

# Estructuras de Datos no Lineales

## 1.5. Búsqueda

José Fidel Argudo Argudo  
José Antonio Alonso de la Huerta  
M<sup>a</sup> Teresa García Horcajadas



Versión 3.0

## **1.5. Búsqueda**

**1.5.1. Árboles binarios de búsqueda**

**1.5.2. Árboles binarios de búsqueda equilibrados**

**1.5.3. Árboles B**

**1.5.4. Tablas de dispersión**

# Estructuras de Datos no Lineales

## 1.5.1. Árboles binarios de búsqueda

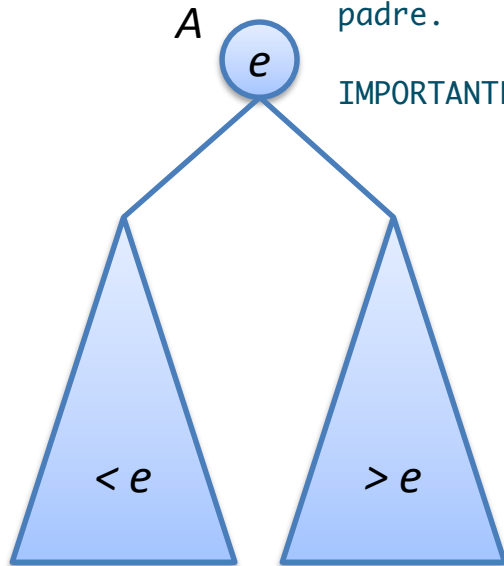
José Fidel Argudo Argudo  
José Antonio Alonso de la Huerta  
M<sup>a</sup> Teresa García Horcajadas



Versión 3.0

# Árboles binarios de búsqueda

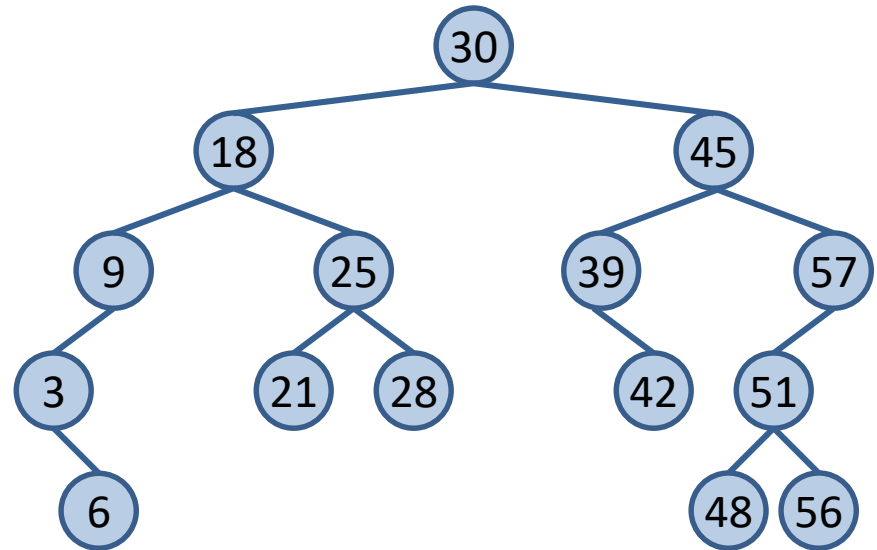
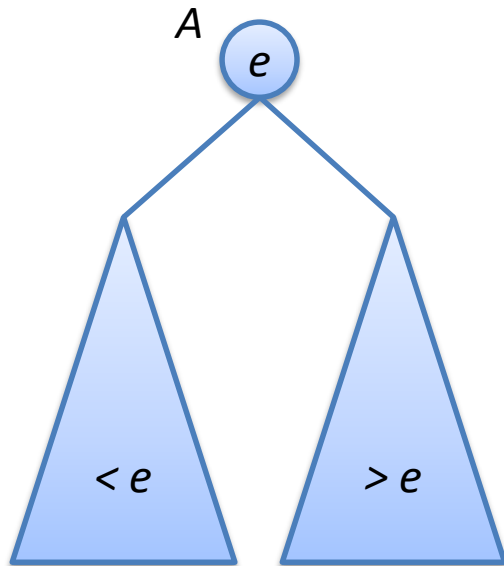
Vemos que el lado izq serán valores menores que el del nodo padre y que los valores del lado drcho son valores mayores que el del nodo padre.



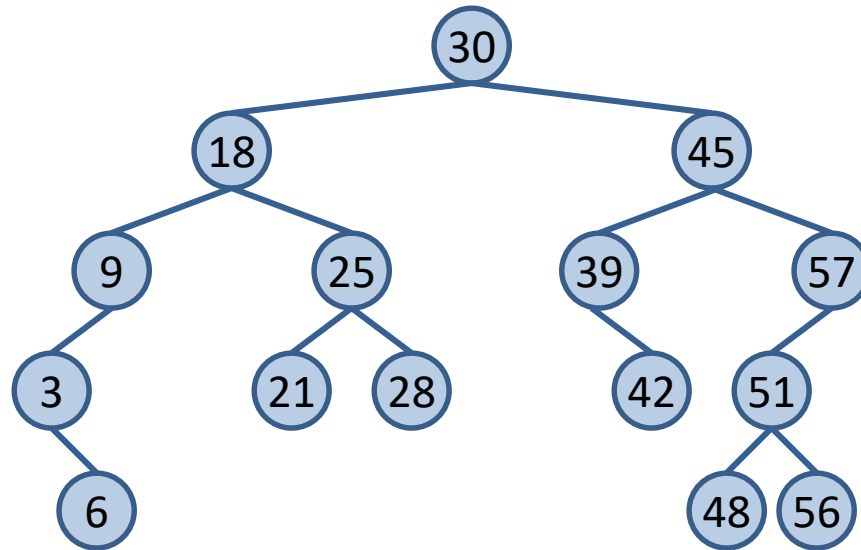
IMPORTANTE: No pueden haber valores repetidos

