# Algoritmos y Estructuras de Datos

## Árboles binarios

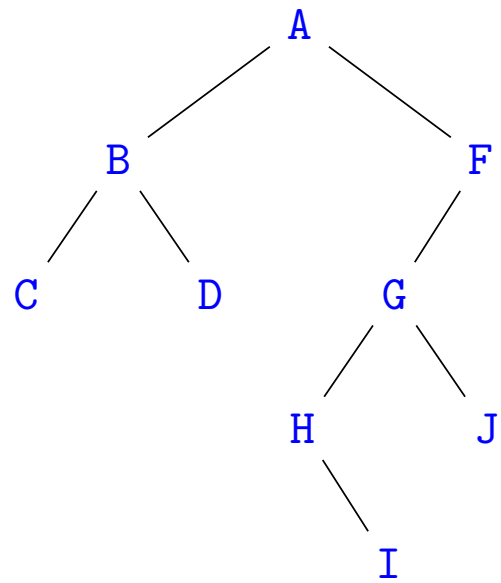
## Búsqueda y balanceo

Árboles binarios

Árboles de búsqueda

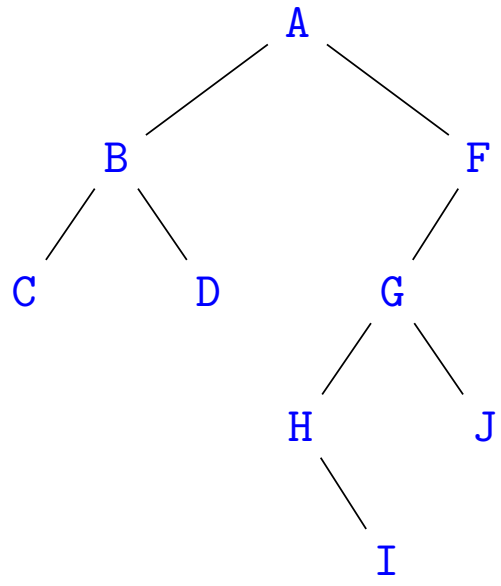
Árboles balanceados

# Árboles binarios: nomenclatura




1. La **raíz** del árbol es A.
2. El nodo G es el **padre** de los nodos H, J.
3. El nodo H es el **hijo izquierdo** de G.
4. El nodo J es el **hijo derecho** de G.
5. El nodo F tiene un solo hijo.
6. Las **hojas** del árbol son los nodos sin hijos (C, D, I, J).
7. Los nodos B y F son **hermanos**.

# Árboles binarios: nomenclatura



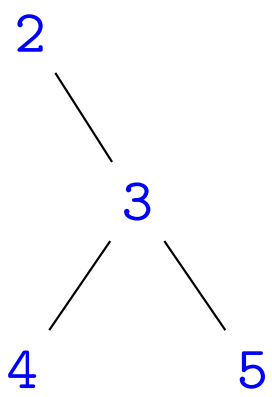

8. El nodo `F` es **ancestro** de los nodos `G`, `H`, `I`, `J`.
9. Los nodos `G`, `H`, `I`, `J` son **descendientes** de `F`.
10. Un **subárbol** consta de un nodo y todos sus descendientes.
11. Una **rama** es un camino desde la raíz hasta una hoja.
    Por ejemplo, `A` $\rightarrow$ `F` $\rightarrow$ `G` $\rightarrow$ `J`.
12. El **nivel** de un nodo es la distancia desde la raíz.
13. La **altura** es el número de nodos en la rama más larga.
    En este caso, la altura es 5.

# Árboles binarios en Python

Usamos `None` para representar un árbol vacío (sin nodos).

```
class Nodo:
    def __init__(self, left, value, right):
        self.left = left
        self.value = value
        self.right = right
```

## Ejemplo




```
Nodo(
    None,
    2,
    Nodo(
        Nodo(None, 4, None),
        3,
        Nodo(None, 5, None),
    )
)
```

# Algoritmos básicos

## Suma de todos los nodos de un árbol binario de enteros

```
def suma(a):
    if a is None:
        return 0
    else:
        return a.value + suma(a.left) + suma(a.right)
```

Complejidad: O(n) en peor caso para un árbol de $n$ nodos.

