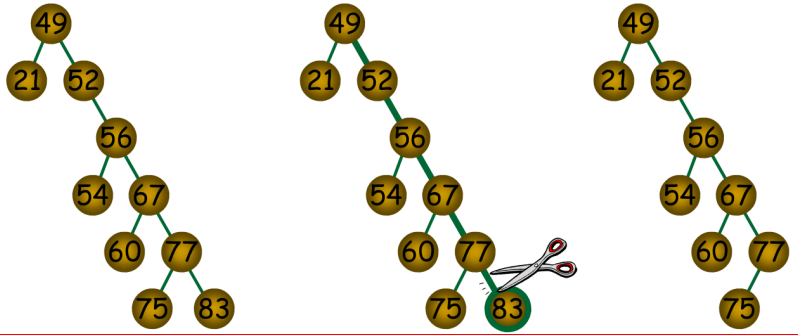
# Conjunto sobre ABB

## Algoritmos - Borrar hoja

1. Borrar un nodo sin hijos (hoja)

   Este es el caso más sencillo: el nodo a borrar no tiene conexiones a otros nodos.

   - Buscar al padre (es el que tiene el puntero a nuestra hoja)
   - Eliminar la hoja (poner Null en el campo correspondiente del nodo)





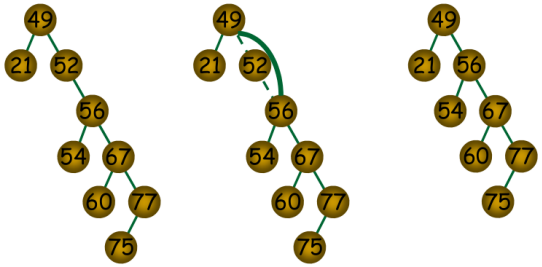

# Conjunto sobre ABB

## Algoritmos - Borrar nodo con 1 hijo

2 Borrar un nodo $u$ con un solo hijo $v$

Este caso no es tan complejo: hay que asegurar que el hijo no quede desconectado.

- Buscar al padre de $u$ ($w$)
- Si existe $w$, reemplazar la conexión $(w, u)$ con la conexión $(w, v)$
- Ahora ya podemos eliminar el elemento (si queremos)





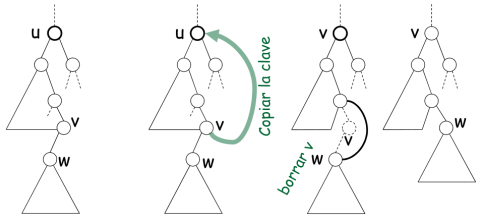

# Conjunto sobre ABB

## Algoritmos - Borrar nodo con 2 hijos

3 Borrar un nodo $u$ con 2 hijos

Este es el caso complicado: para mantener el invRep tenemos que reemplazar $u$ por un elemento que sea mayor a todo el subarbol izquierdo y menor a todo el subarbol derecho

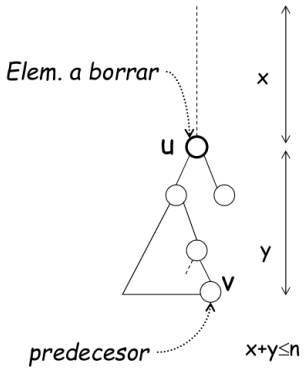
- Buscar al predecesor inmediato de $u$ (máximo del subarbol izquierdo)
- También podemos buscar al sucesor inmediato de $u$ (mínimo del subarbol izquierdo)
- Cualquiera de los dos ($v$) va a tener como máximo 1 hijo
- Copiar el elemento encontrado en el nodo $u$
- Borrar el nodo $v$ (si o si entra en los casos anteriores)



## Conjunto sobre ABB

Algoritmos - Borrar

- Costo del borrado:
- El borrado de un nodo interno requiere encontrar:
  - al nodo que hay que borrar
  - y a su predecesor inmediato
- En el peor caso:
  - Encontrar el que tengo que borrar: O(n)
  - Encontrar predecesor: O(n)
- Es decir: O(n)






## Conjunto sobre ABB

- Hasta ahora vimos que podemos representar Conjuntos (y Diccionarios) con árboles
- Usar Árboles Binarios de Búsqueda parece una buena idea
- Pero en el Peor Caso vamos a seguir tiendo complejidad Lineal
- Principalmente porque el árbol puede estar **desbalanceado**
- ¿Cómo podemos mejorar esto?
- ¿Hay estructuras más eficientes?



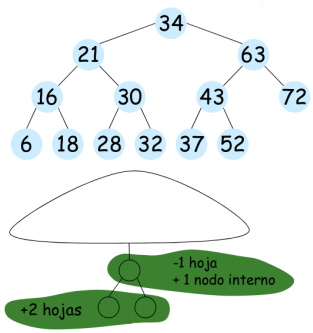


## Introducción al balanceo

- ¿Qué altura tiene un árbol completo (todos los nodos con $k$ hijos)?
  - $\mid arbol(n) \mid = log_k(n)$
- Pero...no podemos pretender tener siempre árboles completos (¡mantenerlos completos sería demasiado caro!)
- Quizás con alguna propiedad más débil...
- Noción intuitiva de balanceo
  - Todas las ramas del árbol tienen "casi" la misma longitud
  - Todos los nodos internos tienen "muchos" hijos
- Caso ideal para un arbol k-ario
  - Cada nodo tiene 0 o k hijos
  - La longitud de dos ramas cualesquiera difiere a lo sumo en una unidad

