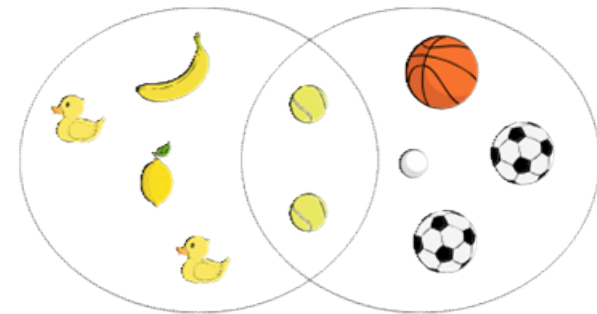


# Diseño de Conjuntos y Diccionarios



# Representación de Conjuntos y Diccionesarios



- **TAD Diccionario(clave, significado)**
- **Observadores básicos**
- $\text{def?}: \text{clave } c \times \text{dicc}(\text{clave}, \text{significado}) d \rightarrow \text{bool}$
- $\text{obtener}: \text{clave } c \times \text{dicc}(\text{clave}, \text{significado}) d \rightarrow \text{significado } (\text{def?}(c, d))$
- **Generadores**
- $\text{vacío}: \rightarrow \text{dicc}(\text{clave}, \text{significado})$
- $\text{definir}: \text{clave} \times \text{sign} \times \text{dicc}(\text{clave}, \text{significado}) \rightarrow \text{dicc}(\text{clave}, \text{significado})$
- **Otras Operaciones**
- $\text{borrar}: \text{clave } c \times \text{dicc}(\text{clave}, \text{significado}) d \rightarrow \text{dicc}(\text{clave}, \text{significado}) (\text{def?}(c, d))$
- $\text{claves}: \text{dicc}(\text{clave}, \text{significado}) \rightarrow \text{conj}(\text{clave})$
- $\cdot = \text{dicc} \cdot : \text{dicc}(\alpha) \times \text{dicc}(\alpha) \rightarrow \text{bool}$

# Definición del TAD Conjunto

```
TAD Conjunto<T> {  
  obs elems: conj<T>
```

```
  proc conjVacio(): Conjunto<T>
```

```
    asegura res.elems == {}
```

```
  proc pertenece(in c: Conjunto<T>, in T e): bool
```

```
    asegura res == true <==> e in c.elems
```

```
  proc agregar(input c: Conjunto<T>, in e: T)
```

```
    asegura c.elems == old(c).elems + {e}
```

```
  proc sacar(inout c: Conjunto<T>, in e: T)
```

```
    asegura c.elems == old(c).elems - {e}
```

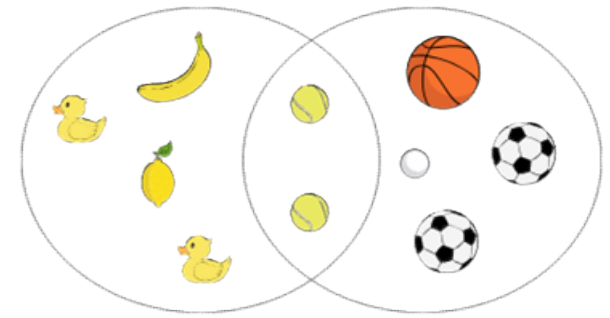
```
  proc unir(inout c: Conjunto<T>, in c': Conjunto<T>)
```

```
    asegura c.elems == old(c).elems + c'.elems
```

```
  proc restar(inout c: Conjunto<T>, in c': Conjunto<T>)
```

```
    asegura c.elems == old(c).elems - c'.elems
```

```
}
```



# Definición del TAD Conjunto

```
TAD Conjunto<T> {  
  obs elems: conj<T>
```

```
  proc conjVacio(): Conjunto<T>
```

```
    asegura res.elems == {}
```

```
  proc pertenece(in c: Conjunto<T>, in T e): bool
```

```
    asegura res == true <==> e in c.elems
```

```
  proc agregar(input c: Conjunto<T>, in e: T)
```

```
    asegura c.elems == old(c).elems + {e}
```

```
  proc sacar(inout c: Conjunto<T>, in e: T)
```

```
    asegura c.elems == old(c).elems - {e}
```

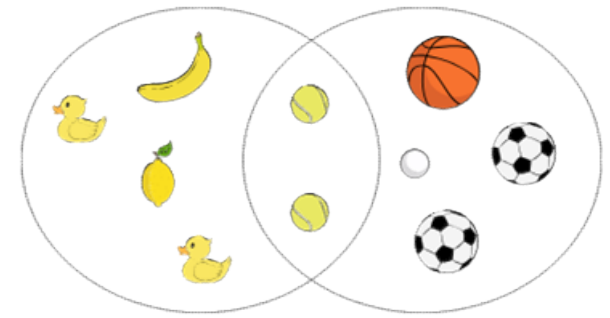
```
  proc unir(inout c: Conjunto<T>, in c': Conjunto<T>)
```

```
    asegura c.elems == old(c).elems + c'.elems
```

```
  proc restar(inout c: Conjunto<T>, in c': Conjunto<T>)
```

```
    asegura c.elems == old(c).elems - c'.elems
```

```
}
```



# Y del 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, 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)
```

```
}
```

