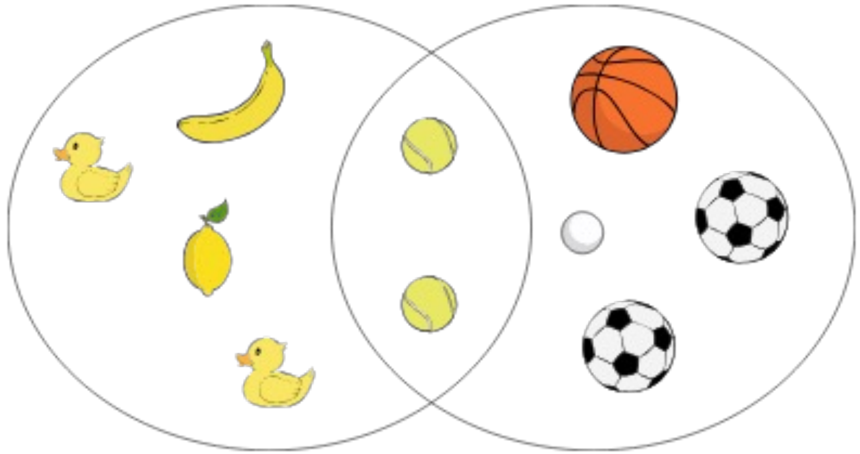
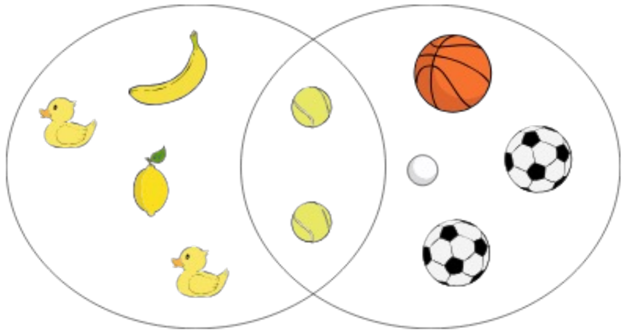
Diseño de Conjuntos y Diccionarios