# Árboles binarios de búsqueda

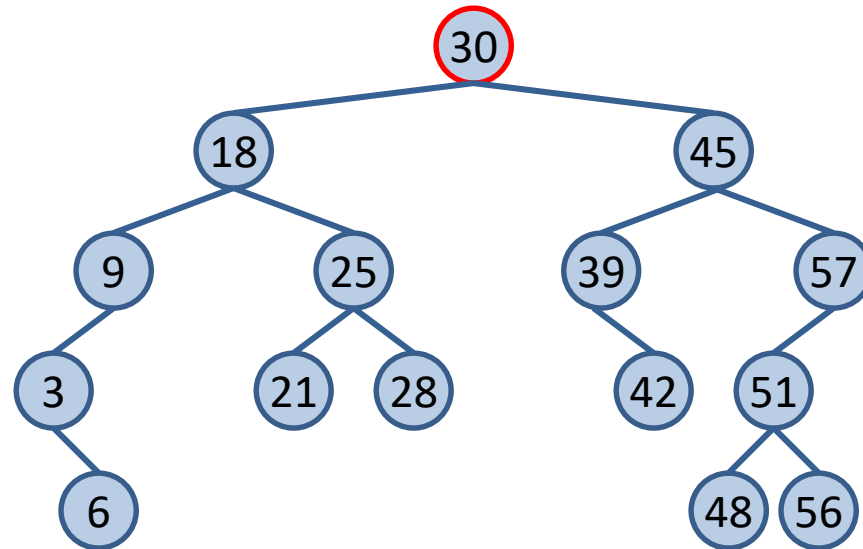
Ejemplo de ABB



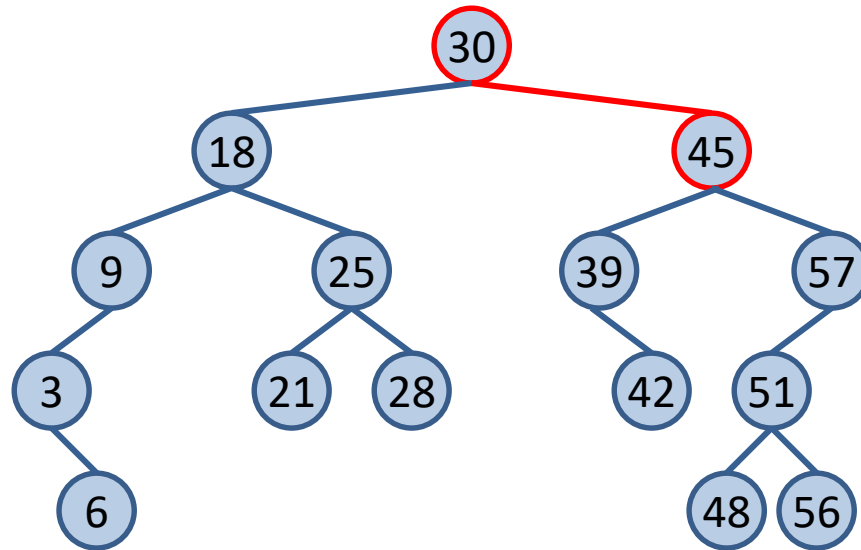
Buscar 51



Buscar 51

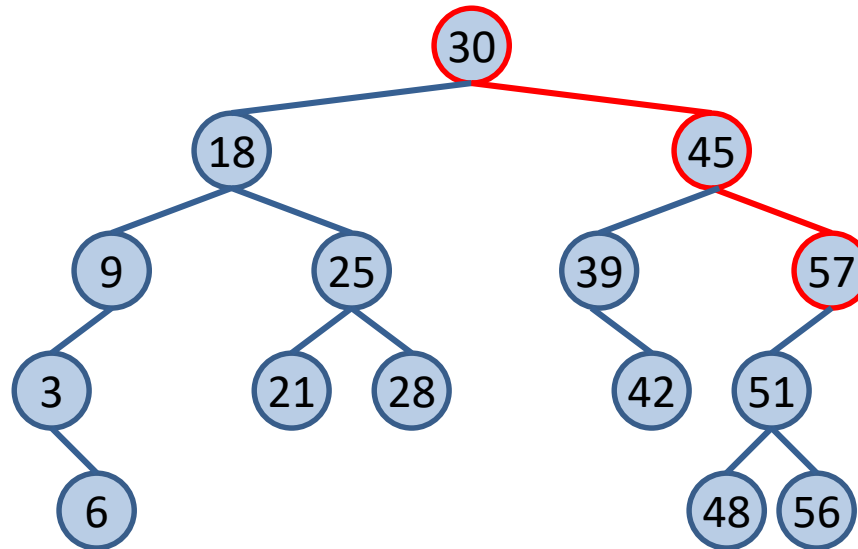


Buscar 51

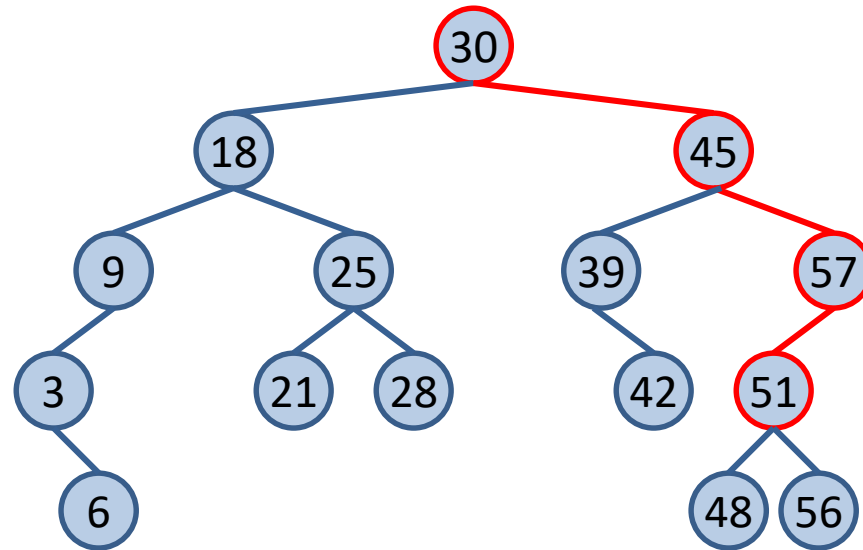




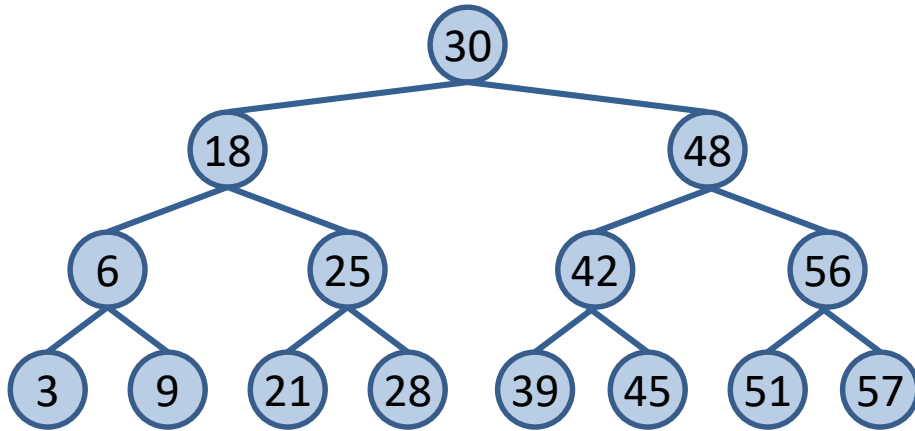
Buscar 51



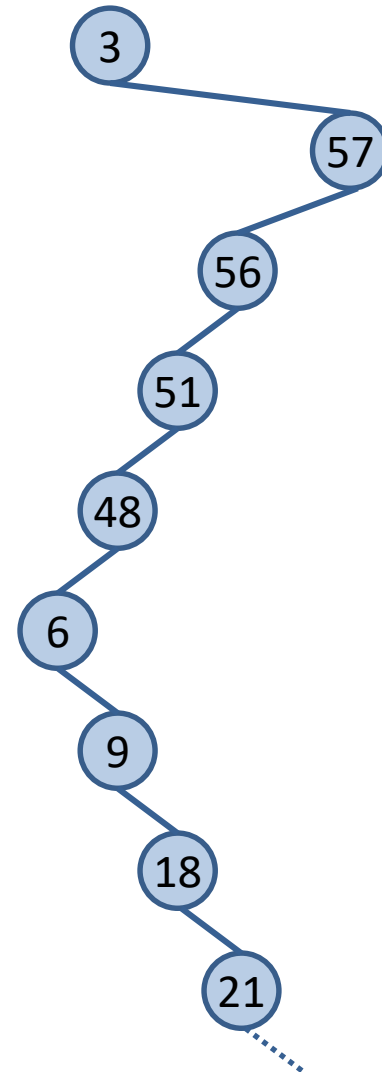
Buscar 51



El tiempo de búsqueda depende de la estructura de ramificación del árbol.



$O(\log_2 n)$



$O(n)$

# TAD Árbol binario de búsqueda

## Definición:

Un árbol binario de búsqueda es un árbol binario en el que los nodos almacenan elementos de un conjunto (no existen elementos repetidos). La propiedad que define a estos árboles es que todos los elementos almacenados en el subárbol izquierdo de cualquier nodo  $n$  son menores que el elemento de  $n$ , y todos los elementos almacenados en el subárbol derecho de  $n$  son mayores que el elemento almacenado en el mismo.

Consideraremos que existe un orden lineal definido sobre el tipo de los elementos dado por el operador  $<$ .

# Operaciones:

*Abb()*

Post: Construye un árbol binario de búsqueda vacío.

*const Abb& buscar(const T& e) const*

Post: Si el elemento *e* pertenece al árbol, devuelve el subárbol en cuya raíz se encuentra *e*; en caso contrario, devuelve un árbol vacío.

*void insertar(const T& e)*

Post: Si *e* no pertenece al árbol, lo inserta; en caso contrario, el árbol no se modifica.

*void eliminar(const T& e)*

Post: Elimina el elemento *e* del árbol. Si *e* no se encuentra, el árbol no se modifica.

*bool vacio() const*

Post: Devuelve **true** si el árbol está vacío y **false** en caso contrario.

Aquí no trabajamos con nodos, si no que trabajamos con subárboles

*const T& elemento() const*

Pre: Árbol no vacío.

Post: Devuelve el elemento de la raíz de un árbol binario de búsqueda.

*const Abb& izqdo() const*

Pre: Árbol no vacío.

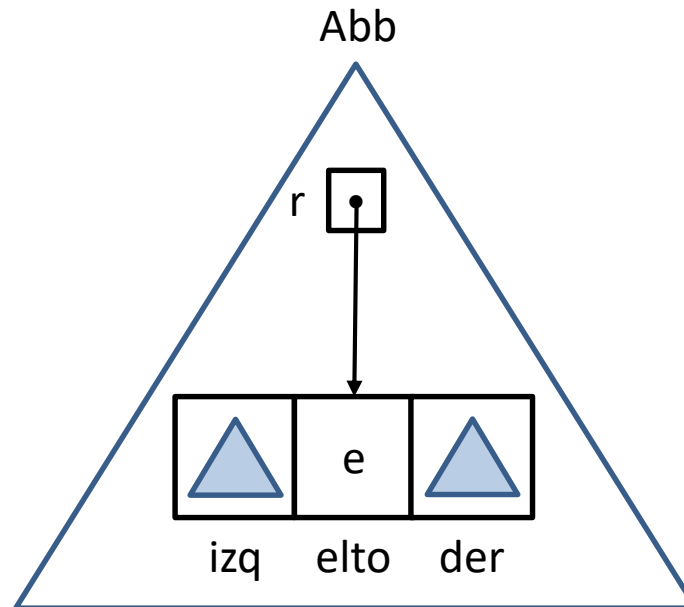
Post: Devuelve el subárbol izquierdo.

