Estructuras de Datos no Lineales

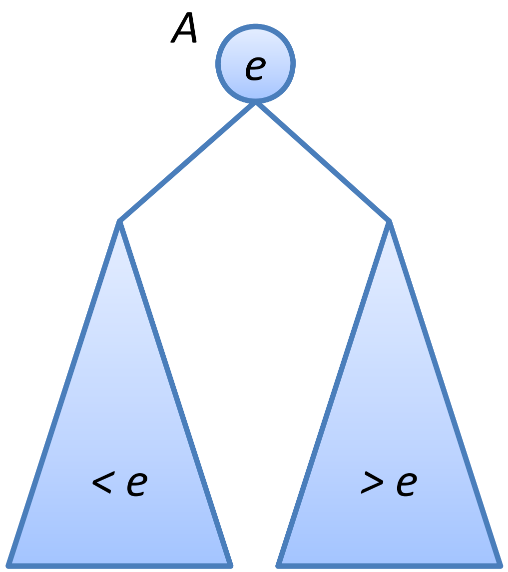
1.5. Árboles binarios de búsqueda

José Fidel Argudo Argudo
José Antonio Alonso de la Huerta
Mª Teresa García Horcajadas

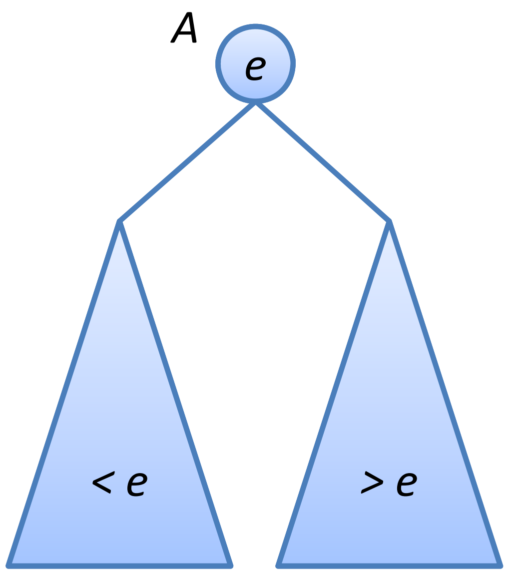
UCA
Universidad de Cádiz

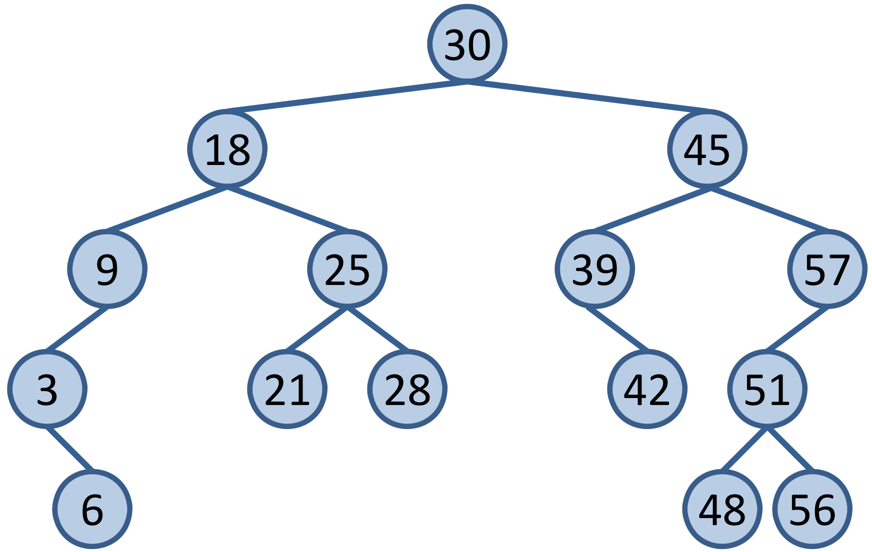
Versión 2.0

Árboles binarios de búsqueda

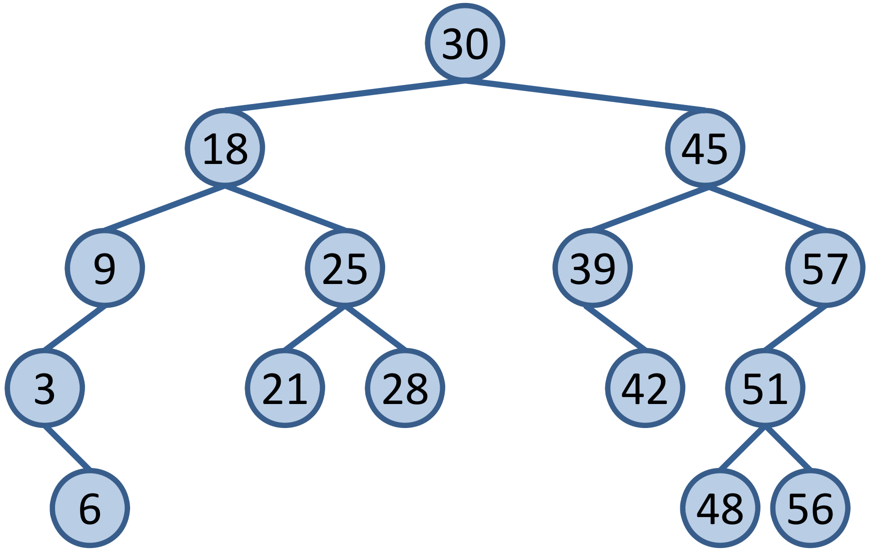


Árboles binarios de búsqueda

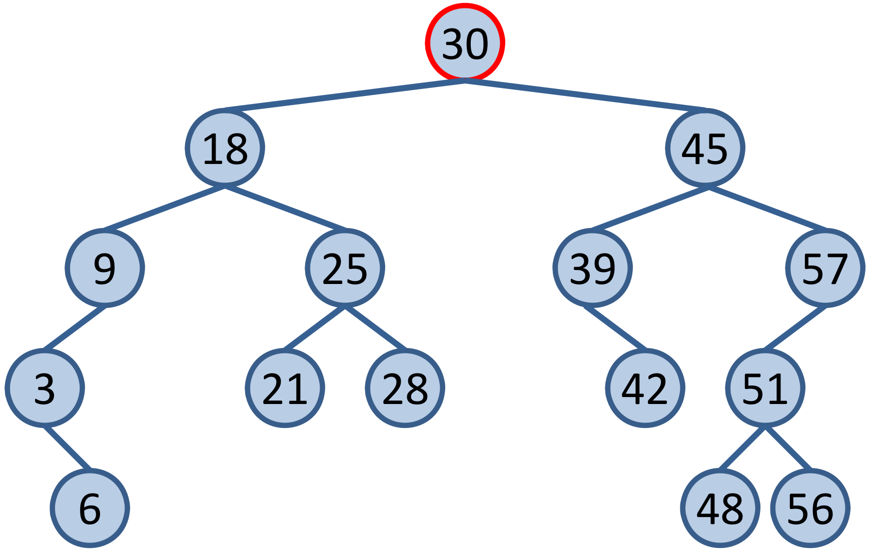




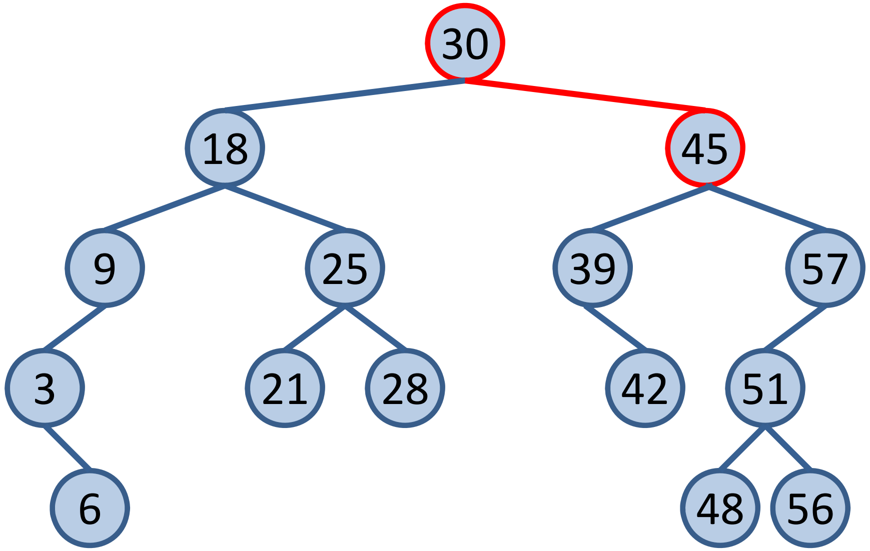
Buscar 51



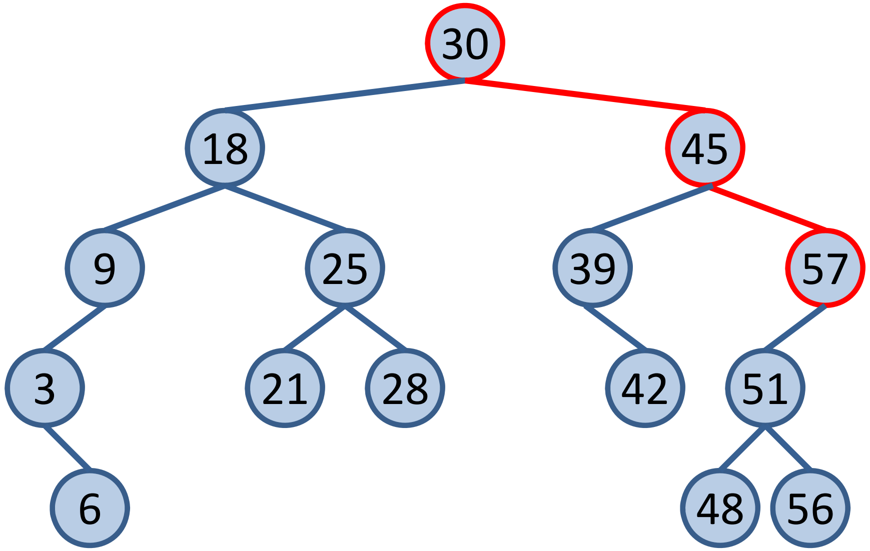
Buscar 51



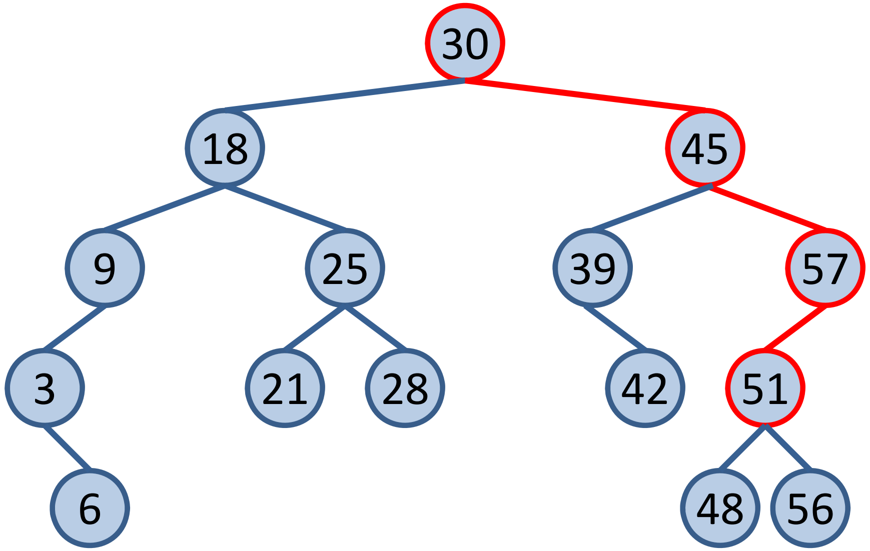