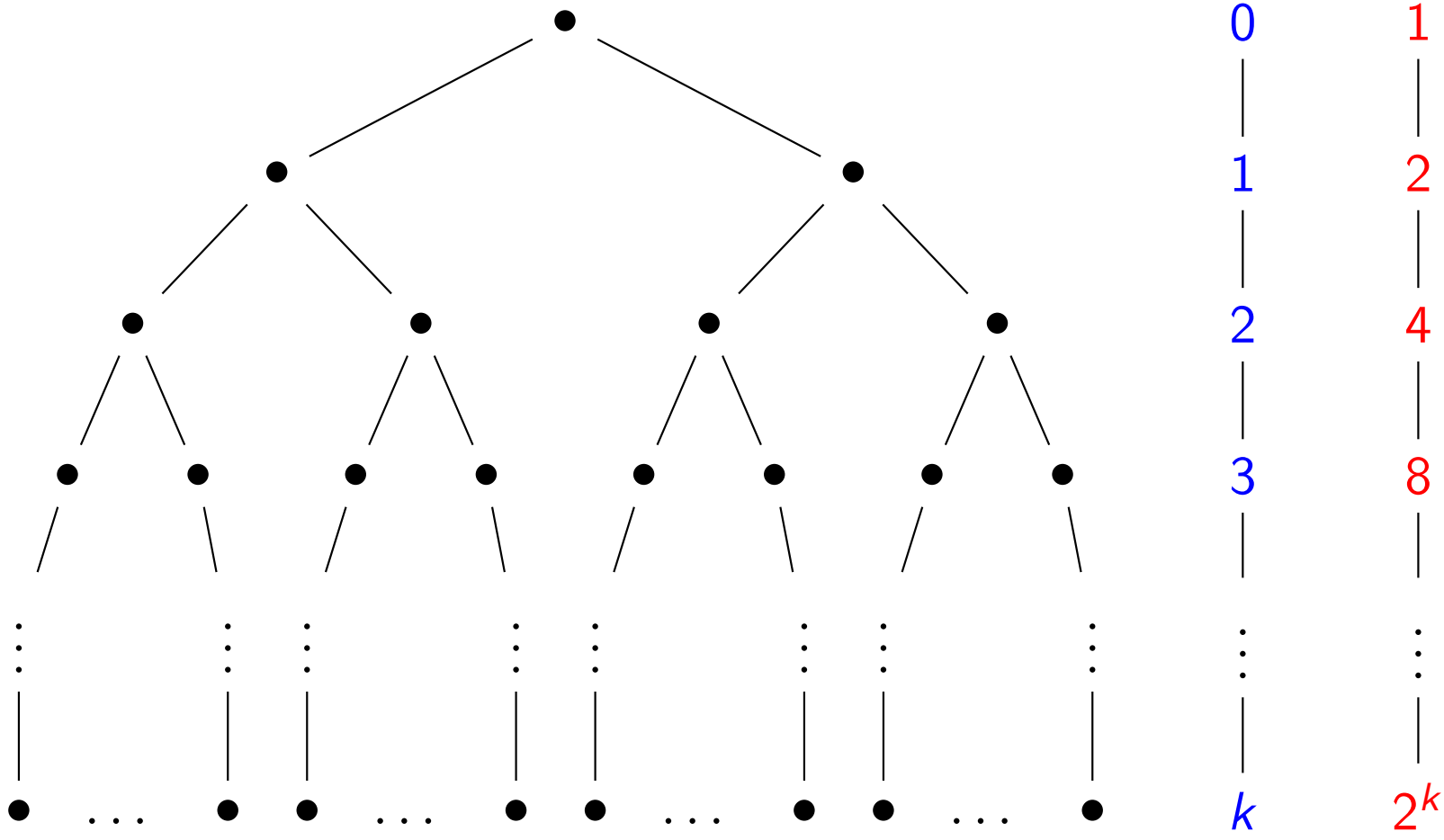
## Altura de un árbol binario

```
def altura(a):
    if a is None:
        return 0
    else:
        return 1 + max(altura(a.left), altura(a.right))
```

Complejidad: O(n) en peor caso para un árbol de $n$ nodos.

# Propiedades básicas

Un AB es **balanceado completo** si no hay nodos con un solo hijo.
El $i$-ésimo nivel tiene $2^i$ nodos:



# Propiedades básicas

## Observación

En un AB balanceado completo:

1. Si la altura es $h$, el número de nodos es $2^h - 1$.

$$\sum_{i=0}^{h-1} 2^i = 2^h - 1$$

2. Si el número de nodos es $n$, la altura es $\log_2(n+1)$.

Árboles binarios

Árboles de búsqueda

Árboles balanceados

# Implementación de conjuntos

Queremos implementar tipo CONJUNTO con la siguiente interfaz:

1. Crear un conjunto vacío. $O(1)$
2. **Insertar** un elemento.
3. **Buscar** si un elemento está en el conjunto.
4. **Eliminar** un elemento.