*const Abb& drcho() const*

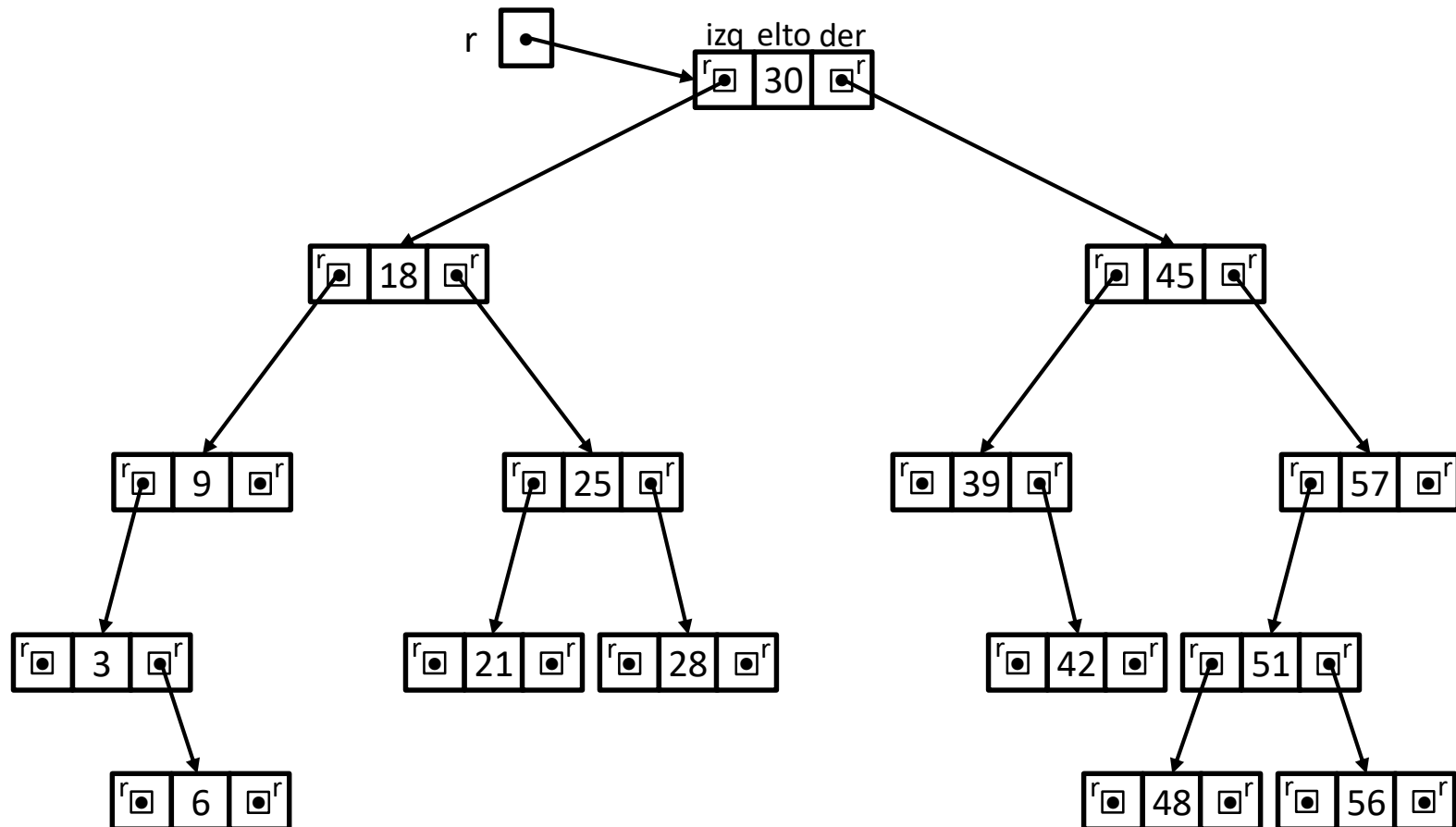
Pre: Árbol no vacío.

Post: Devuelve el subárbol derecho.

# Implementación de árboles binarios de búsqueda mediante una estructura dinámica recursiva



# Implementación de árboles binarios de búsqueda mediante una estructura dinámica recursiva





```

#ifndef ABB_H
#define ABB_H
#include <cassert>

template <typename T> class Abb {
public:
    Abb() ;
    const Abb& buscar(const T& e) const;
    void insertar(const T& e) ;
    void eliminar(const T& e) ;
    bool vacio() const;
    const T& elemento() const;
    const Abb& izqdo() const;
    const Abb& drcho() const;
    Abb(const Abb& A) ;           // ctor. de copia
    Abb& operator =(const Abb& A) ; // asig. árboles
    ~Abb() ;                     // destructor

```

```

private:
    struct arbol {
        T elto;
        Abb izq, der;

        arbol(const T& e): elto{e}, izq{}, der{} {}
    };

    arbol* r;    // raíz del árbol

    T borrarMin();
};

```

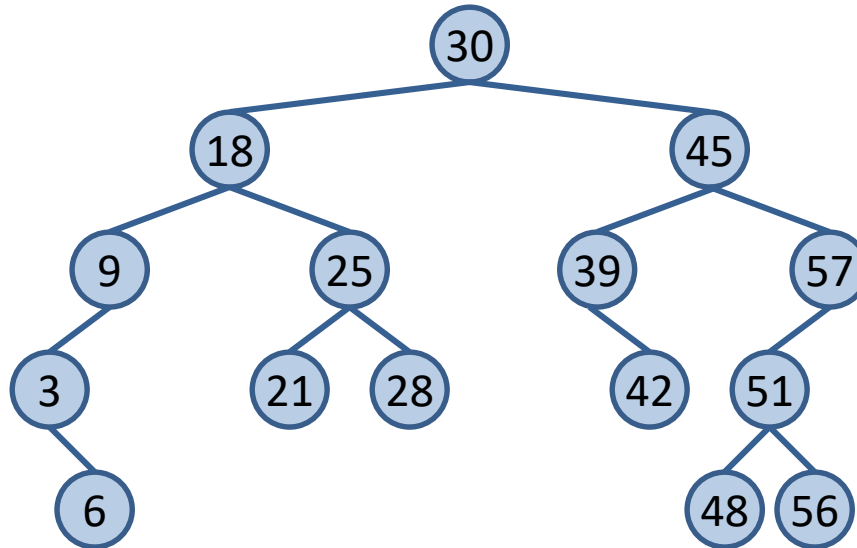
```
template <typename T>
inline Abb<T>::Abb() : r{nullptr} {}
```

```
template <typename T>
inline bool Abb<T>::vacio() const
{
    return (r == nullptr);
}
```

```
template <typename T> método más importante del TAD
const Abb<T>& Abb<T>::buscar(const T& e) const
{
    Si el elemento a buscar no está, devuelve un subárbol vacío
    if (r == nullptr)                // Árbol vacío, e no encontrado.
        return *this;
    else if (e < r->elto)              // Buscar en subárbol izqdo.
        return r->izq.buscar(e);
    else if (r->elto < e)              // Buscar en subárbol drcho.
        return r->der.buscar(e);
    else                             // Encontrado e en la raíz.
        return *this;
}
```