Buscar 51



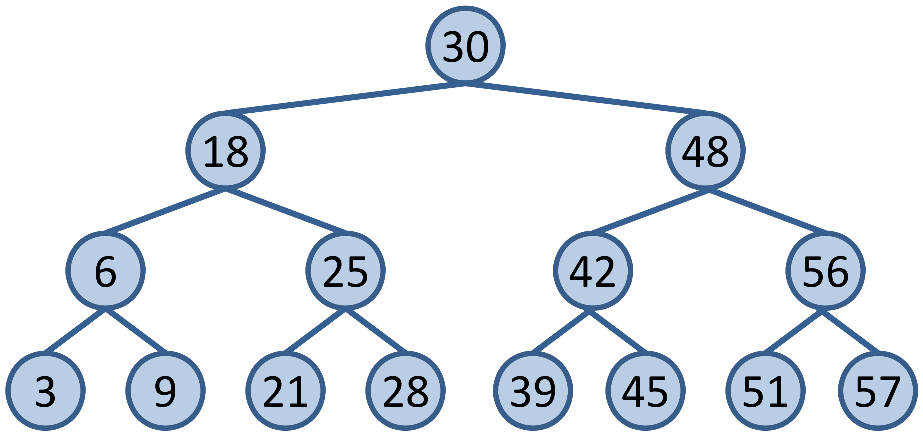
Buscar 51



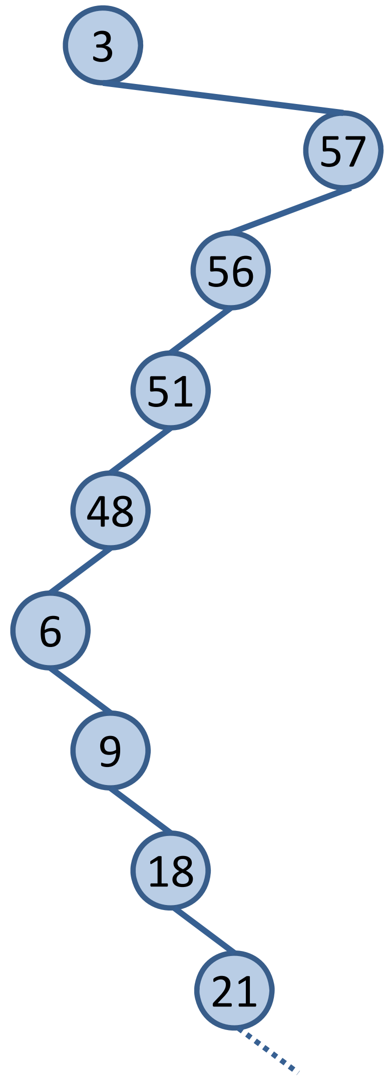
Buscar 51



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()

<u>Post</u>: Construye un árbol binario de búsqueda vacío.

const Abb& buscar(const T& e) const

<u>Post</u>: 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)

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

void eliminar(const T& e)

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

bool vacio() const

<u>Post</u>: Devuelve **`true`** si el árbol está vacío y **`false`** en caso contrario.