¿Qué complejidades obtendríamos con las siguientes estructuras?

| | **Insertar** | **Buscar** | **Eliminar** |
|---|---|---|---|
| Arreglo (sin ordenar) | $O(1)$ *(amortizado)* | $O(n)$ | $O(n)$ |
| Arreglo ordenado | $O(n)$ | $O(\log n)$ | $O(n)$ |
| Lista enlazada | $O(1)$ | $O(n)$ | $O(n)$ |

# Árboles binarios de búsqueda

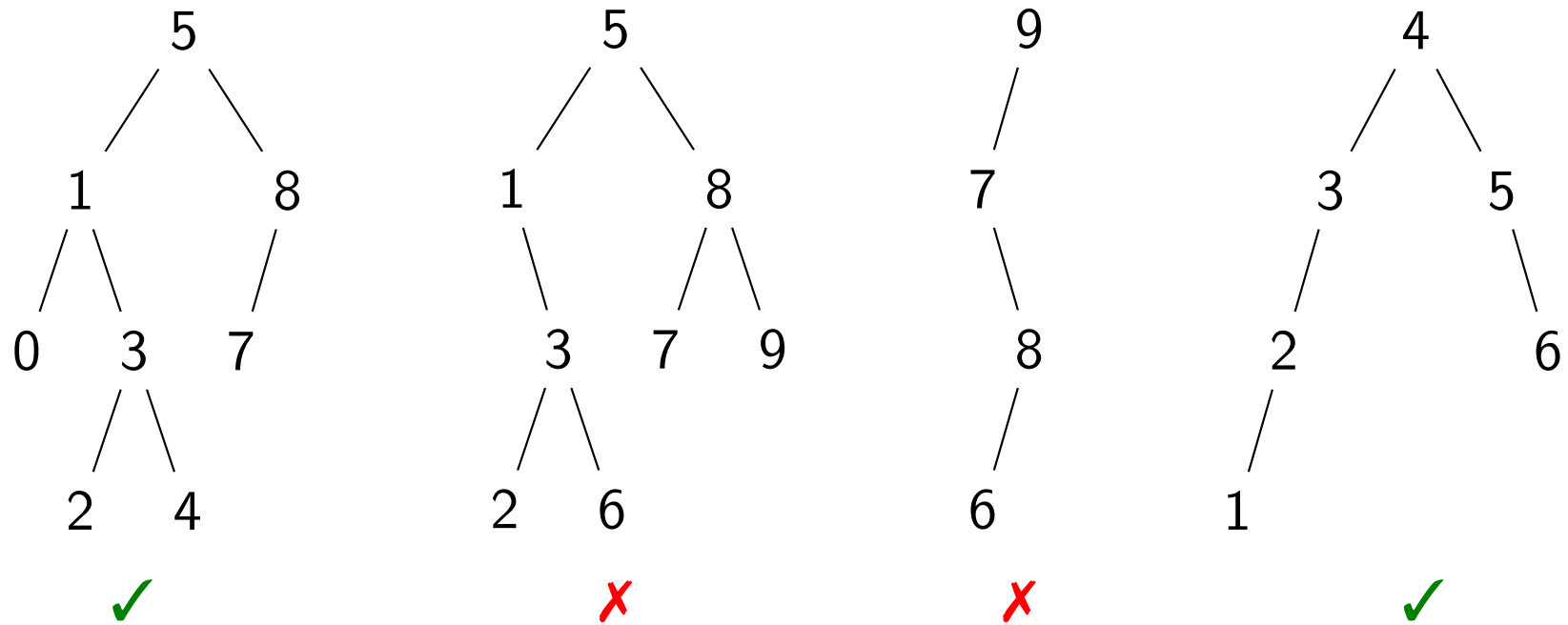
Un **árbol binario de búsqueda** (ABB) es un árbol binario con el siguiente invariante.

## Invariante de ABB

Para todo elemento $x$ en el árbol:

Si $y$ es un descendiente del lado izquierdo, entonces $y < x$.
Si $y$ es un descendiente del lado derecho, entonces $y > x$.