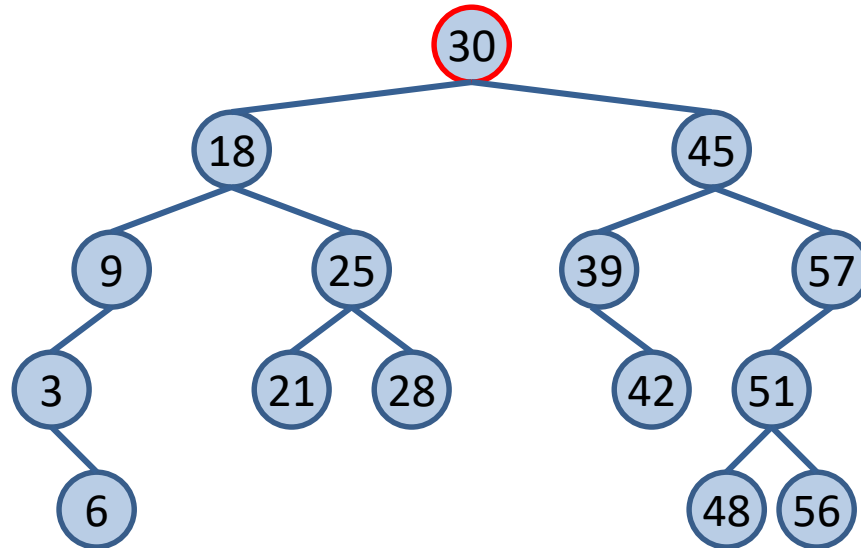
# Inserción en un ABB

Insertar 35



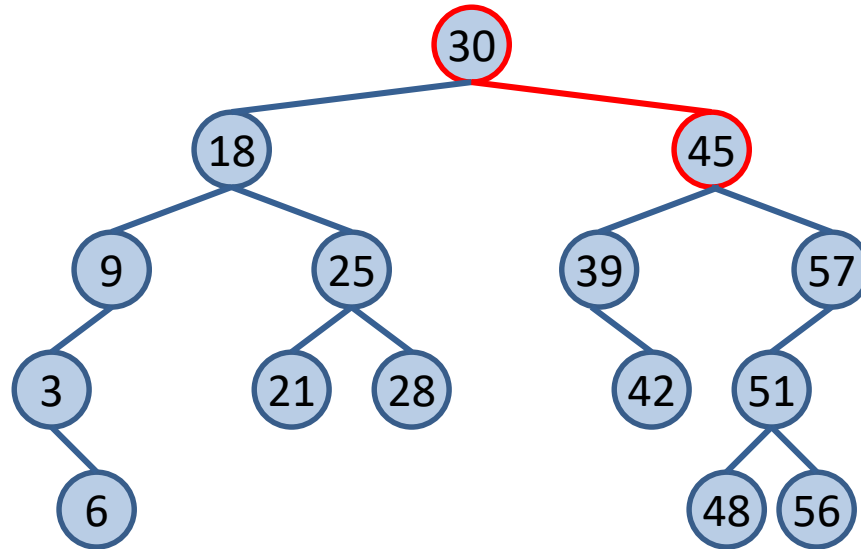
# Inserción en un ABB

Insertar 35



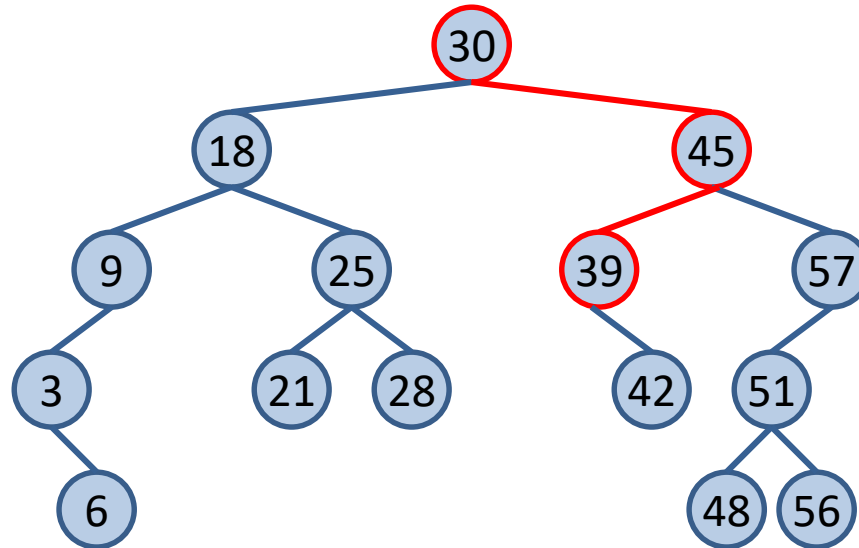
# Inserción en un ABB

Insertar 35



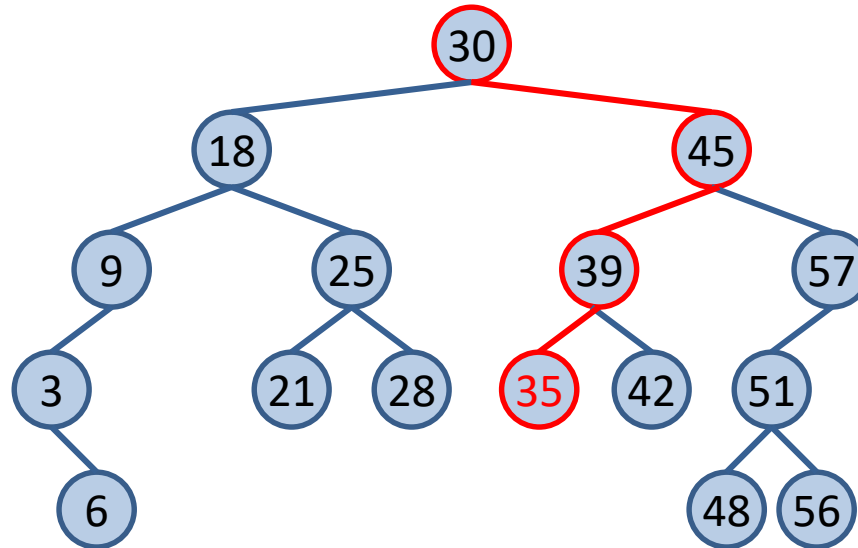
# Inserción en un ABB

Insertar 35



# Inserción en un ABB

Insertar 35





```

template <typename T>
void Abb<T>::insertar(const T& e)
{
    if (r == nullptr)                // Árbol vacío.
        r = new arbol(e);
    else if (e < r->elto)              // Insertar en el subárbol izqdo.
        r->izq.insertar(e);
    else if (r->elto < e)              // Insertar en el subárbol drcho.
        r->der.insertar(e);
}

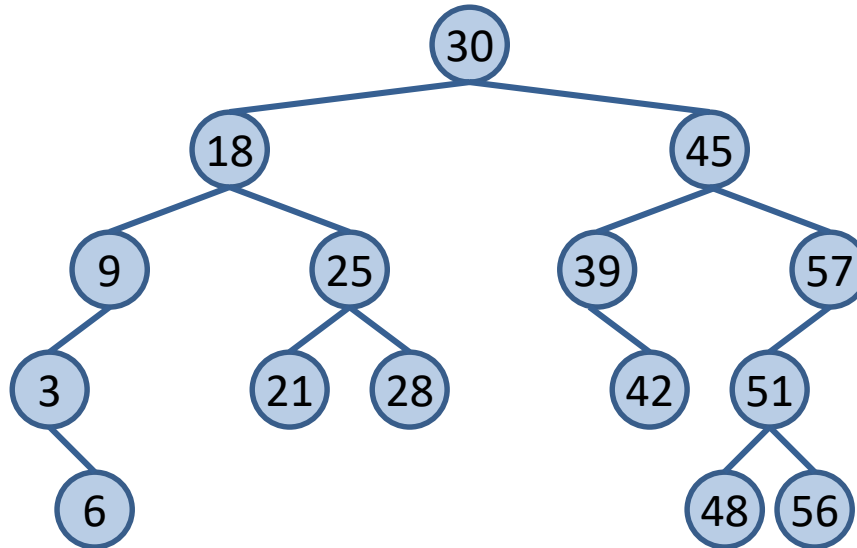
```

Hacemos una búsqueda para encontrar un hueco donde quepe  
Si nos encontramos con que ese elemento ya está insertad no hacemos nada.