# ¿Conjuntos y Dictionarios?



- Vamos a pensar implementaciones de esos diccionarios, pero de paso, otras variantes:
  - Más de un significado es posible
    - Listas de significados
    - Conjuntos de significados
    - ¿qué obtenemos al obtener?
    - ¿y qué borramos al borrar?
  - Diccionarios con un solo significado posible (o sea  $|K|=1$ )
  - Los conjuntos son un caso particular de los diccionarios
  - Además, cualquier diccionario pueden ser pensados como si  $K$  fuera “punteros al significado”
  - En conclusión, lo más interesante es pensar en cómo representar conjuntos.

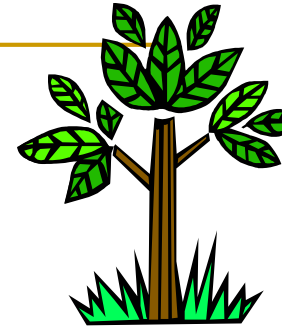


# Representación de conjuntos y diccionarios a través de arrays

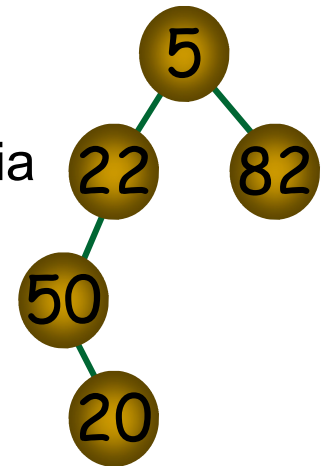
- Conjuntos y diccionarios pueden representarse a través de arrays (con o sin repetidos, ordenados o desordenados).
- Ya vimos varias de esas soluciones.
- Intenten hacer Uds. mismos el ejercicio de escribir INV, ABS, y los algoritmos
- Complejidad de las operaciones: depende de la implementación, pero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



# Árboles/ Árboles Binarios

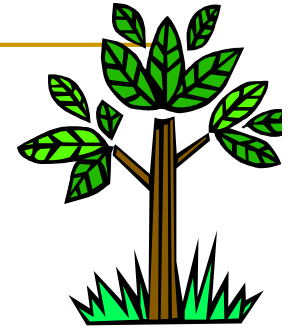


- Podemos definir el tipo conceptual (matemático) árbol<T>.
- Así como con las secuencias, podemos definir árboles de cualquier tipo T
- Se puede definir recursivamente como
  - Nil es un árbol<T>
  - una tupla que contiene un elemento de T y una secuencia de árboles<T>, es un árbol<T>.