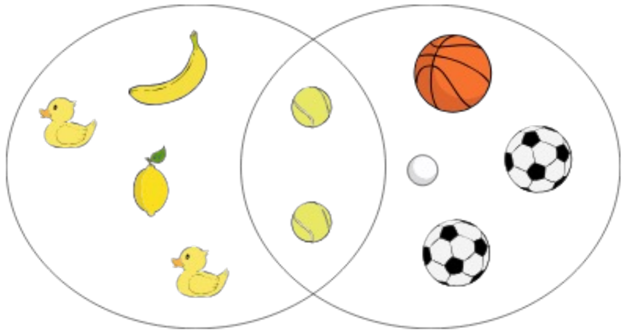


Definición del TAD Conjunto



```
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 Diccionarios?

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

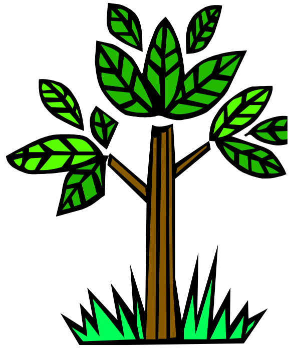
Conjuntos y diccionarios pueden representarse a través de arrays (con o sin repetidos, ordenados o desordenados).

- Ya vimos varias de esas soluciones.

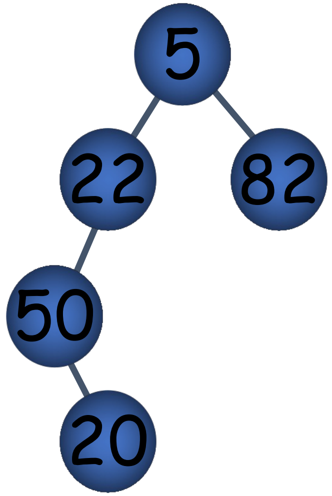
Intenten hacer Uds. mismos el ejercicio de escribir INV, ABS, y los algoritmos

- Complejidad de las operaciones: depende de la implementación, pero
  - Tiempo: alguna de las operaciones requiere O(n) en el peor caso
  - Espacio: O(n).
  - ¿se podrá hacer mejor?

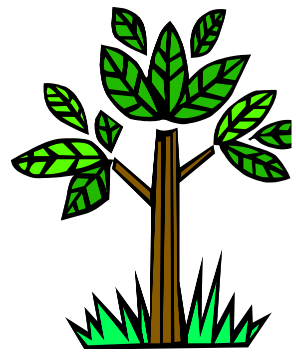
Árboles/Árboles Binarios



- 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]>]>

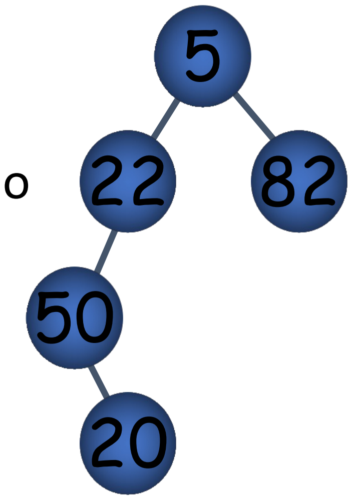


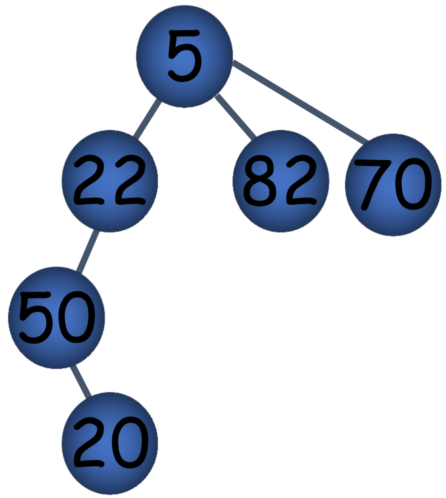
Árboles/Árboles Binarios



- 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

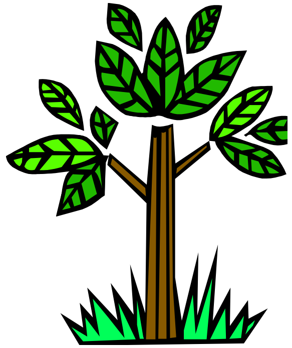
binario





no binario

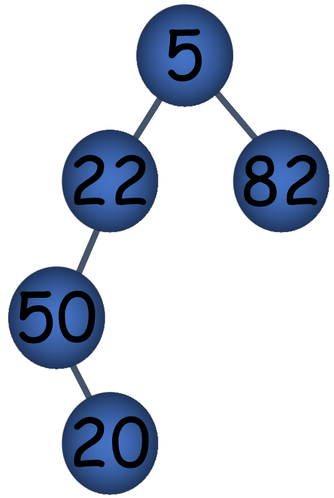
Árboles/Árboles Binarios



El concepto matemático árbol tiene muchos usos, propiedades y funciones muy conocidas.

Por ejemplo, dado un árbol a, podemos hablar de *vacio?(a), raiz(a), altura(a), elementos(a), está(e,a)* y muchas más.

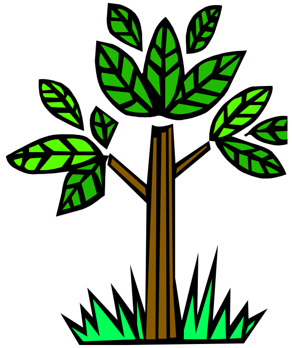
Y para árboles binarios, también *izq(a) y der(a)*



Esas funciones se puede definir recursivamente (por ejemplo en árbol binario)

```
- altura(nil)=0
- altura(<n,i,d>)=1+max{altura(i),altura(d)}
- elementos(nil) = []
- elementos(<n,i,d>) = [n] ++ elementos(i) ++ elementos(d)
```

Árboles/Árboles Binarios



Podemos representar Árboles binarios directamente con punteros:

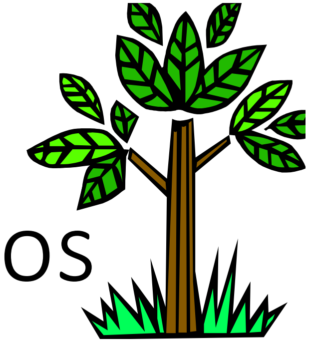
```
Nodo = Struct <dato: N, izq: Nodo, der: Nodo>
```

o opcionalmente…

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

Módulo AB implementa ÁrbolBinario {
  var raíz: Nodo
 }
```

Podriamos definir un TAD ArbolBinario (si queremos) y representarlo con la estructura AB o directamente usar AB para implementar conjuntos.

Representación de conjuntos y diccionarios a través de Arboles Binarios



¿Podríamos representar conjuntos o diccionarios a través de árboles binarios?

- Si!
- ¿Ganaríamos algo? No demasiado en principio ¿no?

Pero.....

Arboles Binarios de Búsqueda (ABB)

Que es un árbol binario de busqueda?