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.

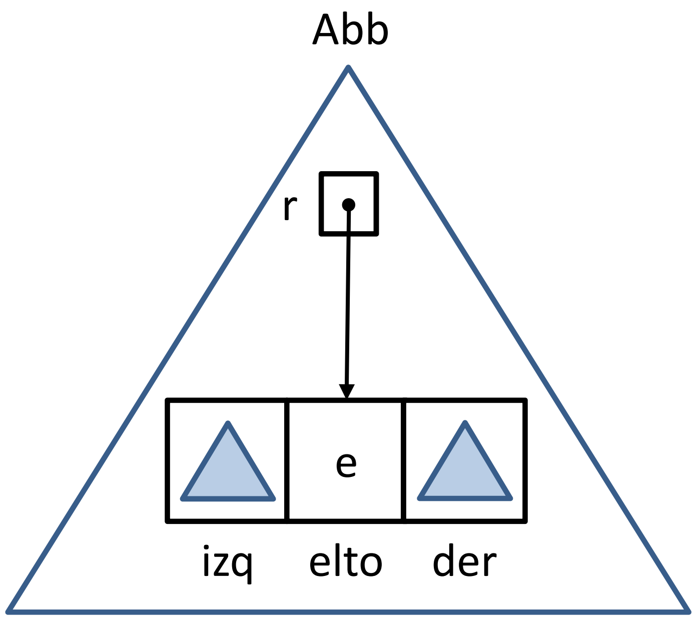
Post: Devuelve el subárbol izquierdo.

const Abb& drcho() const

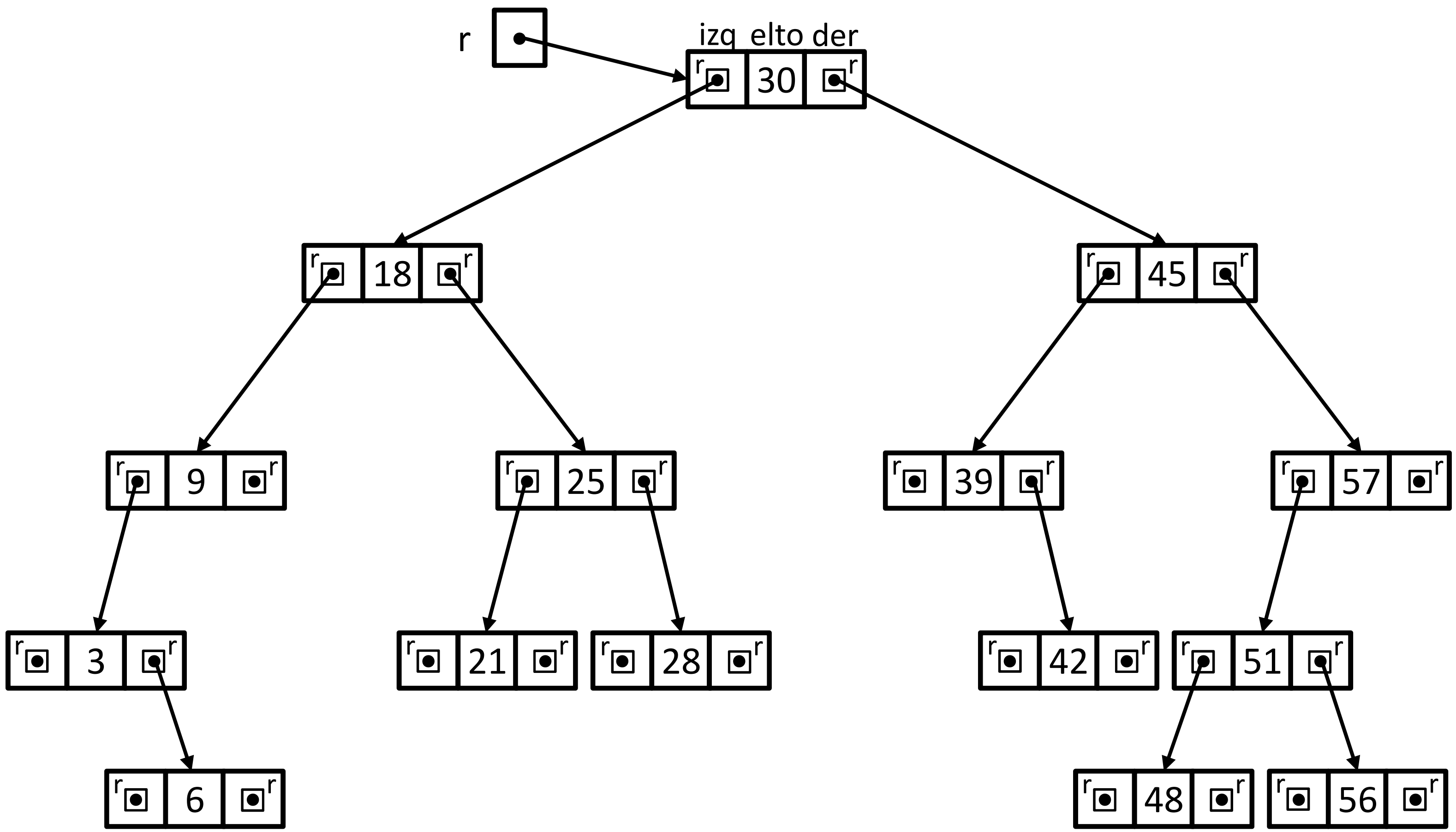
Pre: Árbol no vacío.

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



```cpp
#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
```

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

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

   arbol* r;   // raíz del árbol

   T borrarMin();
};
```

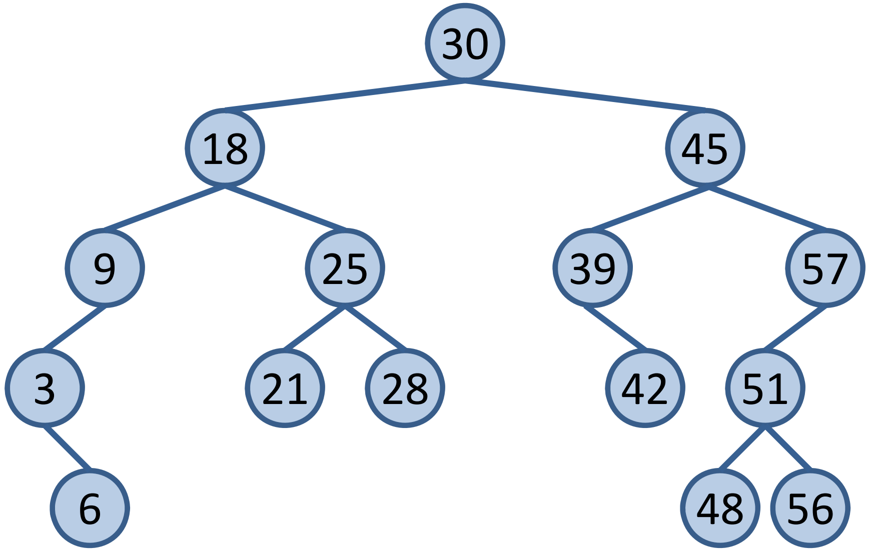
```cpp
template <typename T>
inline Abb<T>::Abb() : r{nullptr} {}