## Ejemplos



# Árboles binarios de búsqueda — búsqueda

## Búsqueda en un ABB

```
def buscar(a, x):
    if a is None:
        return False
    elif x == a.value:
        return True
    elif x < a.value:
        return buscar(a.left, x)
    else:                                   # x > a.value
        return buscar(a.right, x)
```

¿Cuál es la complejidad si el árbol tiene $n$ nodos?

- En peor caso $O(n)$, pero podemos decir algo más preciso.
- Es $O(h)$ en peor caso, donde $h$ es la altura del árbol.
- Si el ABB fuera balanceado completo, $h \in O(\log n)$.

# Árboles binarios de búsqueda — inserción

## Inserción en un ABB

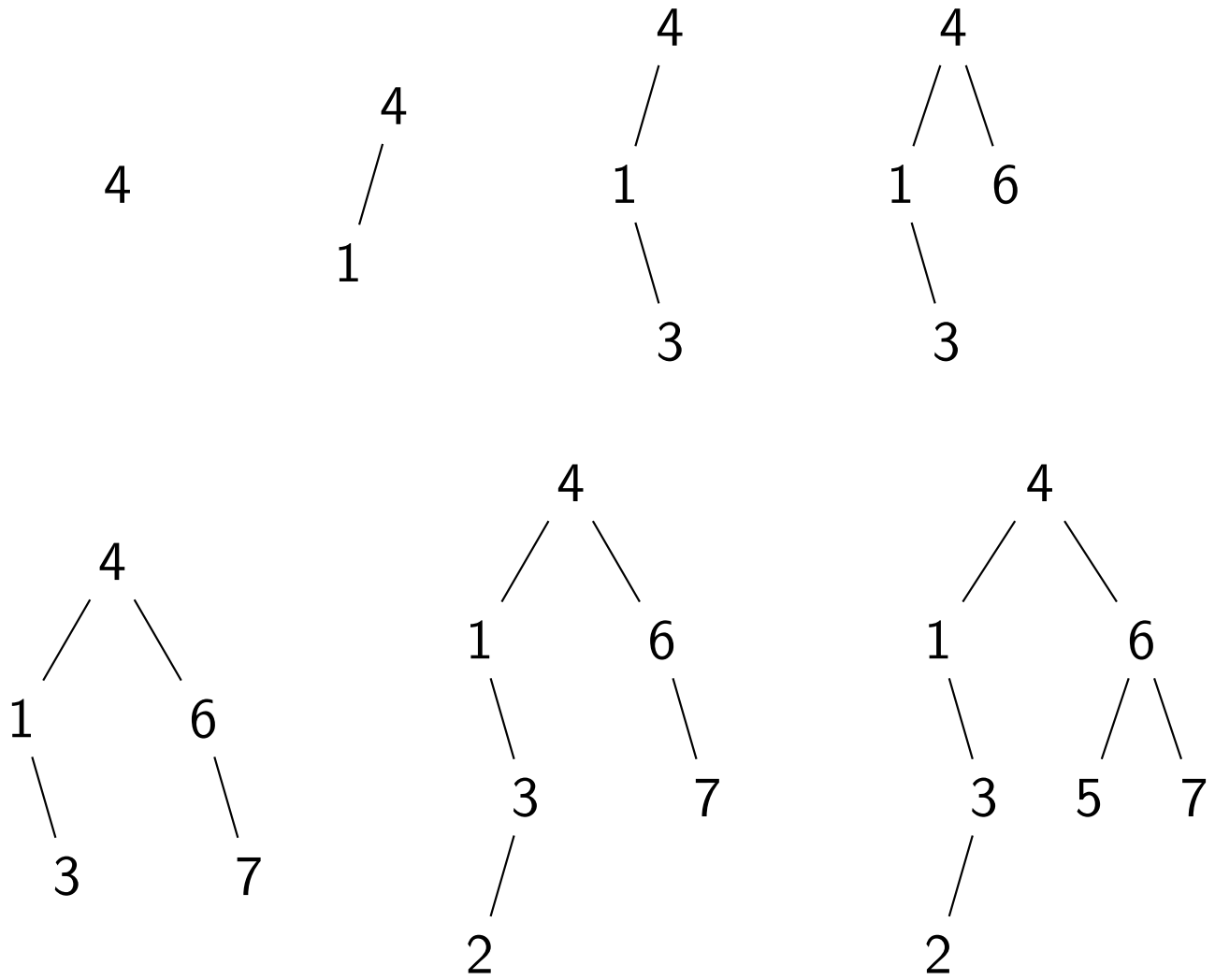
```
def insertar(a, x):
    if a is None:
        return Nodo(None, x, None)
    elif x == a.value:
        return a
    elif x < a.value:
        a.left = insertar(a.left, x)
        return a
    else:                                  # x > a.value
        a.right = insertar(a.right, x)
        return a
```

Complejidad: $O(h)$ en peor caso, donde $h$ es la altura del árbol.