- ¡Y se pueden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

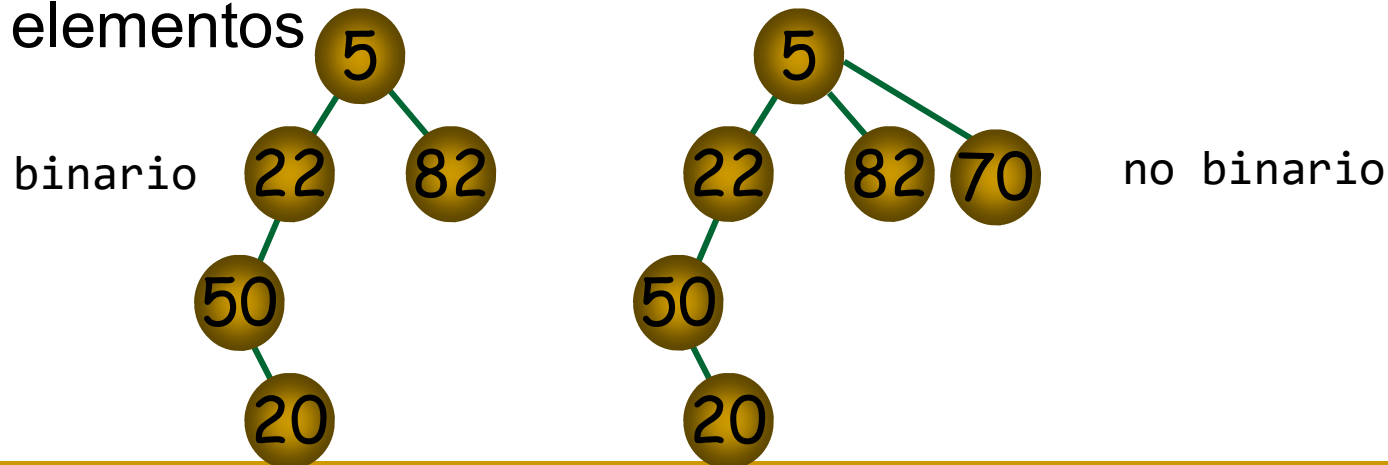




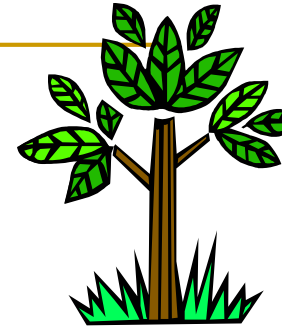
# Árboles/ Árboles Binarios



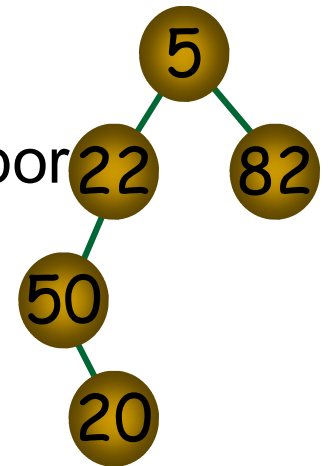
- Sobre árboles, usamos terminología variada:
  - «botánica» (raíz, hoja)
  - «genealógica» (padre, hijo, nieto, abuelo, hermano),
  - «física» (arriba, abajo)
  - «topológica»(?) (nodo interno, externo)
- Hay un tipo particular de árboles, que son los Árboles Binarios: la secuencia de árboles tiene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



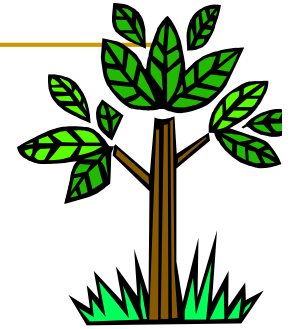
# Árboles / Árboles Binarios



- 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
  - $Altura(nil)=0$
  - $Altura(<a,s>)=1+\max_i\{altura(s_i)\}$



# Árboles/ Árboles Binarios



- 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,l,D)*.

- 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 Árboles Binarios



- ¿Podríamos representar conjuntos o diccionarios a través de árboles binarios?
  - Claro que podríamos
  - ¿Ganaríamos algo? No demasiado en principio ¿no?
  - Pero.....
-

---