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

template <typename T>
const Abb<T>& Abb<T>::buscar(const T& e) const
{
   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;
}
```

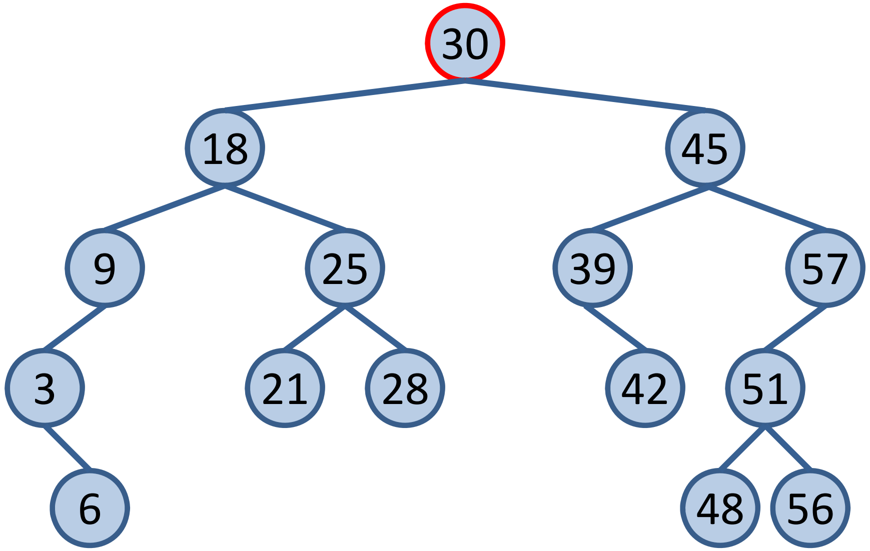
Inserción en un ABB

Insertar 35



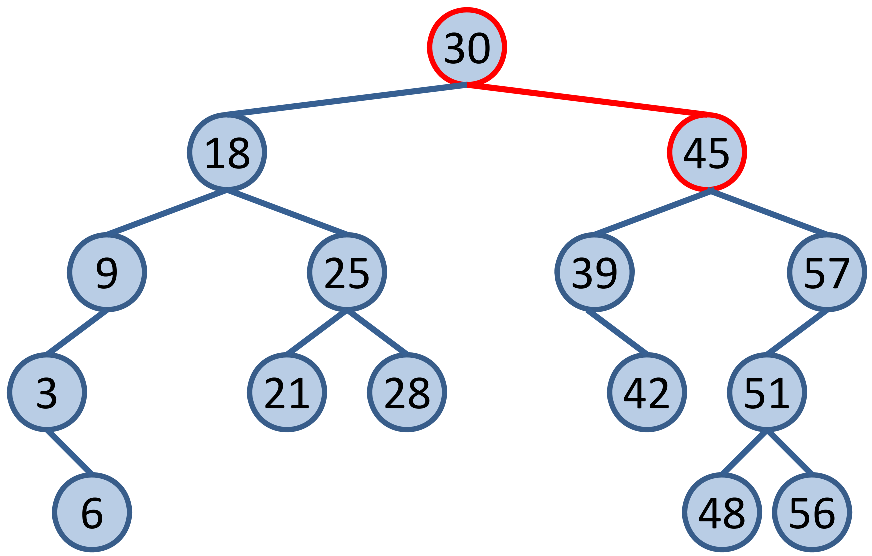
Inserción en un ABB

Insertar 35



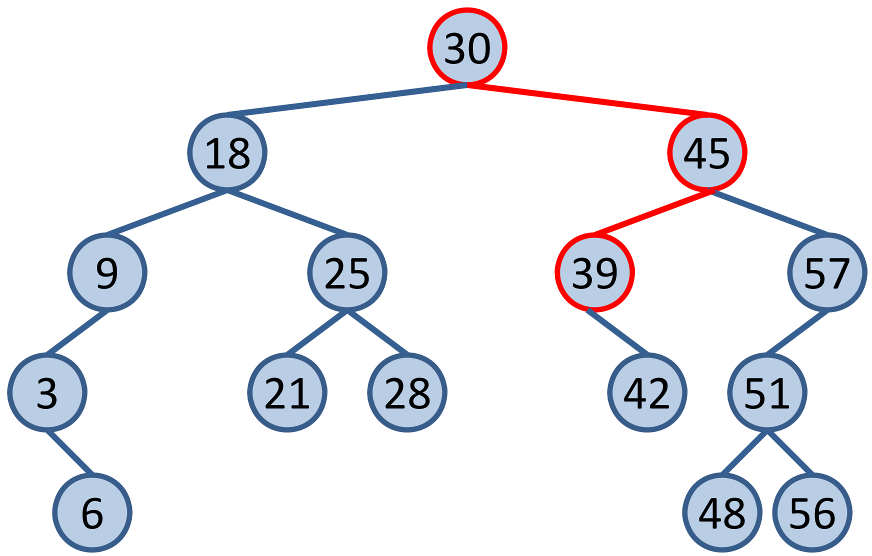
Inserción en un ABB

Insertar 35



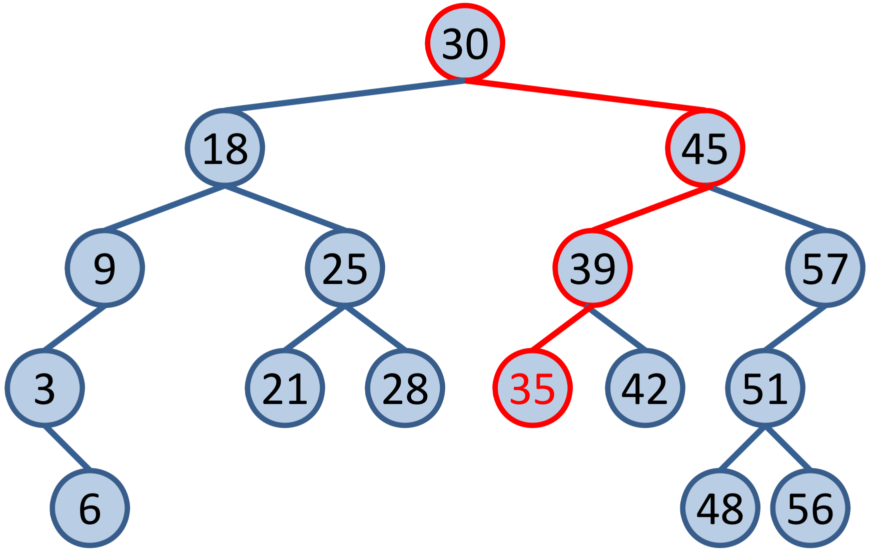
Inserción en un ABB

Insertar 35



Inserción en un ABB

Insertar 35

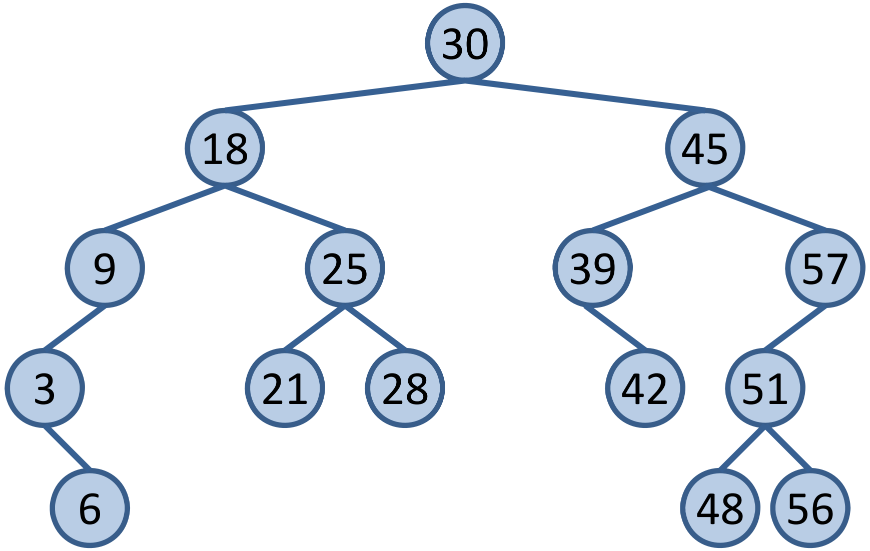