# Árboles binarios de búsqueda — inserción

## Ejemplo

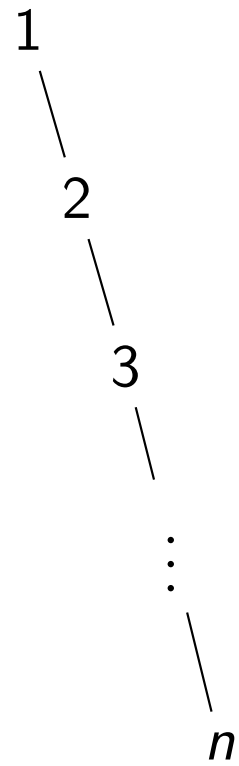
Insertemos: 4, 1, 3, 6, 7, 2, 5.



# Árboles binarios de búsqueda — inserción

## Ejemplo

Insertemos: 1, 2, 3, . . ., $n$.



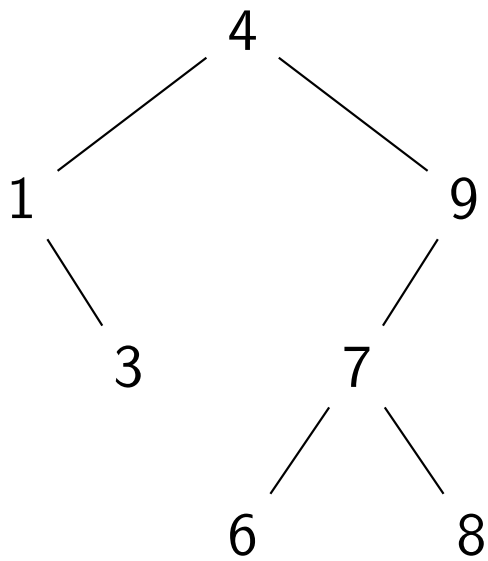

Resulta un árbol degenerado a derecha.

Las operaciones de inserción, búsqueda y eliminación son lineales.

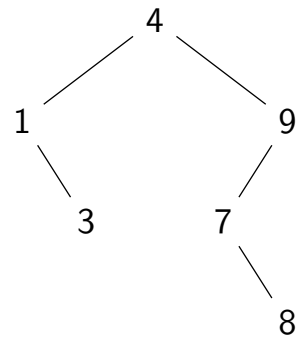
(La estructura es análoga a la de una lista enlazada).

# Árboles binarios de búsqueda — eliminación

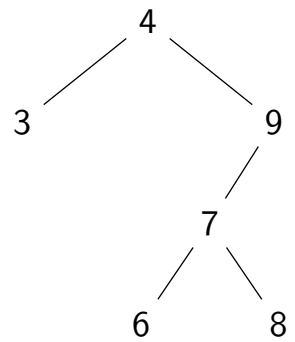
Eliminar una hoja o un nodo con un solo hijo es fácil.