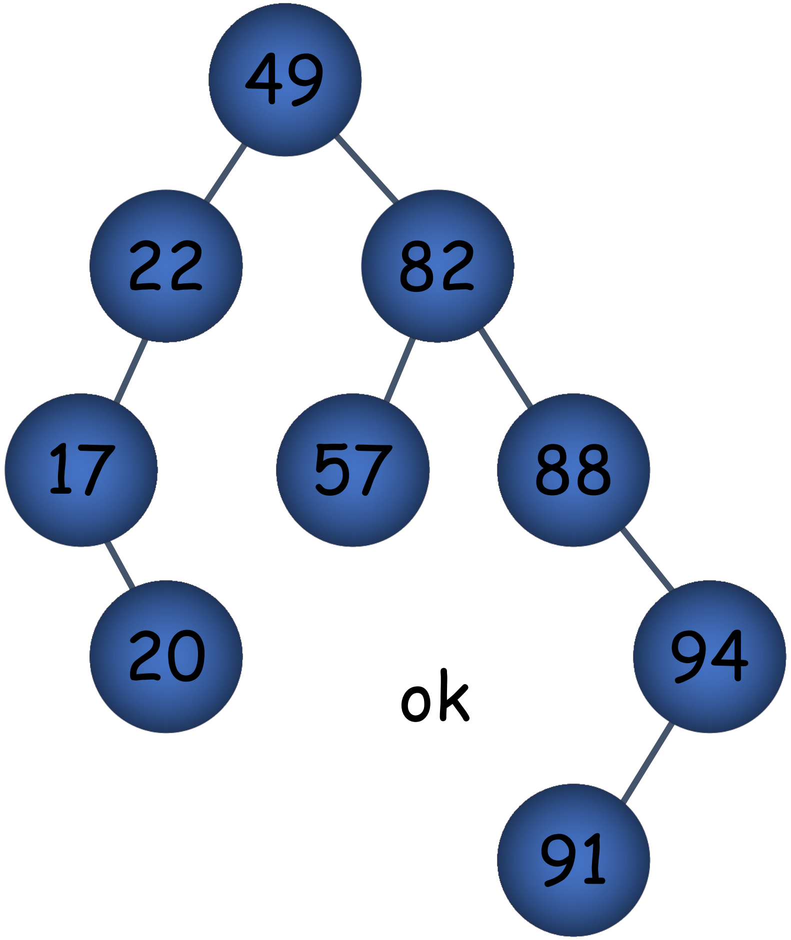
Es un árbol binario que satisfice la siguiente propiedad:

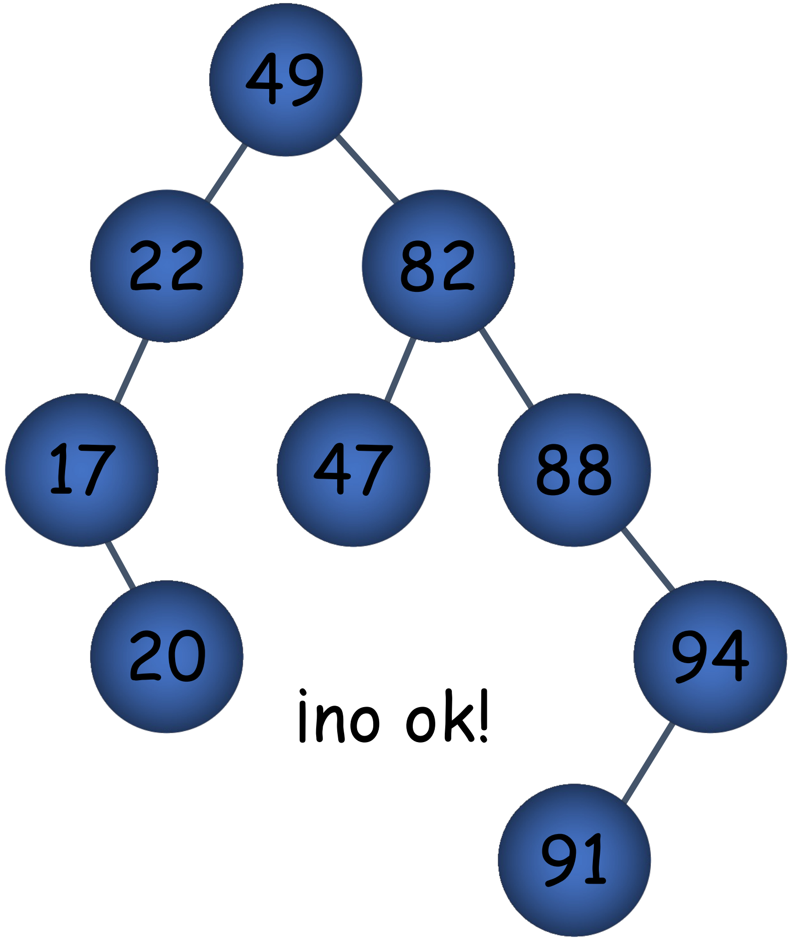
- Para todo nodo, los valores de su subarbol **izquierdo** son **menores** que el valor del nodo y los valores del subarbol **derecho** son **mayores**

O sea, un arbol es ABB si los elementos del subarbolos izquierdoo son menores a la raíz, y los elementos del subarbol derecho son mayores a la raíz.

esABB(a): esArbolBin(a) && esABBN(a.raiz)
esABBN(r) : r = null || (∀x) x in elementos(r.izq) => x<=r.dato && (∀x) x in elementos(r.der) => x>r.dato && esABBN(r.izq) && esABBN(r.der)

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)

      {esABB(e)=TRUE}
```

esABB(a) = esArbolBin(a) && esABBN(a.raiz)
esABBN(r) = r = null || (∀x) x in elementos(r.izq) => x<=r.dato && (∀x) x in elementos(r.der) => x>r.dato && esABBN(r.izq) && esABBN(r.der)

Y la función de abstracción?