```
template <typename T>
void Abb<T>::insertar(const T& e)
{
   if (r == nullptr)          // Árbol vacío.
      r = new arbol(e);
   else if (e < r->elto)      // Insertar en el subárbol izqdo.
      r->izq.insertar(e);
   else if (r->elto < e)      // Insertar en el subárbol drcho.
      r->der.insertar(e);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

Eliminación en un ABB

Caso 1: Suprimir una hoja

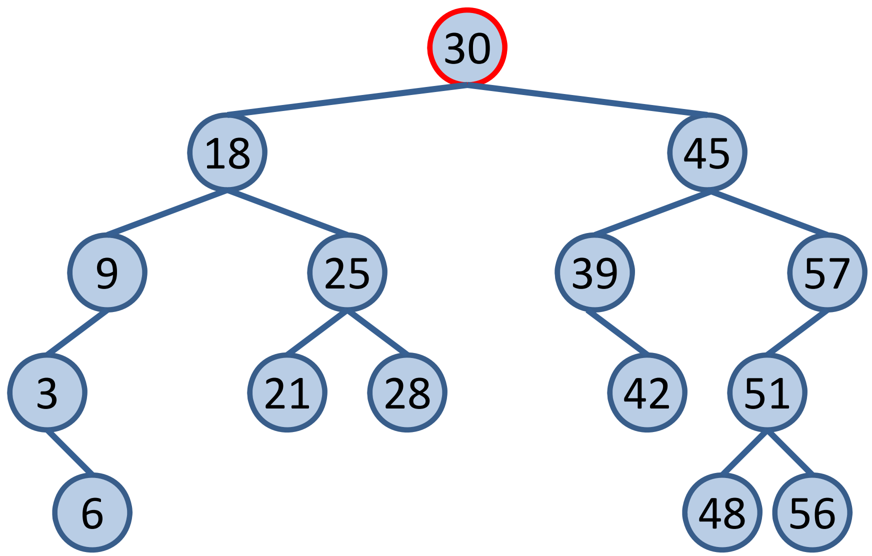
Suprimir 28



Eliminación en un ABB

Caso 1: Suprimir una hoja

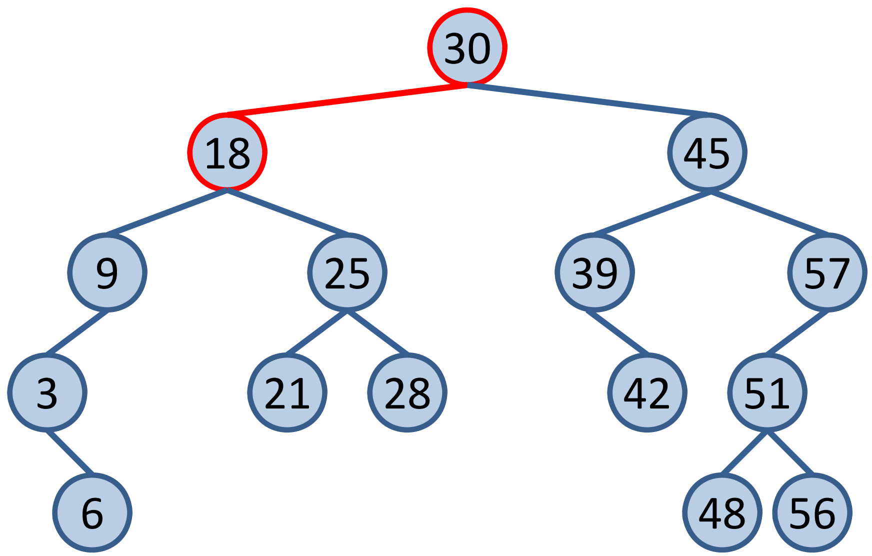
Suprimir 28