# Arboles 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.

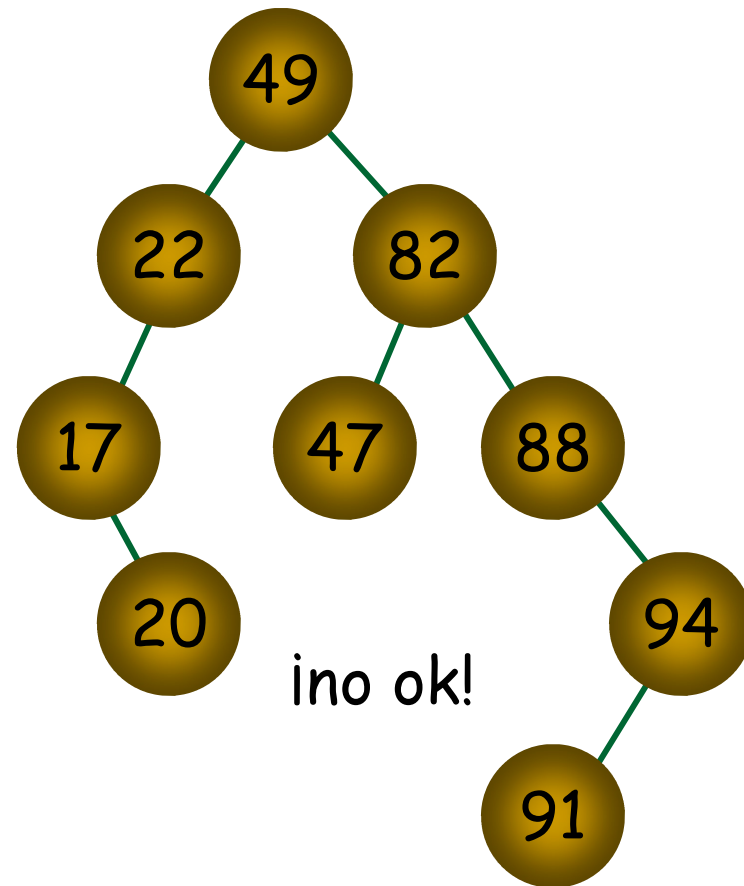
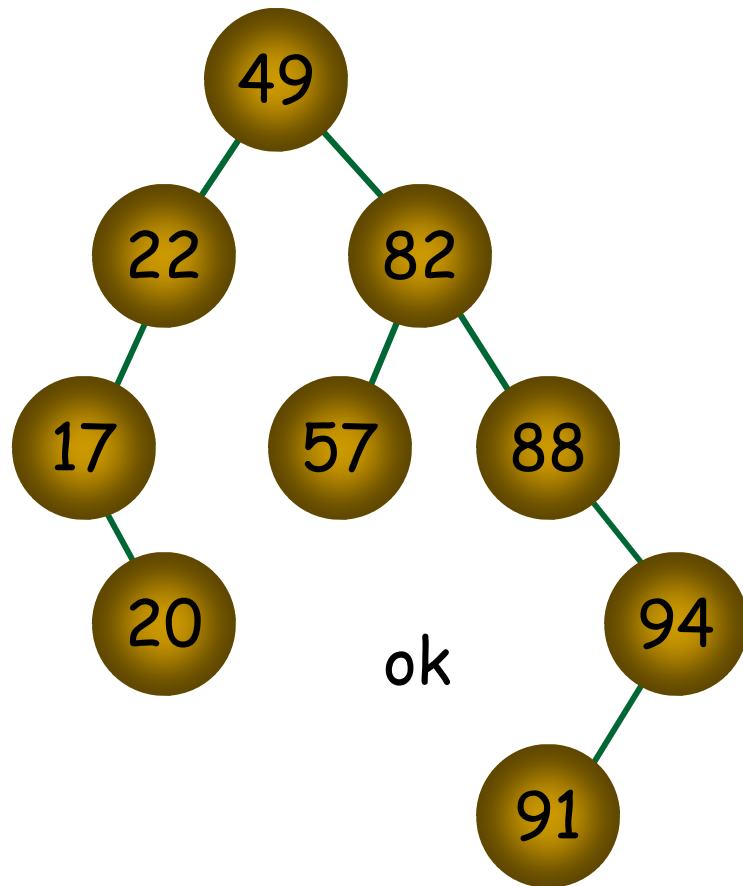
- Formalmente:

`esABB(nil): TRUE`

`esABB(<C,<I,D>>):  $\forall e \in \text{elems}(I) \ e \leq C \wedge \forall e \in \text{elems}(D) \ e > C \wedge \text{esABB}(I) \wedge \text{esABB}(D)$`

---

# Ejemplos



---

# Invariante de Representación

- El invariante de representación de la representación de Conjuntos con Árboles Binarios que son de Búsqueda sería:

pred InvRepABB (e: AB)

$\{ \text{esABB}(e.\text{arbol}) = \text{TRUE} \}$

- La función de abstracción es obvia.
  - Como adelantamos, vamos a dejar de usar la notación que usamos hasta ahora, para pasar a usar....alguna que se entienda.