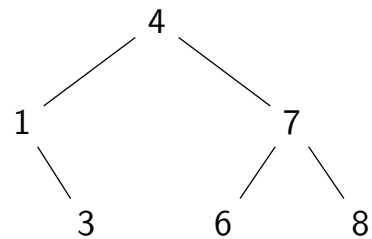

**eliminar(6)** →




**eliminar(1)** →




**eliminar(9)** →

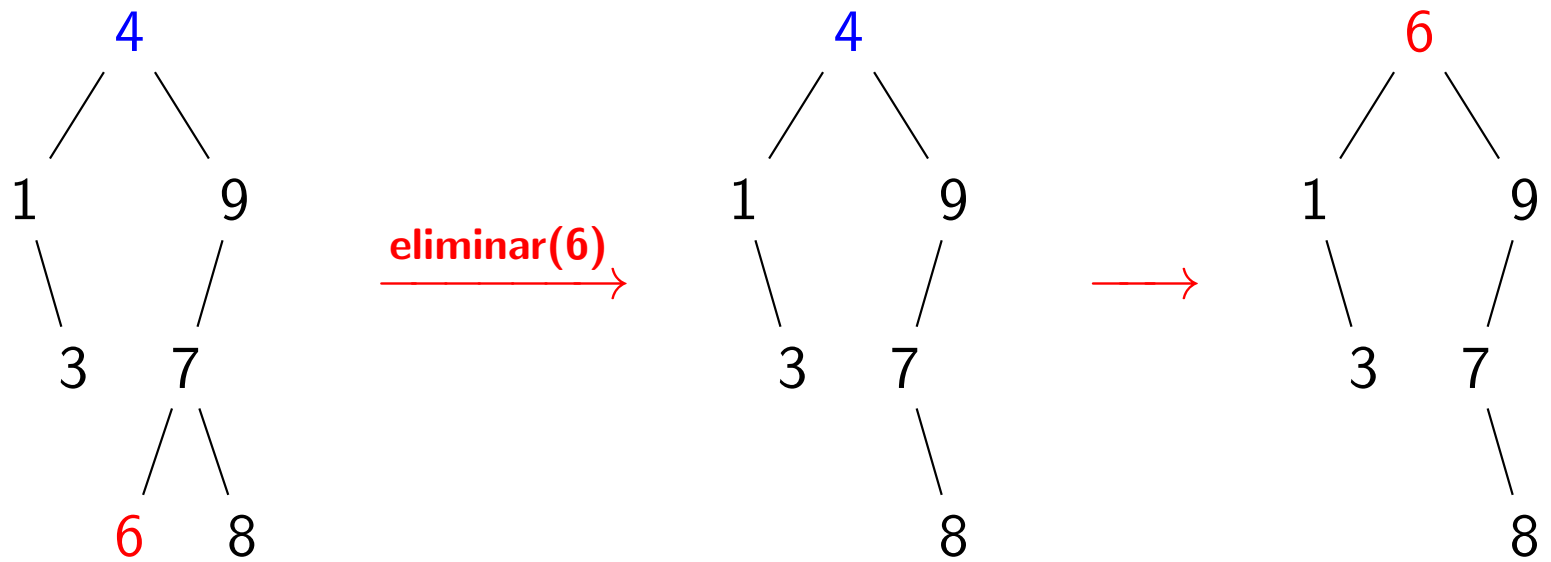


# Árboles binarios de búsqueda — eliminación

Para eliminar un nodo con dos hijos:

1. Eliminamos su **sucesor inmediato**.
   El sucesor inmediato es el **mínimo** del subárbol derecho.
   Se encuentra en la rama más izquierda en el subárbol derecho.
2. Reemplazamos el elemento por su sucesor inmediato.

## Ejemplo: para eliminar el 4



# Árboles binarios de búsqueda — eliminación

## Algoritmo para eliminar el mínimo elemento de un ABB

Entrada: un ABB `a` no vacío.

Salida: un par `(m, a')` donde:

- `m` es el mínimo del árbol original,
- `a'` es el ABB que resulta de eliminar el mínimo de `a`.

```
def eliminarMinimo(a):
    if a.left is None:
        return (a.value, a.right)
    else:
        (m, a.left) = eliminarMinimo(a.left)
        return (m, a)
```

# Árboles binarios de búsqueda — eliminación

## Eliminación de un ABB

```
def eliminar(a, x):
    if a is None:
        return a
    elif x == a.value:
        if a.right is None:
            return a.left
        else:
            (a.value, a.right) = eliminarMinimo(a.right)
            return a
    elif x < a.value:
        a.left = eliminar(a.left, x)
        return a
    else:                                   # x > a.value
        a.right = eliminar(a.right, x)
        return a
```

Complejidad: $O(h)$ en peor caso, donde $h$ es la altura del árbol.

Árboles binarios

Árboles de búsqueda

Árboles balanceados

# Árboles balanceados

Vimos que en un ABB de $n$ nodos y altura $h$:

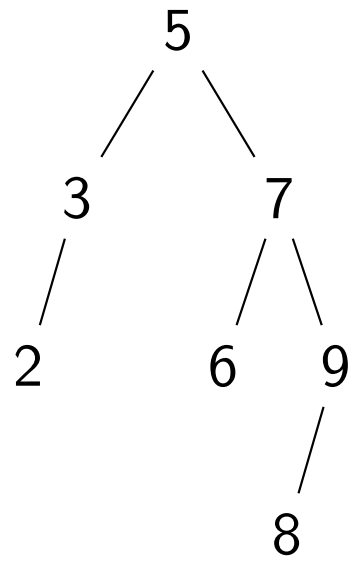
- La búsqueda, la inserción y el borrado son $O(h)$.
- Si el ABB es balanceado, $h \in O(\log n)$.
- Pero en un árbol no balanceado, puede ser que $h \in \Theta(n)$.
- Las operaciones pueden producir árboles no balanceados.