Eliminación en un ABB

Caso 1: Suprimir una hoja

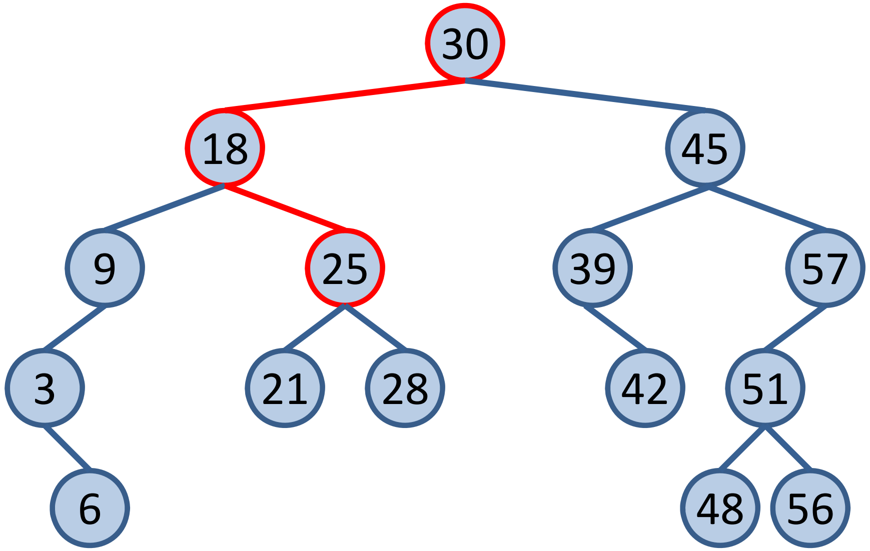
Suprimir 28



Eliminación en un ABB

Caso 1: Suprimir una hoja

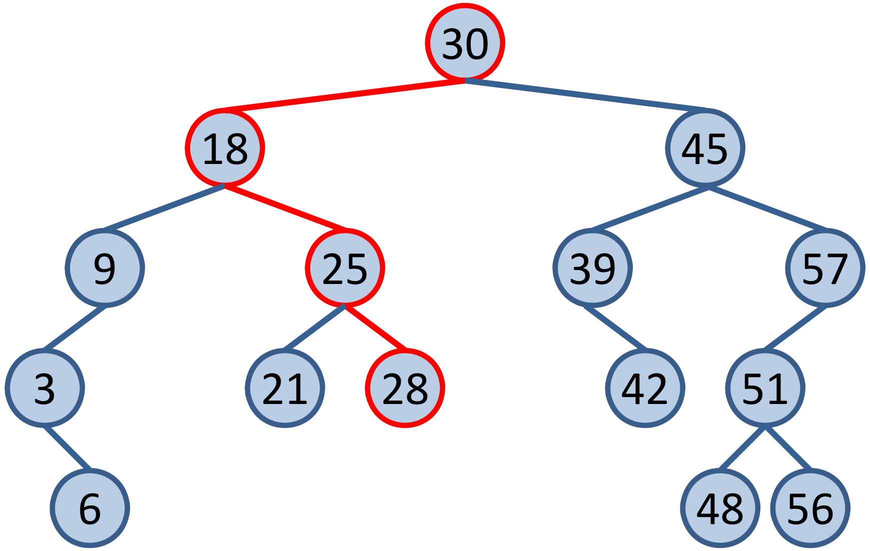
Suprimir 28



Eliminación en un ABB

Caso 1: Suprimir una hoja

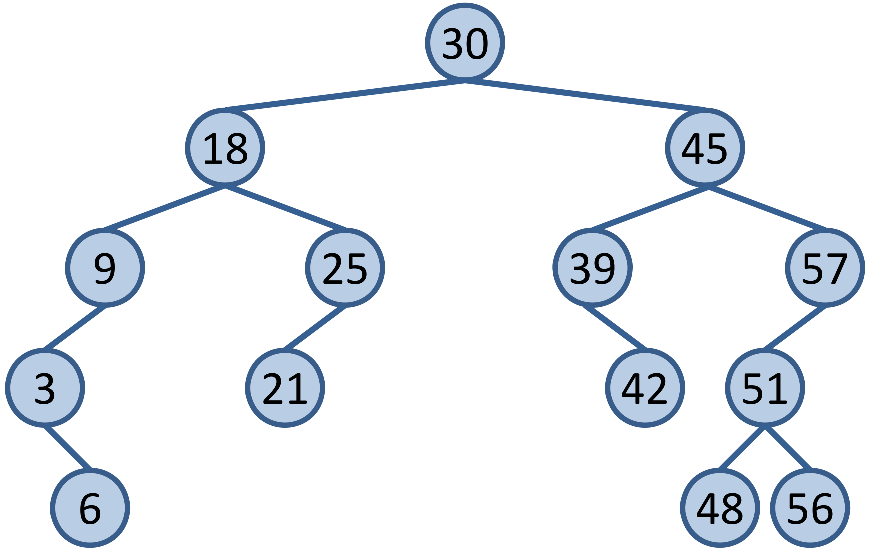
Suprimir 28



Eliminación en un ABB

Caso 1: Suprimir una hoja

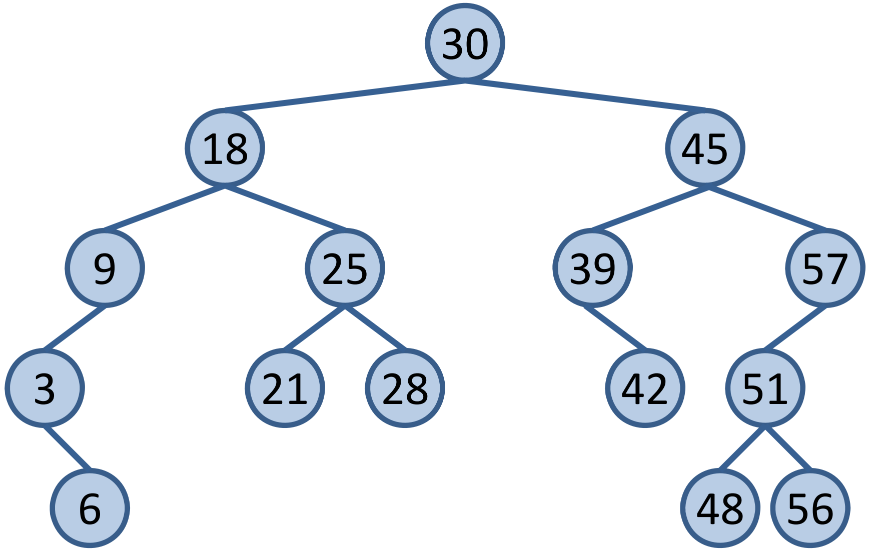
Suprimir 28



Eliminación en un ABB

Caso 2: Suprimir un nodo con sólo hijo izquierdo

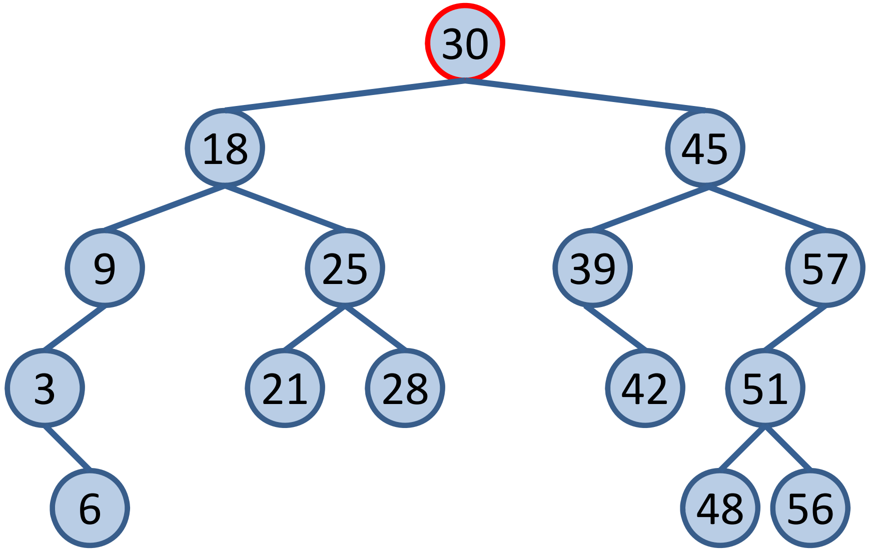
Suprimir 9



Eliminación en un ABB