## Balanceo

- Teorema: Un árbol binario perfectamente balanceado de $n$ nodos tiene altura $\lfloor log_2(n) \rfloor + 1$
- Demotración: Si cada nodo tiene 0 o 2 hijos
  - "Podamos" el árbol eliminando primero las hojas de las ramas más largas
  - Luego podamos todas las hojas, nos queda otro árbol con las mismas características (cantidad de hijos por nodo y balanceo)
  - ¿Cuántas veces podemos "podar" el árbol?
  - Fácilmente generalizable a árboles k-arios
    $n_h = (k-1)n_i + 1 \Rightarrow n_h = \frac{(k-1)n+1}{k}$
  - costo de búsqueda/inserción/borrado O(log n)
  - Pero... sucesiones de inserciones y borrados pueden destruir el balanceo!





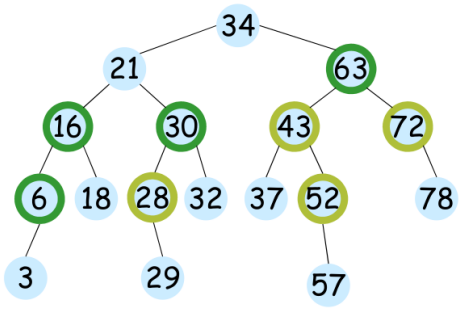



## AVL

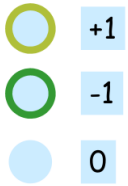
- Un árbol se dice balanceado en altura si las alturas de los subárboles izquierdo y derecho de cada nodo difieren en a lo sumo una unidad
- Árboles AVL:
  - Fueron propuestos en 1962
  - Se llaman árboles AVL, por sus creadores Gueorgui Maksimovich **A**delsón-**V**elski (1922-2014) y Yevgueni Mijáilovich **L**andis (1921-1997)
  - La ventaja principal de éstos árboles es que mantienen un nivel de balanceo que permiten que todas las operaciones sean en máximo O(log n)

# AVL

## Factor de Balanceo



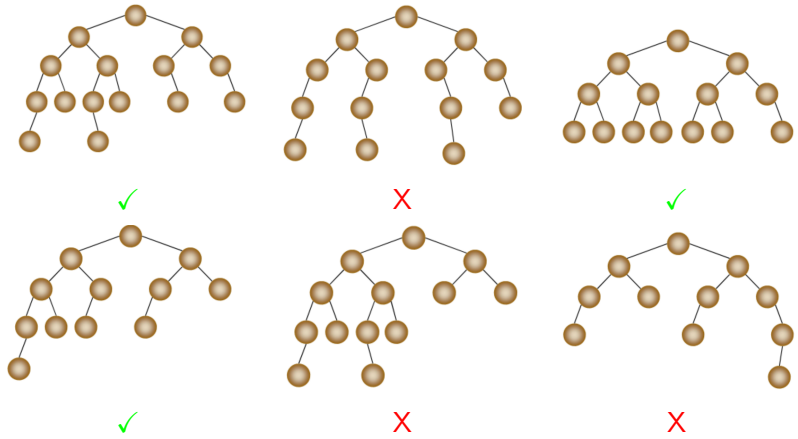

*factor de balanceo (FDB)*:
altura subárbol Der -
altura subárbol Izq




- En un árbol balanceado en altura | $FDB$ |$\leq 1$, para cada nodo
- Si tenemos algún nodo con | $FDB$ |$> 1$, no está balanceado (no es AVL)



# AVL








# AVL

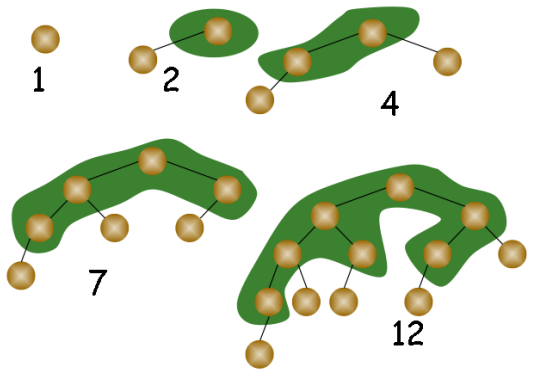
## Árboles de Fibonacci

Árboles de Fibonacci:

- Tiene todos los factores de balanceo de sus nodos internos $\pm 1$
- Es el árbol balanceado más cercano a la condición de no-balanceo
- Un árbol de Fibonacci con n nodos tiene altura $< 1{,}44 lg(n+2) - 0{,}328$
  - demostrado por Adelson-Velskii y Landis
  - $\Rightarrow$ un AVL de n nodos tiene altura $\Theta$(lg n)

# AVL

## Árboles de Fibonacci




| $h$ | $F_h$ | $AVL_h$ |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 1 | 1 |
| 2 | 1 | 2 |
| 3 | 2 | 4 |
| 4 | 3 | 7 |
| 5 | 5 | 12 |
| 6 | 8 | 20 |
| 7 | 13 | 33 |