-

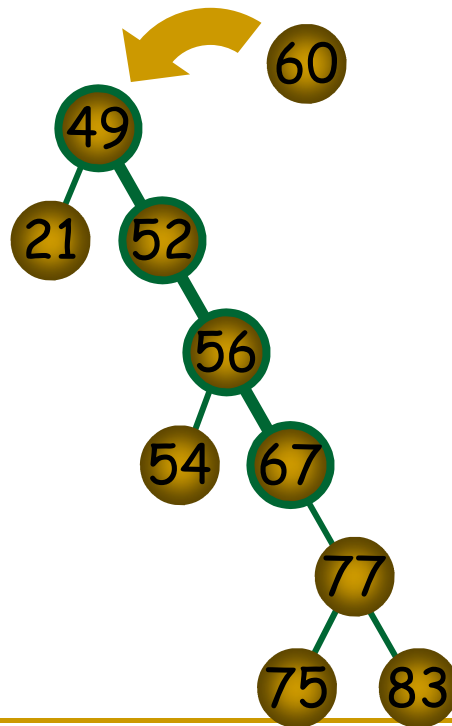
# Los algoritmos para ABB

- vacío()
  - nil
- agregar(A,c)
  - If nil?(A) then bin(c,nil,nil) else
    - if  $c < \text{raíz}(A)$  then
      - bin(raíz(A),agregar(izq(A),c),der(A))
      - else bin(raíz(A),izq(A), agregar(der(A),c))



# Los algoritmos para ABB

- O sea:
  - Buscar al padre del nodo a insertar
  - Insertarlo como hijo de ese padre



---

# Los algoritmos para ABB

- Costo de la inserción:
    - Depende de la distancia del nodo a la raíz
  - En el peor caso
    - $O(n)$
  - En el caso promedio (suponiendo una distribución uniforme de las claves):