# Eliminación en un ABB

## Caso 1: Suprimir una hoja

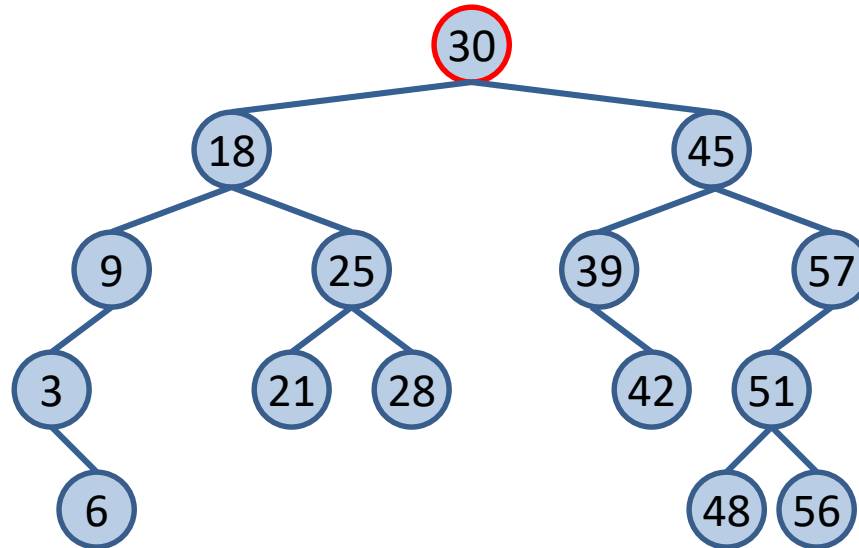
Suprimir 28



# Eliminación en un ABB

## Caso 1: Suprimir una hoja

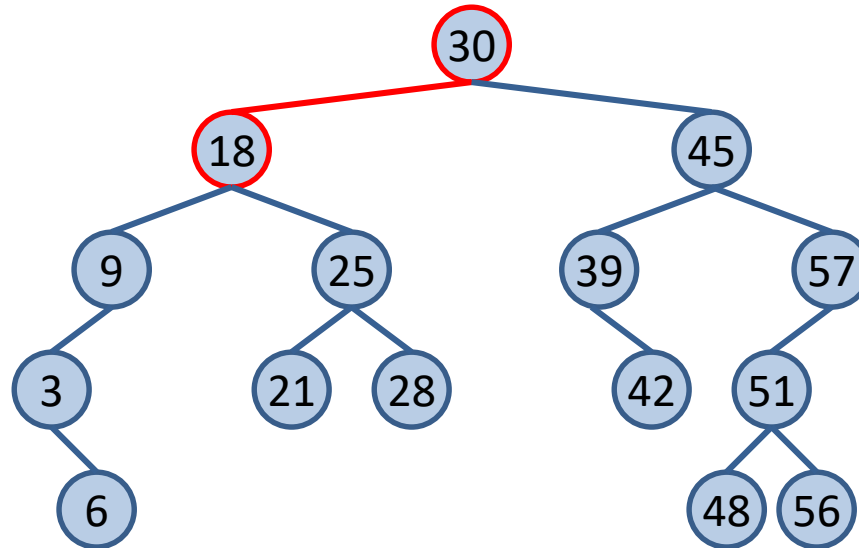
Suprimir 28



# Eliminación en un ABB

## Caso 1: Suprimir una hoja

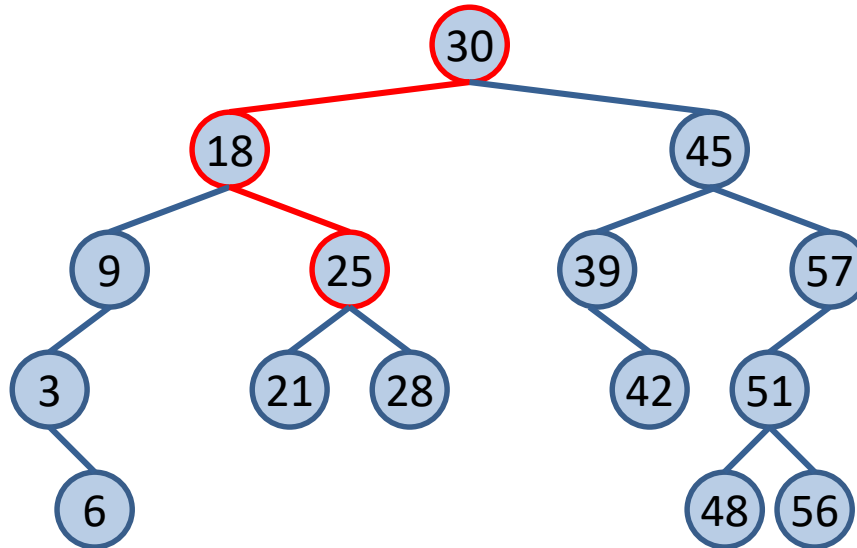
Suprimir 28



# Eliminación en un ABB

## Caso 1: Suprimir una hoja

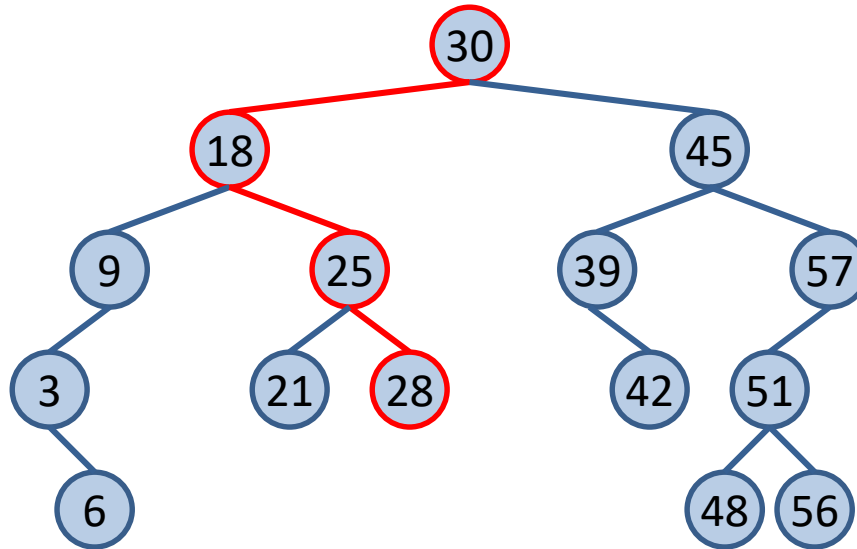
Suprimir 28



# Eliminación en un ABB

## Caso 1: Suprimir una hoja

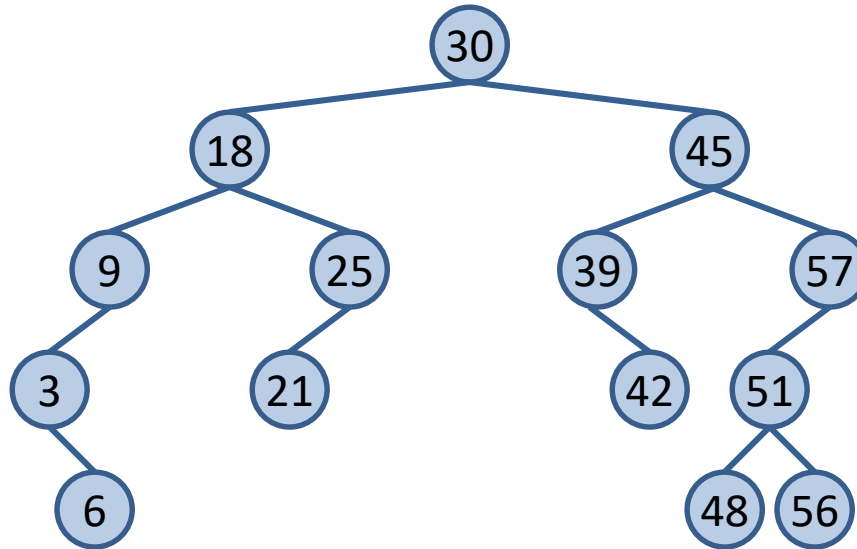
Suprimir 28



# Eliminación en un ABB

## Caso 1: Suprimir una hoja

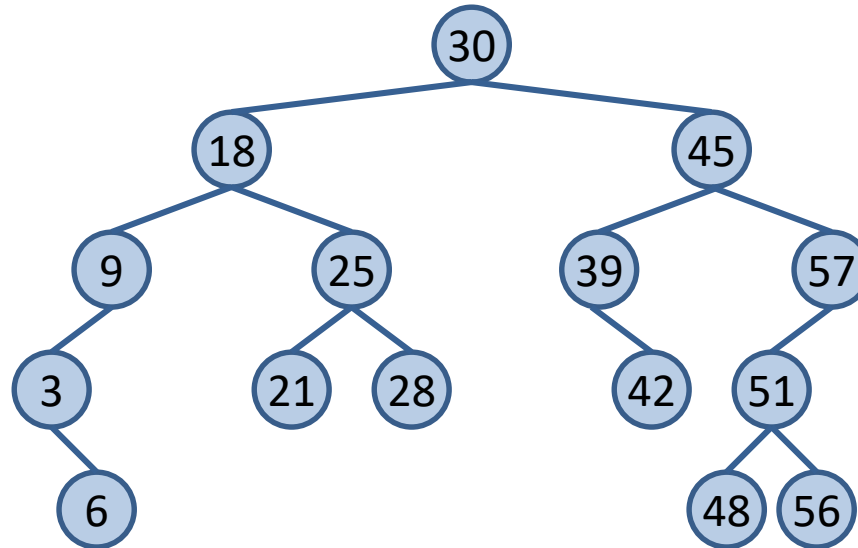
Suprimir 28



# Eliminación en un ABB

Caso 2: Suprimir un nodo con sólo hijo izquierdo

Suprimir 9

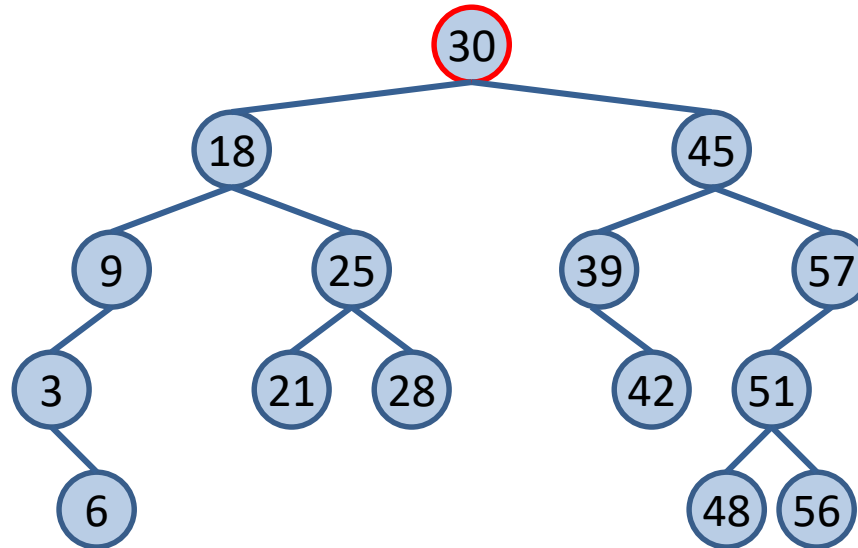




# Eliminación en un ABB

Caso 2: Suprimir un nodo con sólo hijo izquierdo

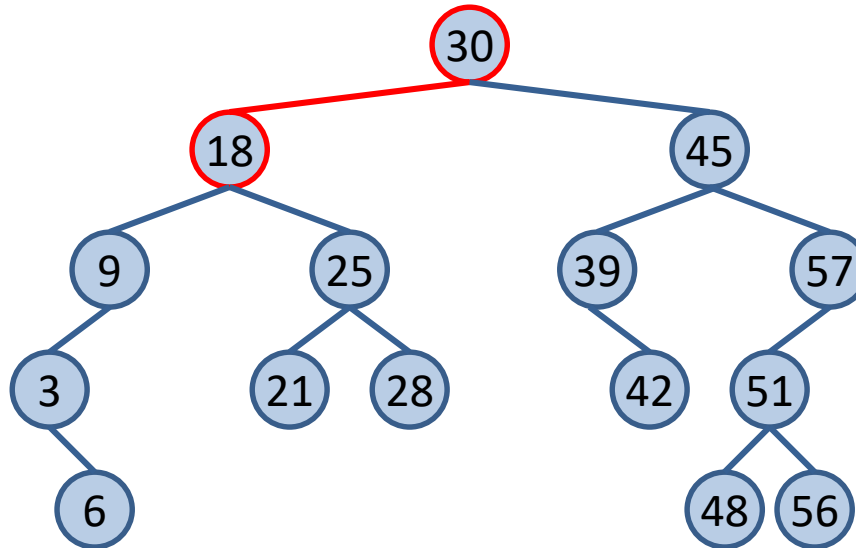
Suprimir 9



# Eliminación en un ABB

## Caso 2: Suprimir un nodo con sólo hijo izquierdo

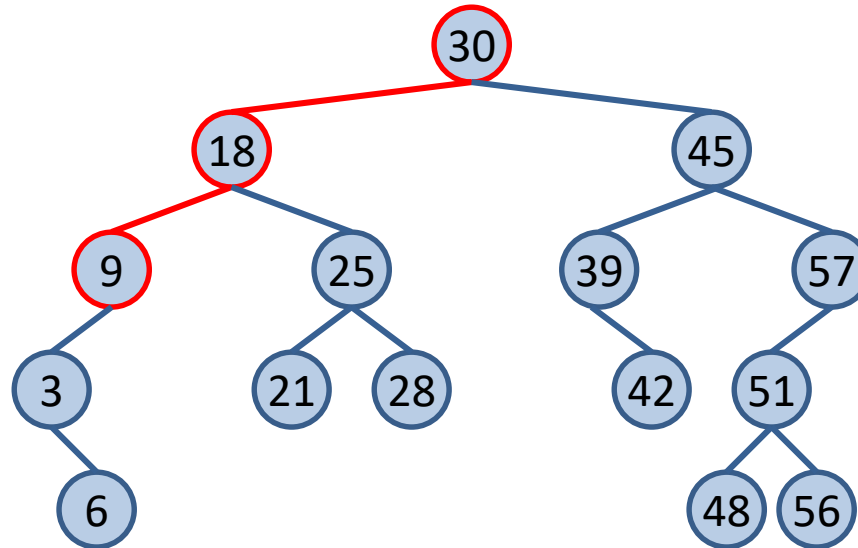
Suprimir 9



# Eliminación en un ABB

## Caso 2: Suprimir un nodo con sólo hijo izquierdo

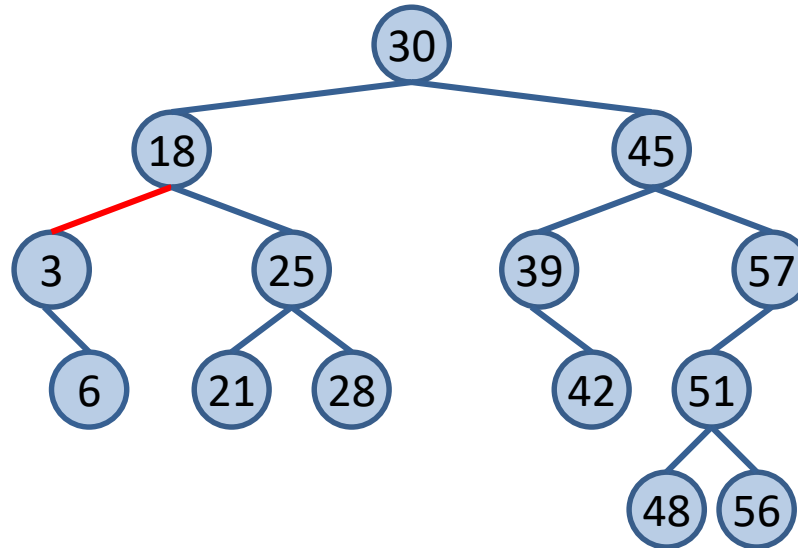
Suprimir 9



# Eliminación en un ABB

Caso 2: Suprimir un nodo con sólo hijo izquierdo

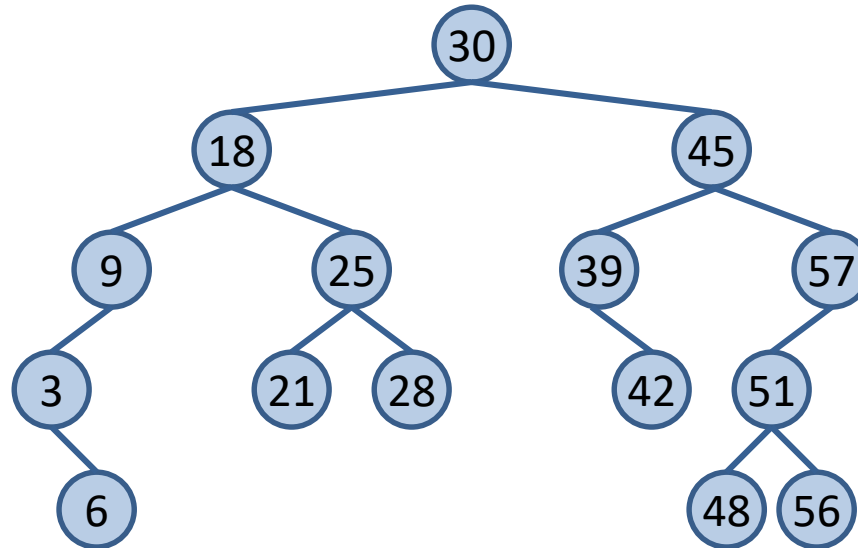
Suprimir 9