```
FuncAbs(a:AB): Conjunto c|
  c.elems = { n:N | n in elementos(a.raiz) }
```

elementos(r) = if r = null then {} else {r.dato} U elementos(r.izq) U elementos(r.der)

Algoritmos para ABB

- Vacio
- Búsqueda
- Inserción
- Eliminar

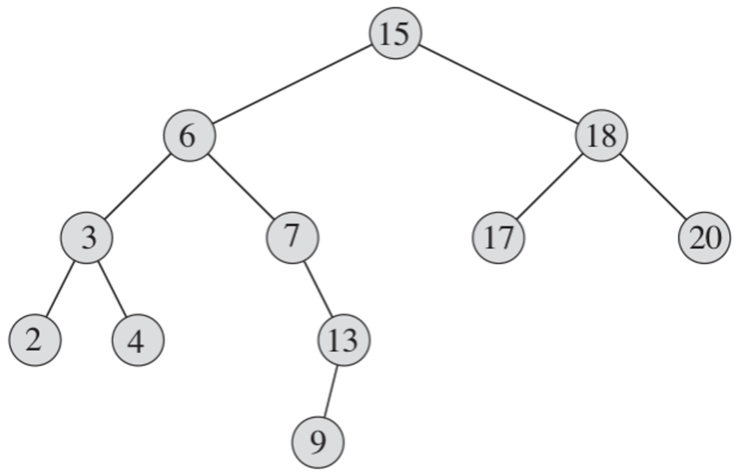
```
Nodo = Struct <dato: N, izq: Nodo, der: Nodo>
o opcionalmente…
Nodo = Struct <dato: N, izq: Nodo, der: Nodo, padre:Nodo>

Módulo AB implementa Conjunto {
  var raíz: Nodo
}
```

Los algoritmos para ABB

```
impl vacío(): ABB {
   a = new ABB;
   a.raíz = null;
   return a;
}
```

Búsqueda (search)

```
impl busqueda(a: ABB, k:int):bool {
 return busqueda(a.raiz, k) !=null
}
impl busqueda(n: Nodo, k:int): nodo {
  if n == null || k = n.dato
     return n
  if k < n.dato then busqueda(n.izq, k)
  else busqueda(n.der, k)
}
```



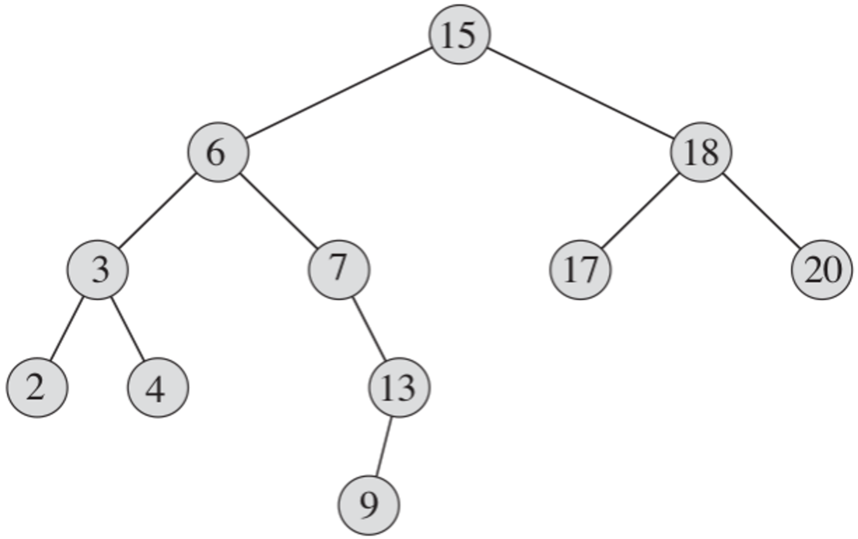
Corrección: devuelve True sii k esta en el árbol

Hip: si k<n.dato ➔ k **más chico** que elementos(n.der) → busca izq
si k>n.dato ➔ k **más grande** que elementos(n.izq) → busca der

Complejidad: O(h), con h la altura del árbol.

Busqueda iterativa