    - $O(\lg n)$
-

---

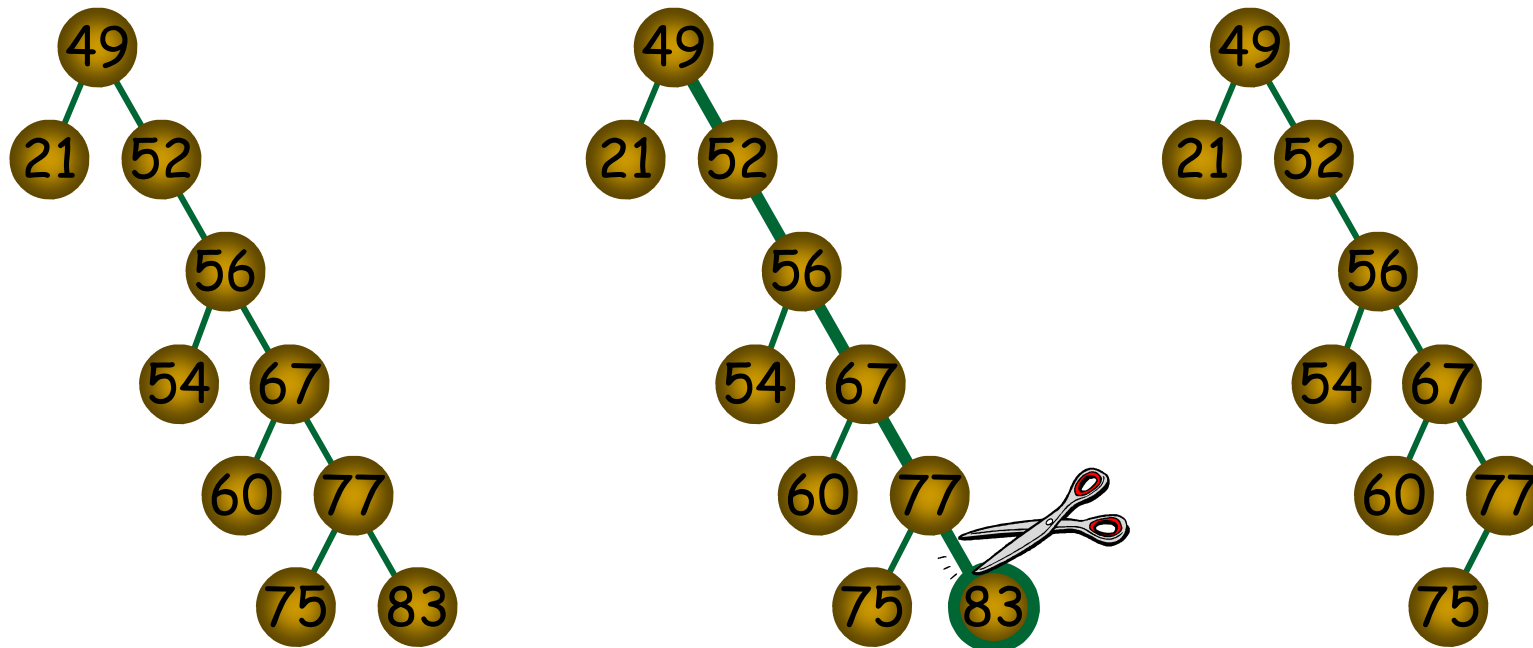
# Los algoritmos para ABB: borrado

- **Borrar( $u, A$ )**
  - **Tres casos**
    1.  $u$  es una hoja
    2.  $u$  tiene un solo hijo
    3.  $u$  tiene dos hijos
  - **Vamos a ver la idea, Uds. la pueden formalizar luego**
-

# Borrado en ABB

## 1. Borrar una hoja

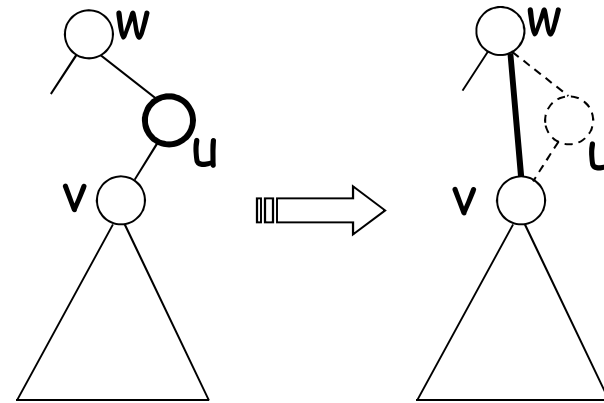
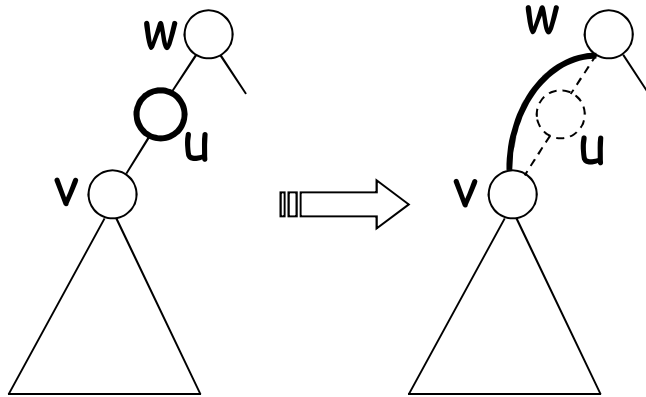
- ❑ Buscar al padre
- ❑ Eliminar la hoja



# Borrado en ABB

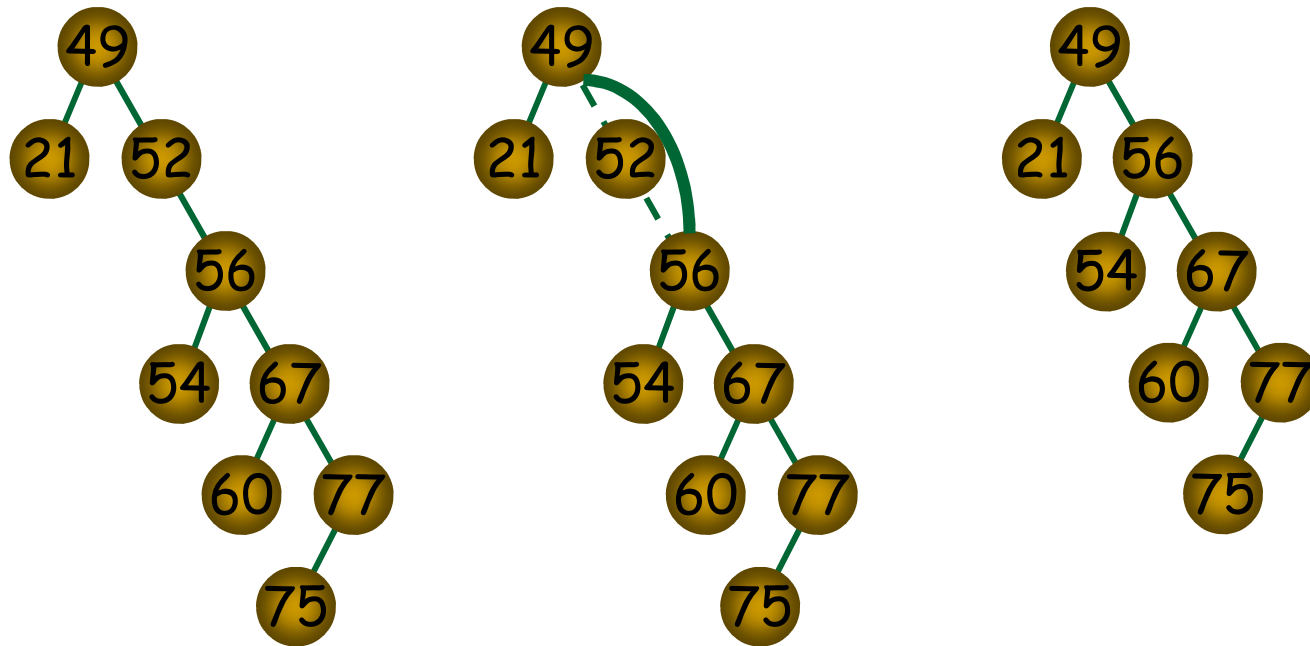