# Eliminación en un ABB

## Caso 3: Suprimir un nodo con sólo hijo derecho

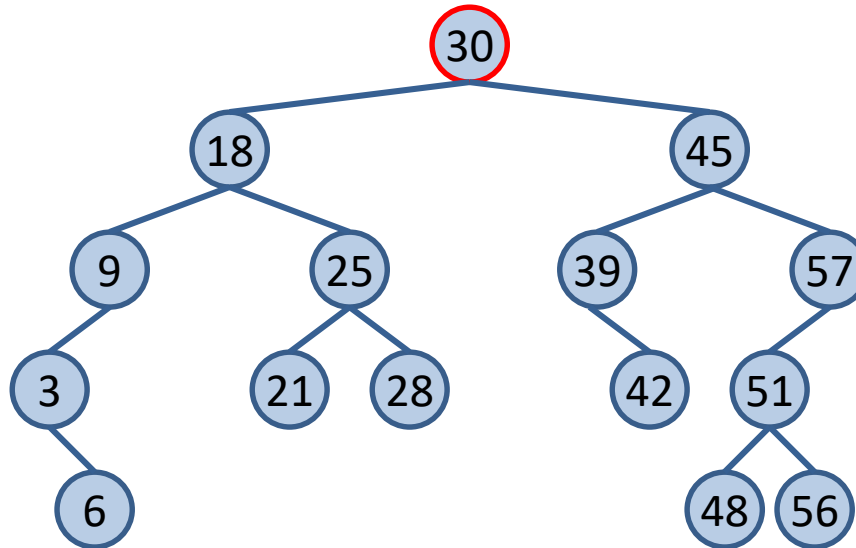
Suprimir 39



# Eliminación en un ABB

## Caso 3: Suprimir un nodo con sólo hijo derecho

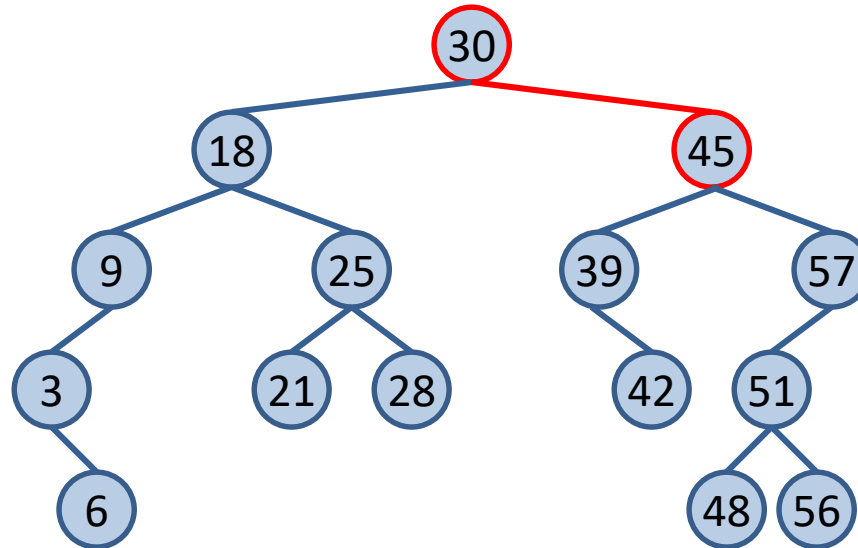
Suprimir 39



# Eliminación en un ABB

## Caso 3: Suprimir un nodo con sólo hijo derecho

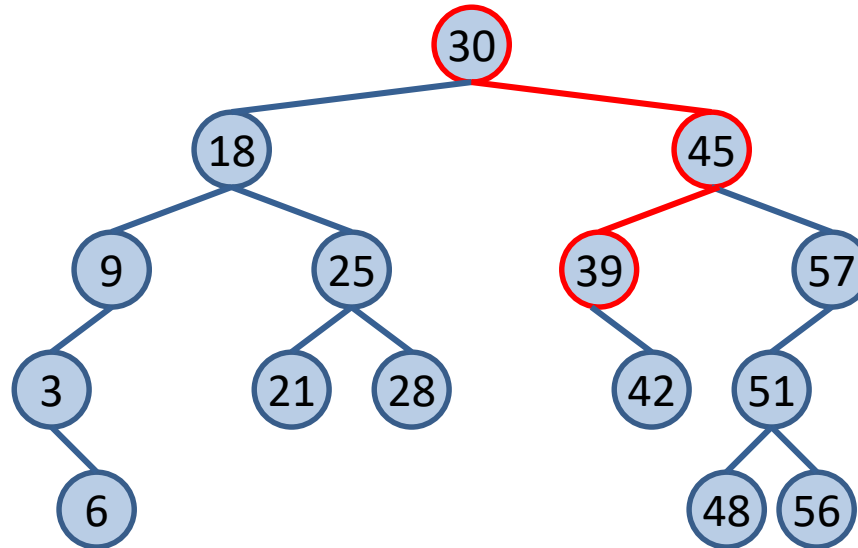
Suprimir 39



# Eliminación en un ABB

## Caso 3: Suprimir un nodo con sólo hijo derecho

Suprimir 39

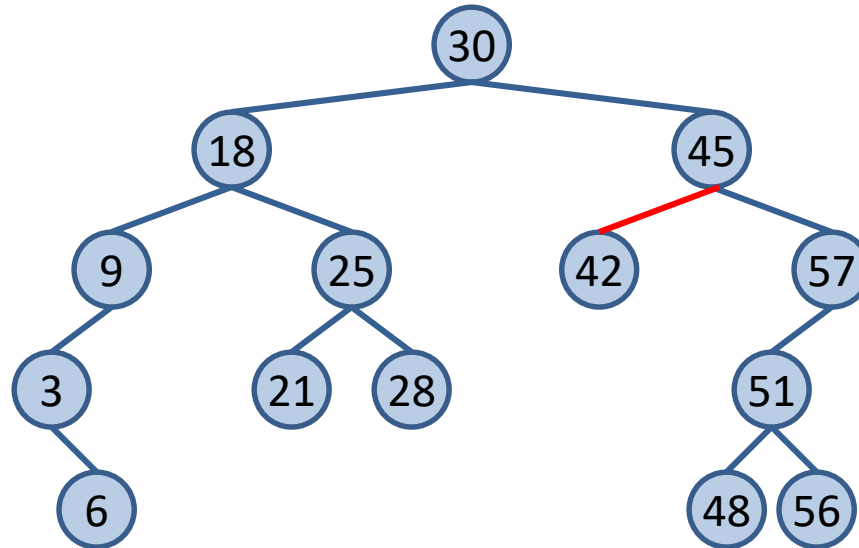




# Eliminación en un ABB

## Caso 3: Suprimir un nodo con sólo hijo derecho

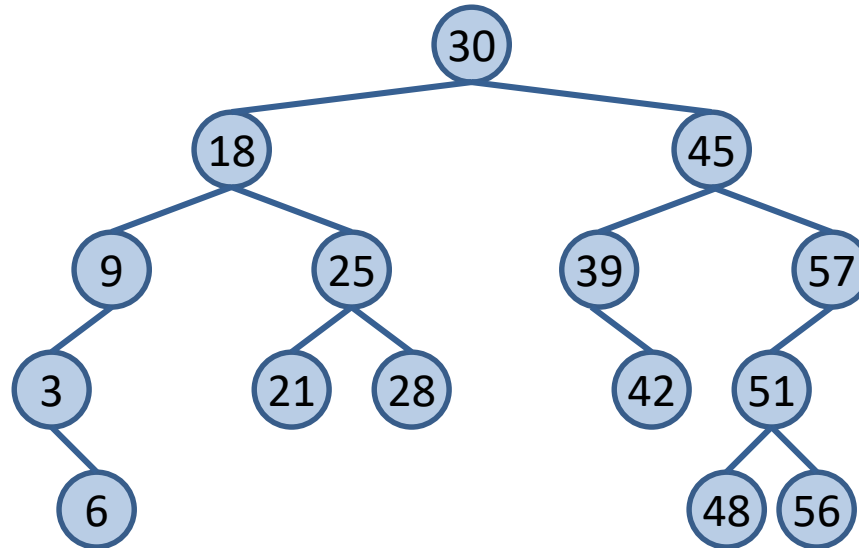
Suprimir 39



# Eliminación en un ABB

## Caso 4: Suprimir un nodo con dos hijos

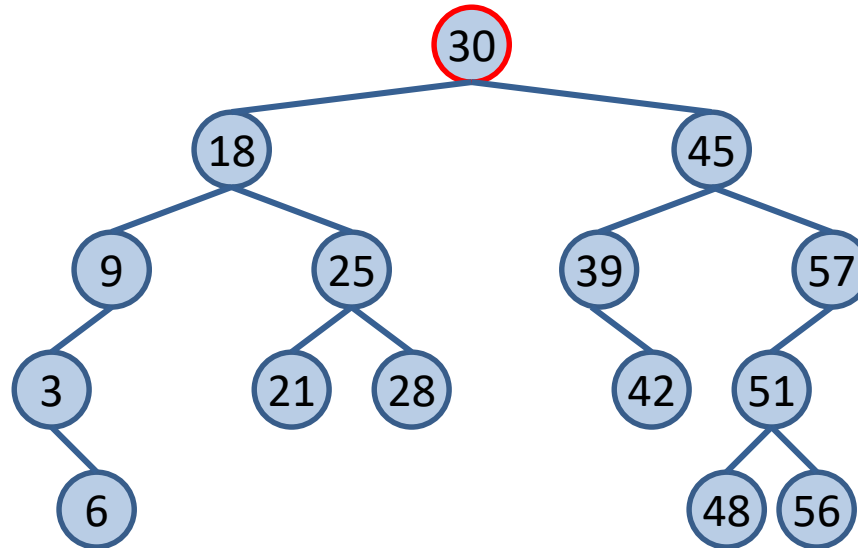
Suprimir 30



# Eliminación en un ABB

## Caso 4: Suprimir un nodo con dos hijos

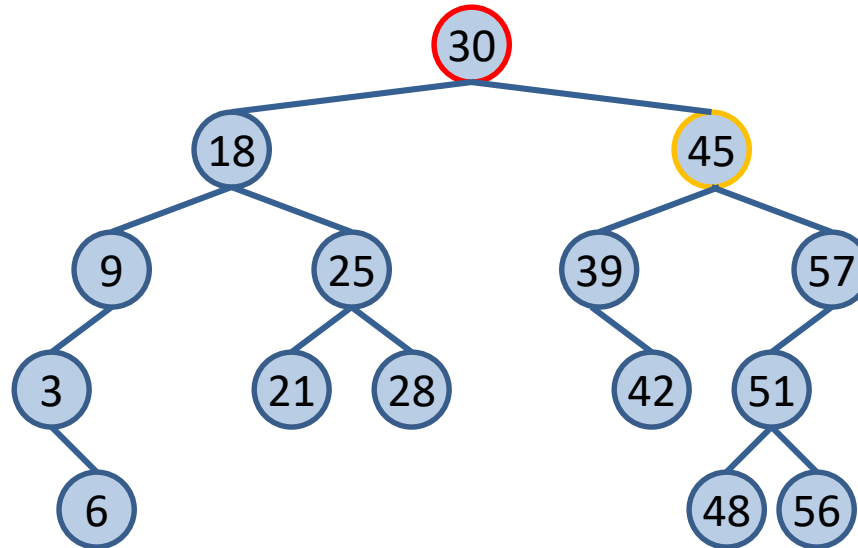
Suprimir 30



# Eliminación en un ABB

## Caso 4: Suprimir un nodo con dos hijos

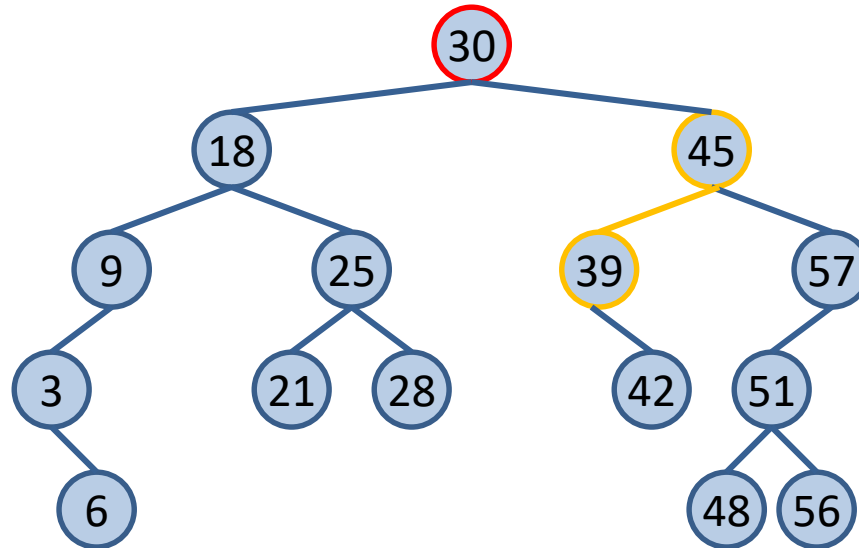
Suprimir 30



# Eliminación en un ABB

## Caso 4: Suprimir un nodo con dos hijos

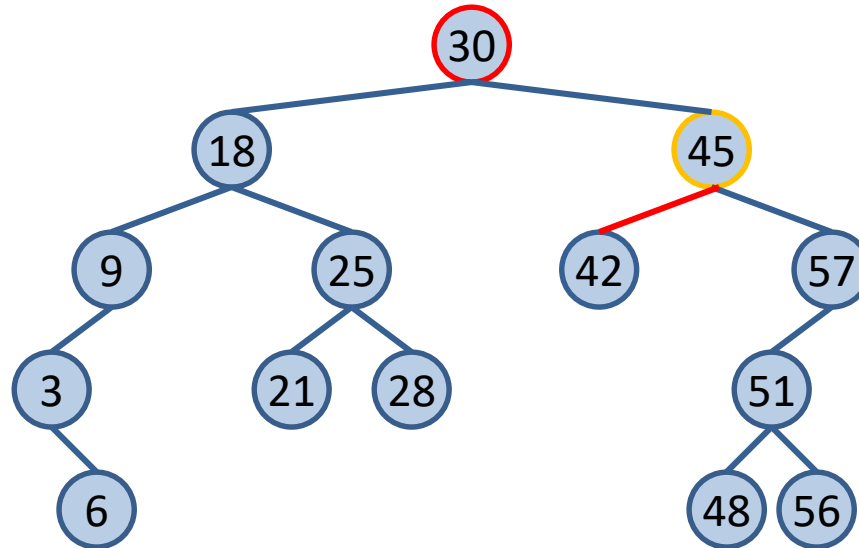
Suprimir 30



# Eliminación en un ABB

## Caso 4: Suprimir un nodo con dos hijos

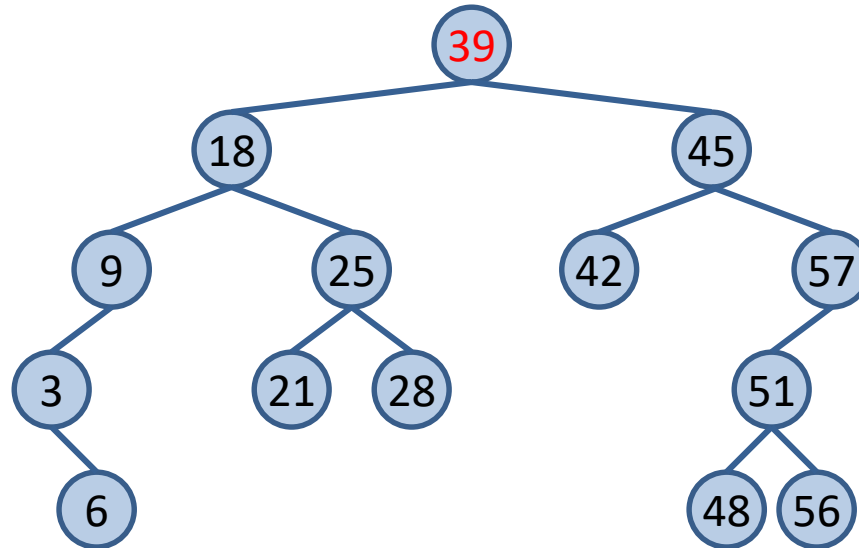
Suprimir 30



# Eliminación en un ABB

## Caso 4: Suprimir un nodo con dos hijos

Suprimir 30