Caso 2: Suprimir un nodo con sólo hijo izquierdo

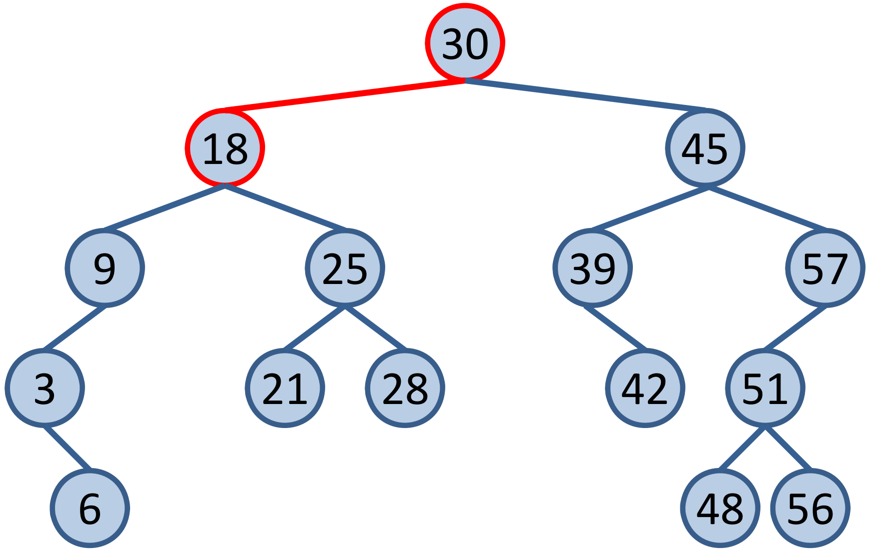
Suprimir 9



Eliminación en un ABB

Caso 2: Suprimir un nodo con sólo hijo izquierdo

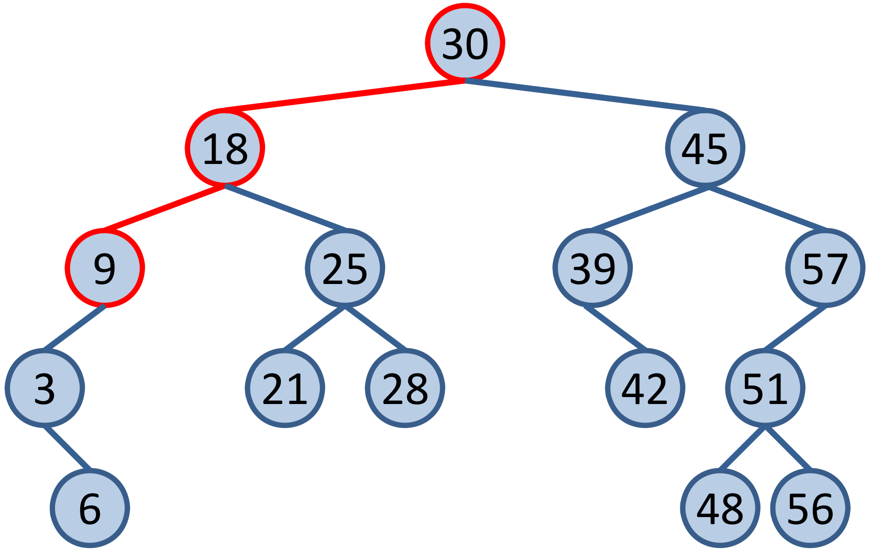
Suprimir 9



Eliminación en un ABB

Caso 2: Suprimir un nodo con sólo hijo izquierdo

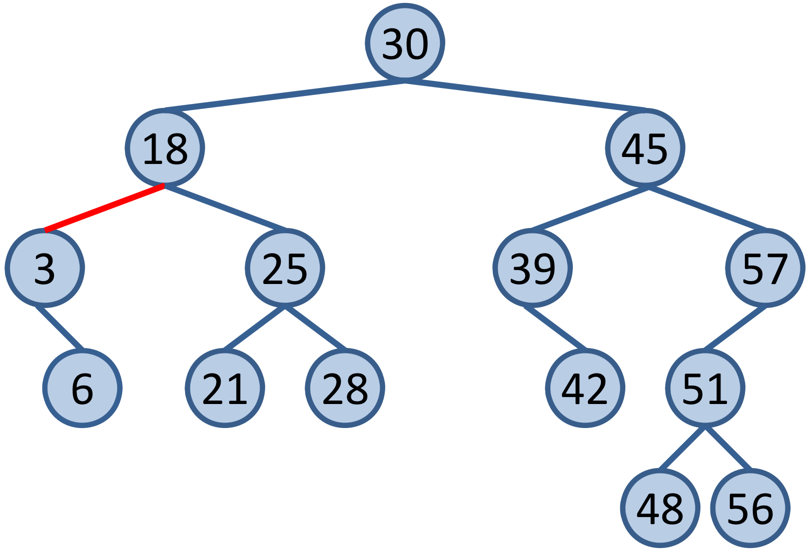
Suprimir 9



Eliminación en un ABB

Caso 2: Suprimir un nodo con sólo hijo izquierdo

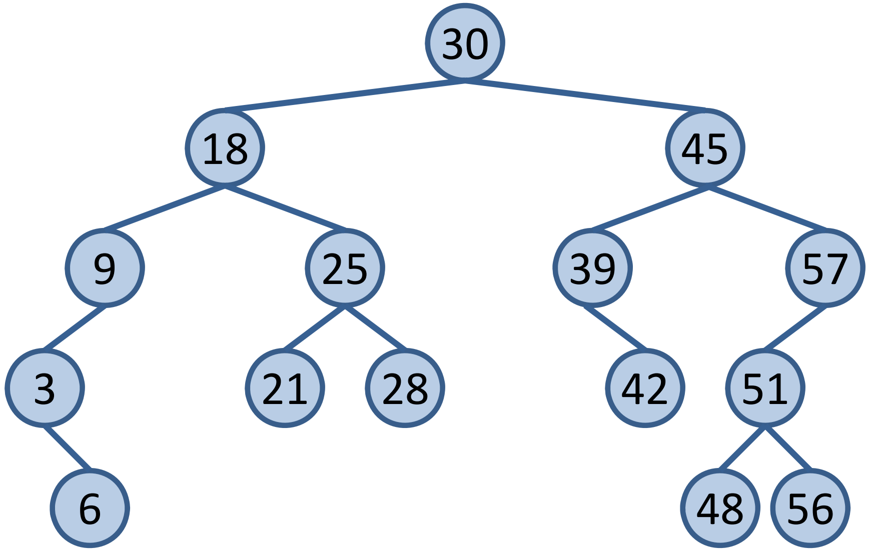
Suprimir 9



Eliminación en un ABB

Caso 3: Suprimir un nodo con sólo hijo derecho

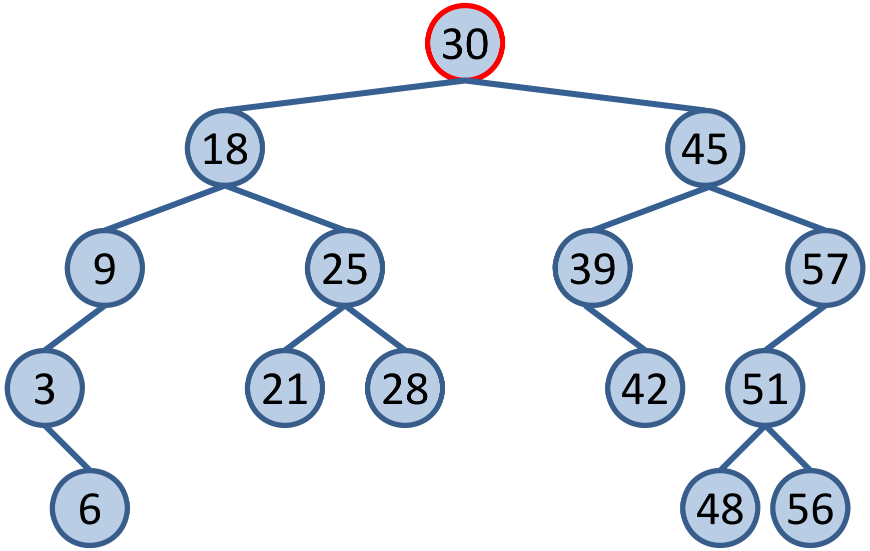