Vamos a refinar la noción de ABB con un invariante de balanceo.

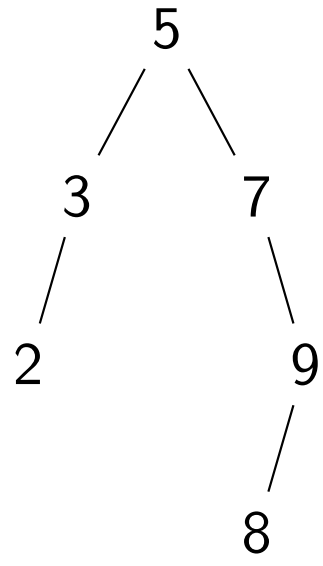
(Hay muchas estructuras; ej. AVLs, Red-Black trees, B-trees, etc.).

# Árboles AVL (Adelson-Velski–Landis, 1962)

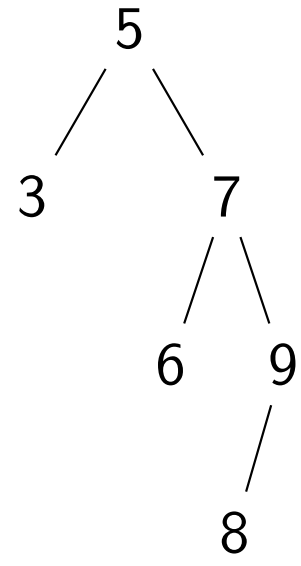
- El **factor de balanceo** de un nodo es la diferencia entre las alturas de sus hijos.
- Si todos los nodos tienen factor de balanceo 0, el árbol es balanceado completo.
- Un **árbol AVL** es un ABB en el cual todos los nodos tienen factor de balanceo 0, 1 ó $-1$.




✓




✗




✗

- En un árbol AVL vale $\boxed{h \in O(\log n)}$.

## Inserción/eliminación en un AVL

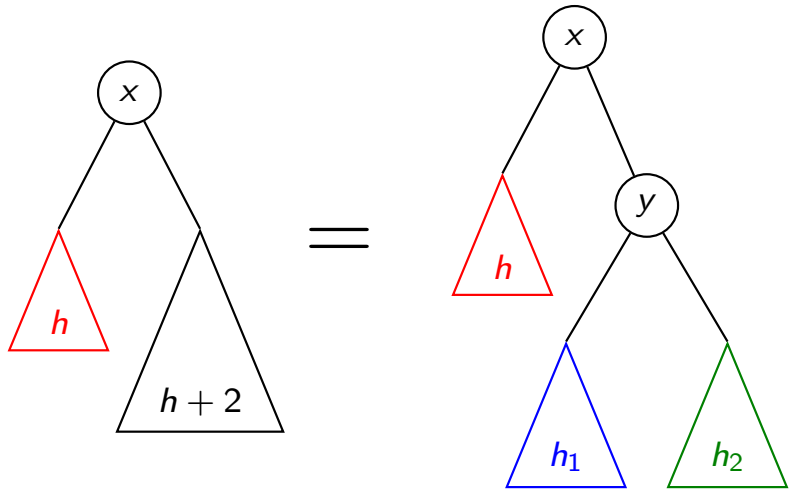
Se usa el mismo método de inserción/eliminación que en un ABB.
El elemento afectado se encuentra en un nodo del árbol.

Se visitan todos los nodos en la rama afectada.
Desde el padre del nodo afectado hasta la raíz:

- Si el factor de balanceo es 0, 1 o $-1$, seguir subiendo.
- Si el factor de balanceo es 2 o $-2$, rebalancear.