```

template <typename T>
void Abb<T>::eliminar(const T& e)
{
    if (r != nullptr) {                // Árbol no vacío.
        if (e < r->elto)                // Quitar e del subárbol izqdo.
            r->izq.eliminar(e);
        else if (r->elto < e)           // Quitar e del subárbol drcho.
            r->der.eliminar(e);
    }
}

```



```

else    // Quitar e de la raíz.
    if (!r->izq.r && !r->der.r) { // 1. Raíz es hoja.
        delete r;
        r = nullptr; // El árbol queda vacío.
    }
    else if (!r->der.r) { // 2. Raíz sólo tiene hijo izqdo.
        arbol* a = r->izq.r;
        r->izq.r = nullptr; // Evita destruir el subárbol izqdo.
        delete r;
        r = a;
    }
    else if (!r->izq.r) { // 3. Raíz sólo tiene hijo drcho.
        arbol* a = r->der.r;
        r->der.r = nullptr; // Evita destruir el subárbol drcho.
        delete r;
        r = a;
    }
    else    // 4. Raíz tiene dos hijos
        // Eliminar el mínimo del subárbol derecho y sustituir
        // el elemento de la raíz por éste.
        r->elto = r->der.borrarMin();
}
}

```

// Método privado

```
template <typename T>
T Abb<T>::borrarMin()
// Elimina el nodo que almacena el menor elemento
// del árbol. Devuelve el elemento del nodo eliminado.
{
    if (r->izq.r == nullptr) { // Subárbol izquierdo vacío.