## 2. Borrar un nodo $u$ con un solo hijo $v$

- ❑ Buscar al padre  $w$  de  $u$
- ❑ Si existe  $w$ , reemplazar la conexión  $(w,u)$  con la conexión  $(w,v)$



# Borrado en ABB

## Ejemplo del caso 2



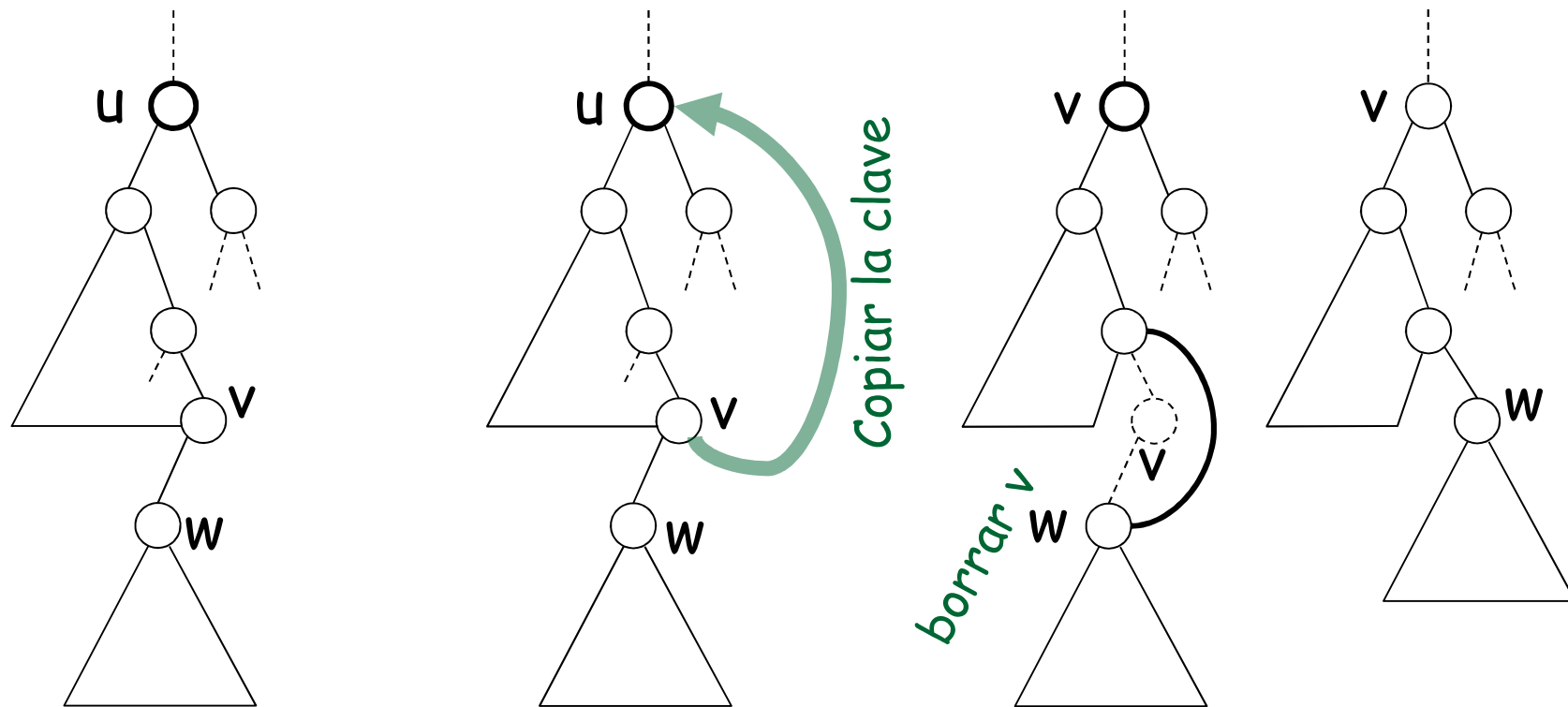
---

# Borrado en ABB

## 3. Borrado de un nodo $u$ con dos hijos

- Encontrar el “predecesor inmediato”  $v$  (o sucesor inmediato) de  $u$ 
    - $v$  no puede tener dos hijos, en caso contrario no sería el predecesor inmediato (sucesor)
  - copiar la clave de  $v$  en lugar de la de  $u$
  - Borrar el nodo  $v$ 
    - $v$  es hoja, o bien tiene un solo hijo, lo que nos lleva los casos anteriores