```
busqueda(n:Nodo, k:int):
  while n!=null || k != n.dato
   if k < n.dato then n = n.izq
   else n = n.der
  return n;
```

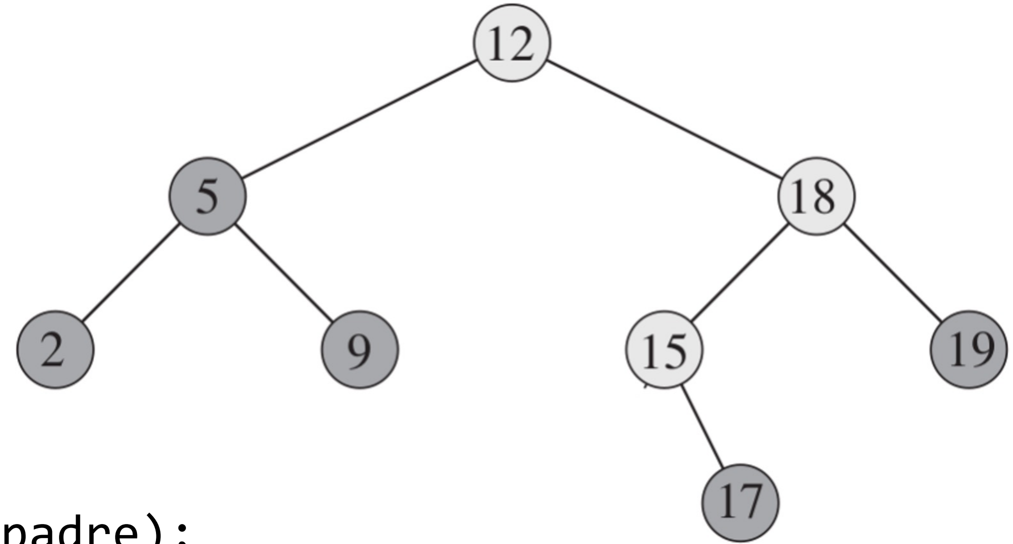


Hipótesis de la recursion similar a la búsqueda binaria:

- Si k<n.dato, k menor que elementos(n.der), buscar a la izquierda
- Si k>n.dato, k mayor que elementos(n.izq), buscar a la derecha

Los algoritmos para ABB

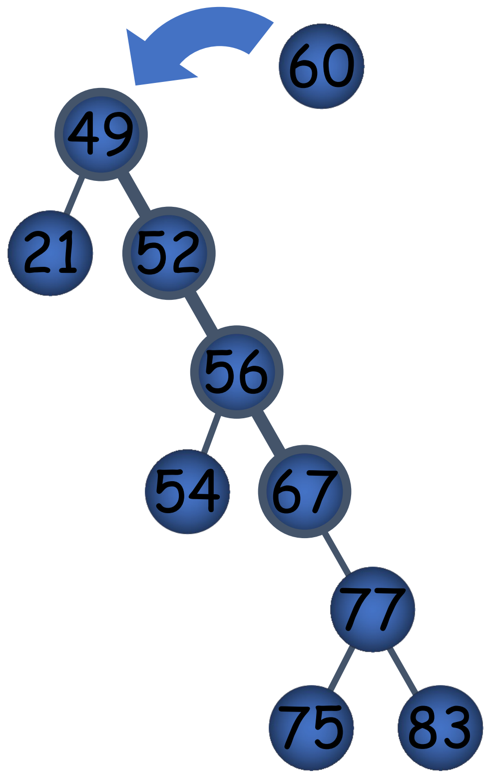
insertar(a,13)