        T e = r->elto;
        arbol* hd = r->der.r;
        r->der.r = nullptr; // Evita destruir subárbol drcho.
        delete r;
        r = hd; // Sustituir r por el subárbol drcho.
        return e;
    }
    else
        return r->izq.borrarMin();
}
```

```

template <typename T>
inline const T& Abb<T>::elemento() const
{
    assert(r != nullptr);
    return r->elto;
}

template <typename T>
inline const Abb<T>& Abb<T>::izqdo() const
{
    assert(r != nullptr);
    return r->izq;
}

template <typename T>
inline const Abb<T>& Abb<T>::drcho() const
{
    assert(r != nullptr);
    return r->der;
}

```

# Copia y destrucción de un ABB

```
template <typename T>
inline Abb<T>::Abb(const Abb<T>& A) : r{nullptr}
{
    if (A.r != nullptr)
        r = new arbol(*A.r); // Copiar raíz y descendientes.
}

template <typename T>
Abb<T>& Abb<T>::operator =(const Abb<T>& A)
{
    if (this != &A) { // Evitar autoasignación.
        this->~Abb(); // Vaciar el árbol.
        if (A.r != nullptr)
            r = new arbol(*A.r); // Copiar raíz y descendientes.
    }
    return *this;
}
```

```

template <typename T>
Abb<T>::~~Abb()
{
    if (r != nullptr) { // Árbol no vacío.
        delete r; // Destruir raíz y descendientes con r->~arbol()
        r = nullptr; // El árbol queda vacío.
    }
}

#endif // ABB_H

```