-

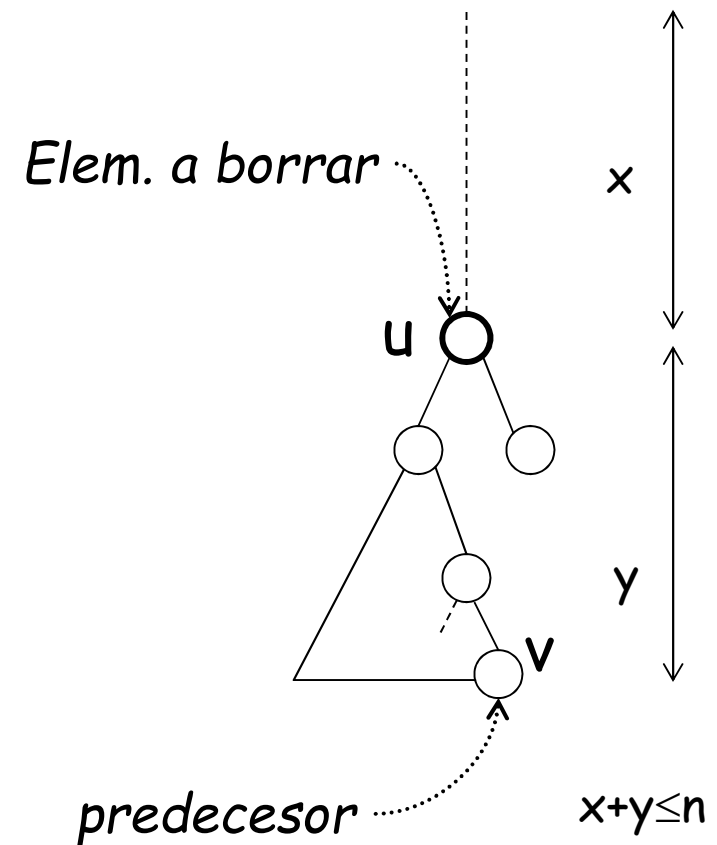
# Borrado en ABB





# Costo del borrado en un ABB

- El borrado de un nodo interno requiere encontrar al nodo que hay que borrar y a su predecesor inmediato
- En el caso peor ambos costos son lineales:  
 $O(n) + O(n) = O(n)$



---

# Representación de conjuntos y diccionarios a través de AVL

- Todas las representaciones vistas hasta ahora tienen al menos una operación de costo lineal en función de la cantidad de elementos
- En muchos casos, eso puede ser inaceptable
- ¿Habr  estructuras m s eficientes?