```
impl insertar(inout ABB a, k: int){
 n = a.raíz; padre = null;
 while a!=null
    padre = n;
    if k < n.dato
    then n = n.izq
    else n = n.der
 newnodo = new nodo(k, nil, nil, padre);
 if padre == null
    then a.raíz = newnodo
    else if k<prev.dato then padre.izq = newnodo
         else padre.der = newnodo;
}
```

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 raiz

En el peor caso: O(n)

En el caso promedio (suponiendo una distribución uniforme de las claves): O(lg n)

Los algoritmos para ABB: eliminar

eliminar(u,A) (asumiendo que u está)

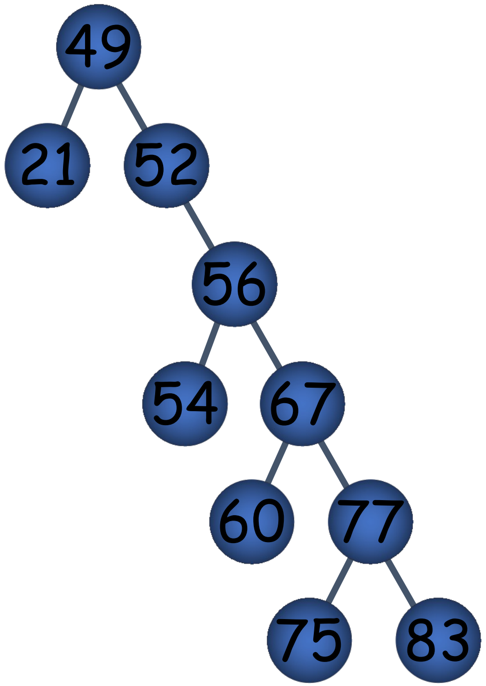
Tres casos

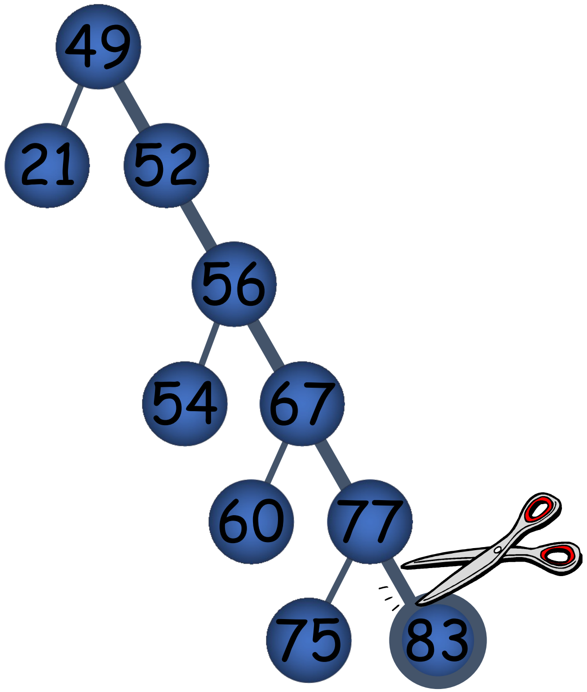
1. u es una hoja
2. u tiene un solo hijo
3. u tiene dos hijos

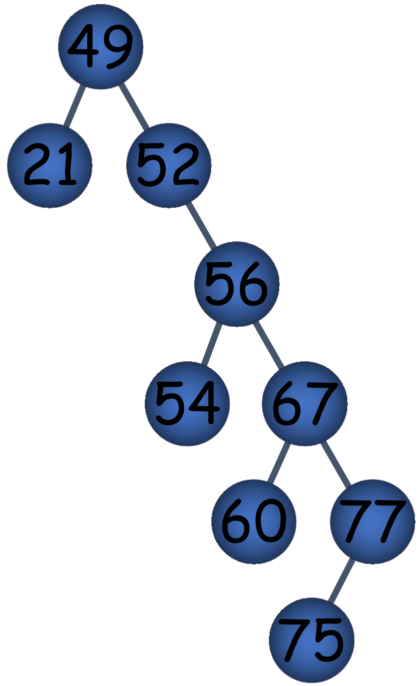
Vamos a ver la idea, la van a implementar en el taller