# Árboles AVL — rebalanceo

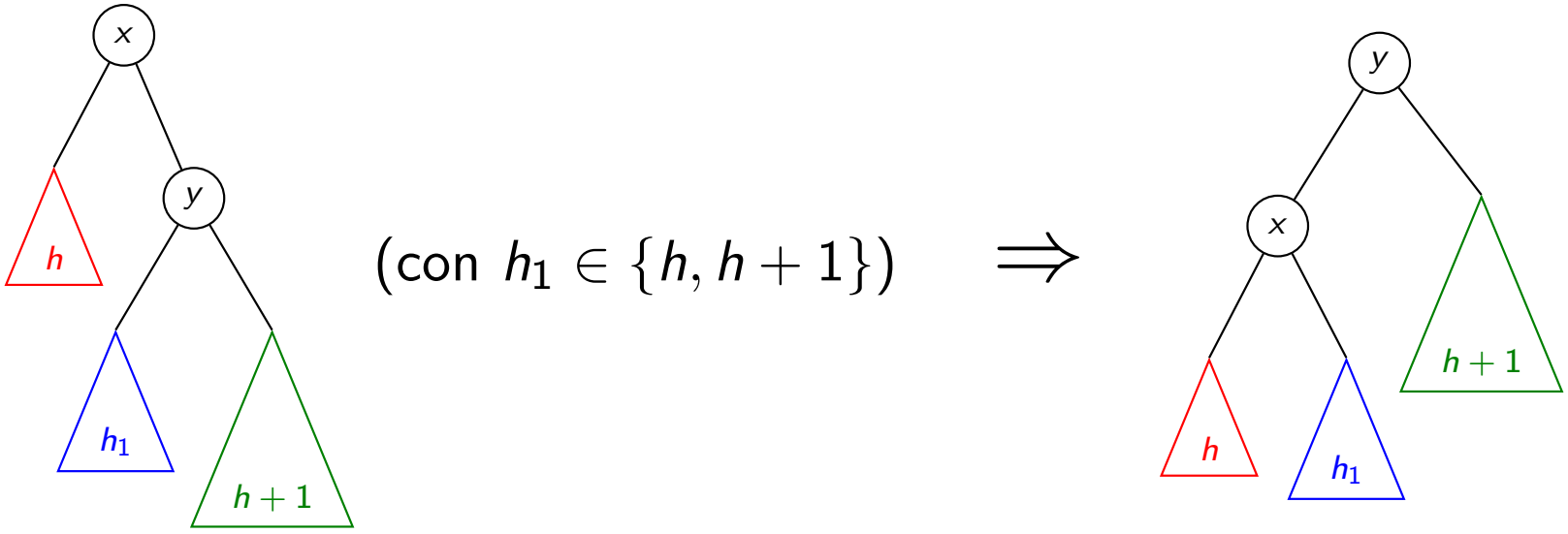
La situación es la siguiente (o la simétrica):




**Valores posibles**

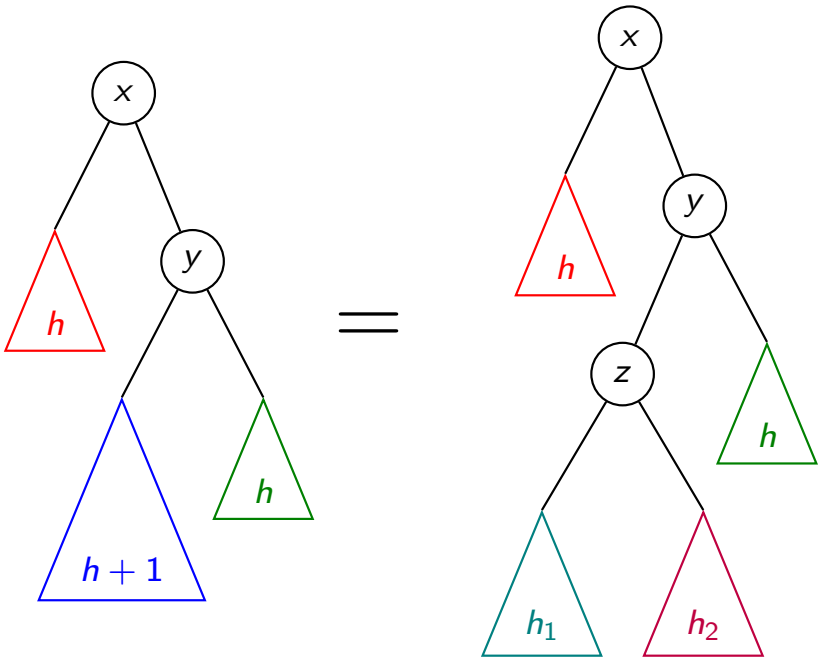
| $h_1$ | $h_2$ |
|---|---|
| $h$ | $h+1$ |
| $h+1$ | $h+1$ |
| $h+1$ | $h$ |

**Caso 1:** el subárbol derecho tiene hijos de altura $h_1$ y $h+1$



# Árboles AVL — rebalanceo

**Caso 2:** el subárbol derecho tiene hijos de altura $h+1$ y $h$




(con $h_1, h_2 \in \{h, h-1\}$)