Eliminar en ABB

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



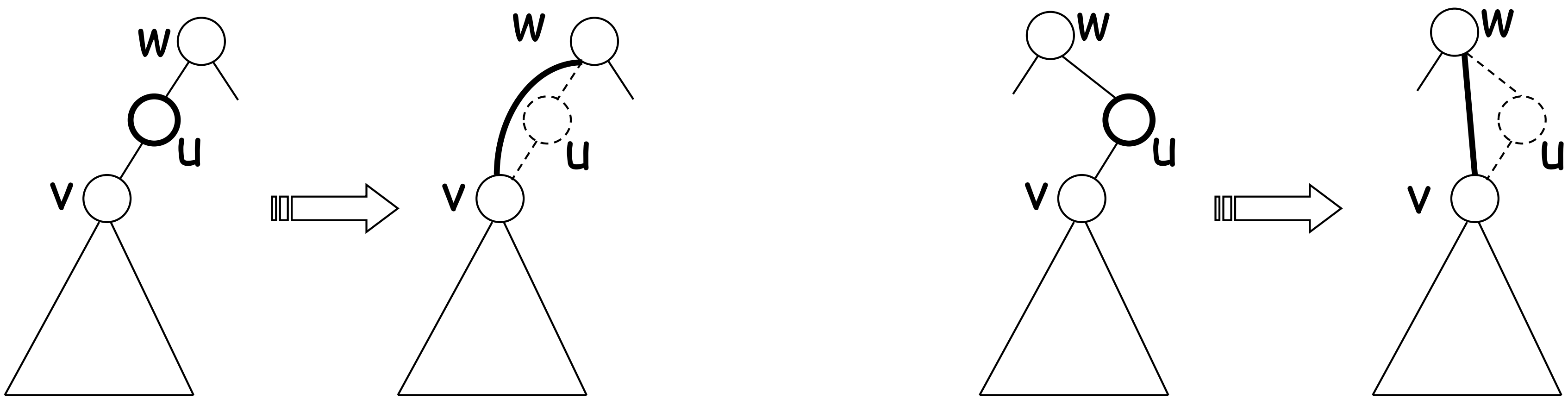




Eliminar en ABB

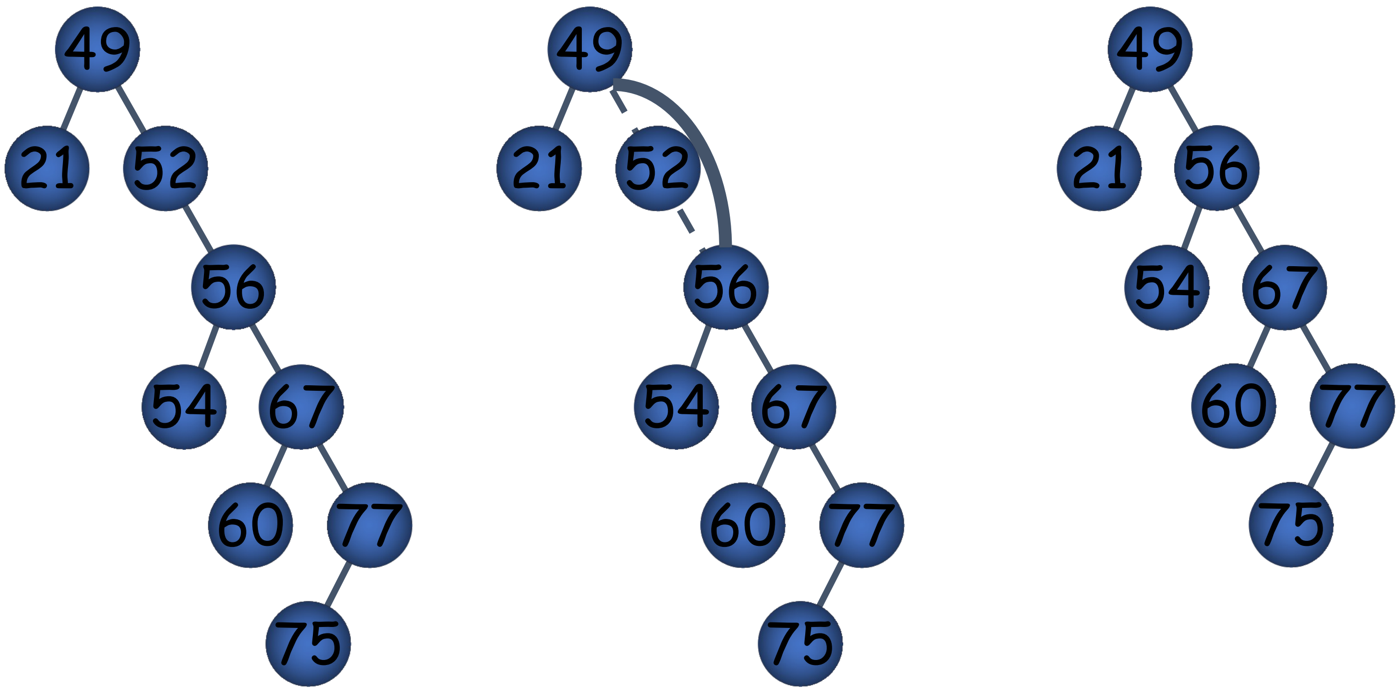
2. Eliminar un nodo u con un solo hijo v

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



Eliminar en ABB

Ejemplo del caso 2



Borrado en ABB

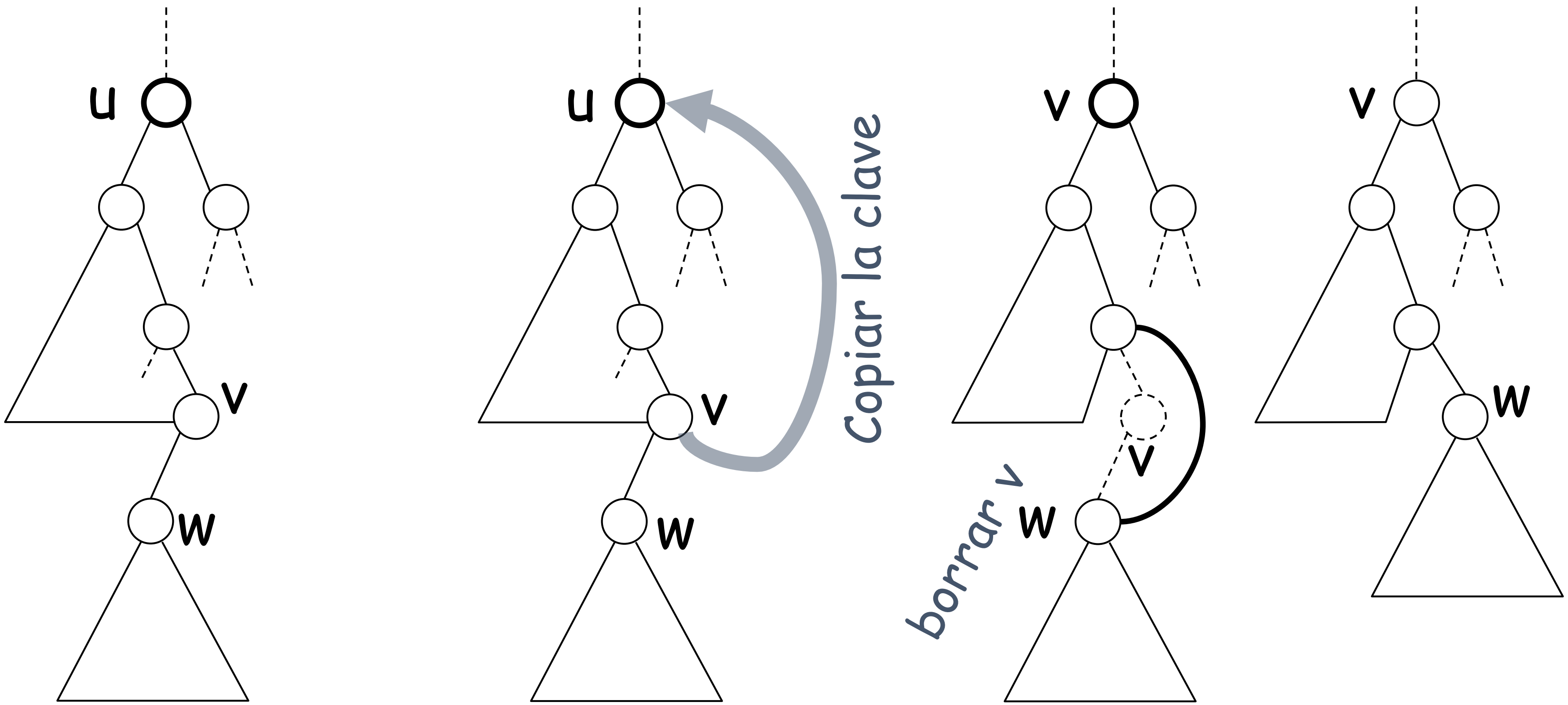
3. Borrado de un nodo u con dos hijos

Encontrar el “predecesor inmediato” v de u

- v **no puede** tener dos hijos, en caso contrario no sería el predecesor inmediato
- 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

Podemos aplicar la misma idea con sucesor inmediato

Eliminar en ABB



Eliminación (resumen)

Eliminación de un nodo u de un árbol de búsqueda binaria T:

Primero obtenemos el nodo u (buscamos u usando la clave):

1. Si u**no tiene hijos**, simplemente lo eliminamos modificando su padre para reemplazar u con NIL como su hijo.

2. Si u tiene **un solo hijo** (subarbol v), entonces elevamos v para que ocupe la posición de u en el árbol.

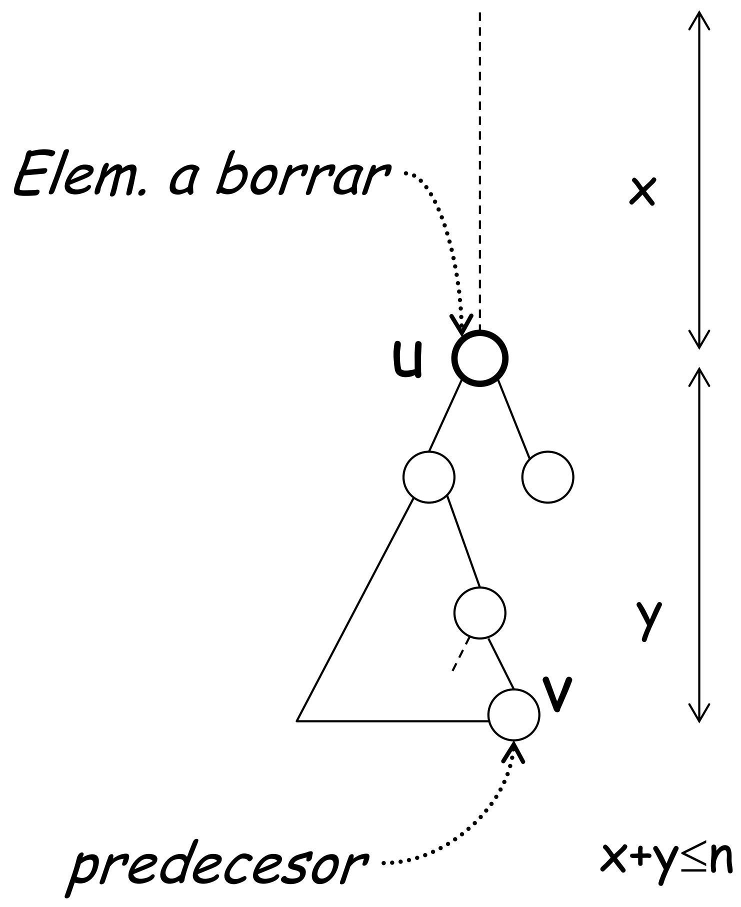
3. Si u tiene **dos hijos,** hallamos el predecesor de u (pred), que debe estar en el subárbol **izquierdo** de u, y hacemos que pred tome la posición de u en el árbol. (idem con sucesor)
- Ahora solo temenos que borrar el nodo en la posicion pred (es como un caso 2)

Costo de la eliminación en un ABB

La eliminación de un nodo interno requiere encontrar al nodo que hay que borrar y 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 linear en función de la cantidad de elementos
- En muchos casos, eso puede ser inaceptable
- ¿Habrá estructuras más eficientes?