Suprimir 39



Eliminación en un ABB

Caso 3: Suprimir un nodo con sólo hijo derecho

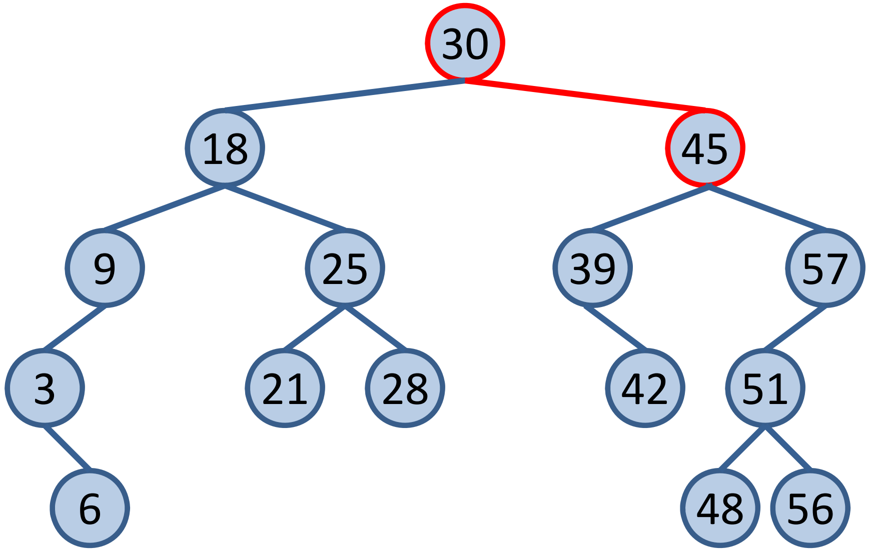
Suprimir 39



Eliminación en un ABB

Caso 3: Suprimir un nodo con sólo hijo derecho

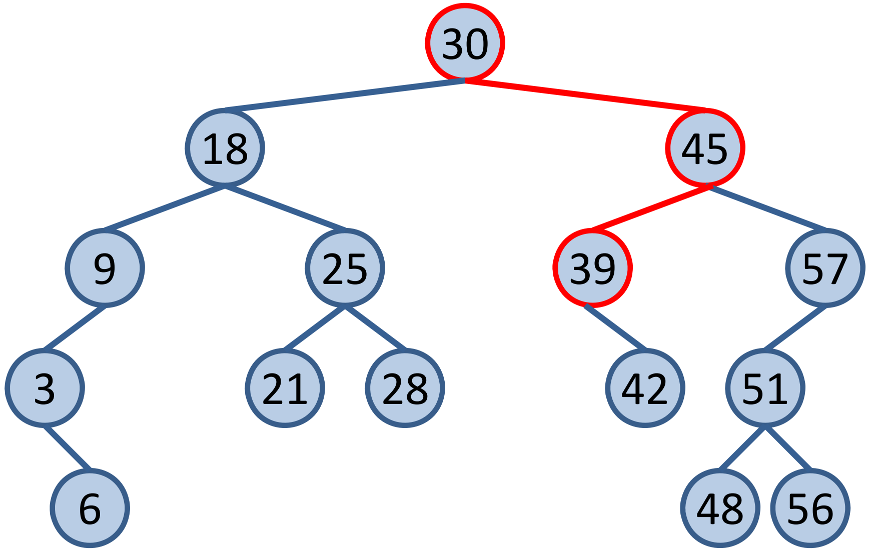
Suprimir 39



Eliminación en un ABB

Caso 3: Suprimir un nodo con sólo hijo derecho

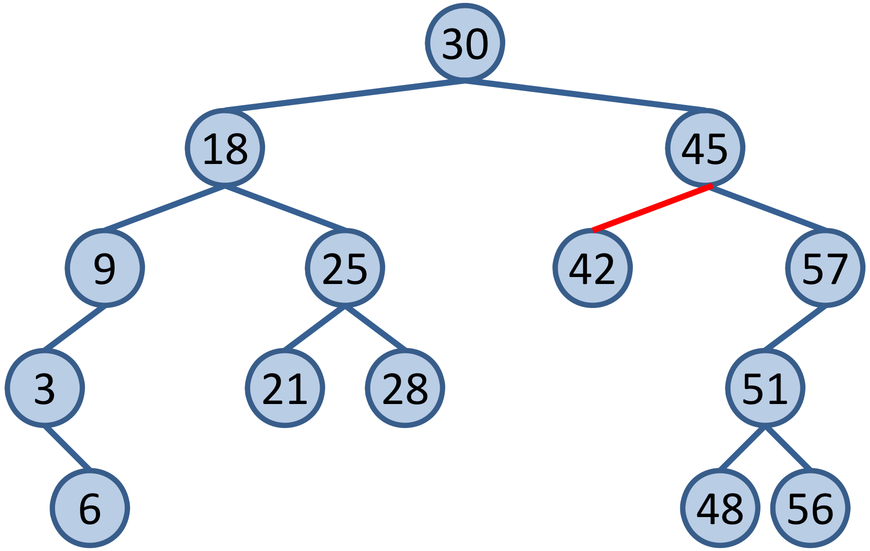
Suprimir 39



Eliminación en un ABB

Caso 3: Suprimir un nodo con sólo hijo derecho

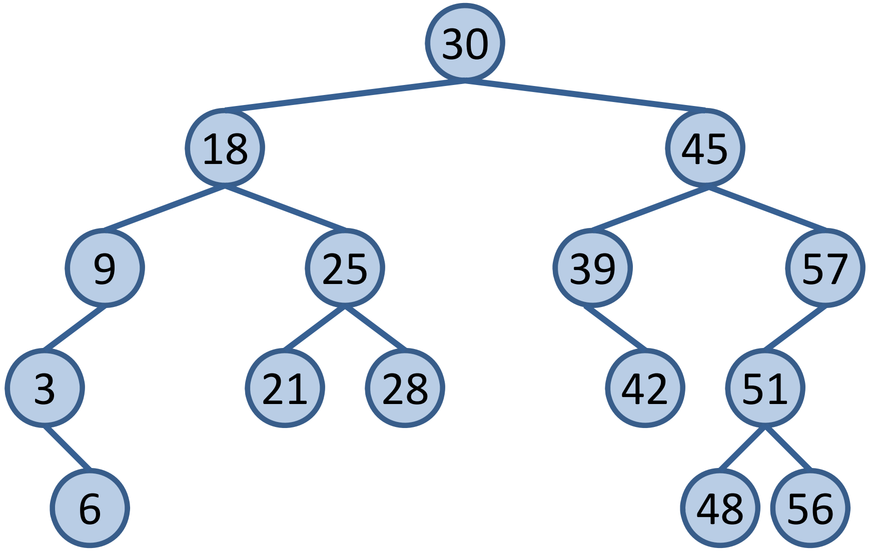
Suprimir 39



Eliminación en un ABB

Caso 4: Suprimir un nodo con dos hijos

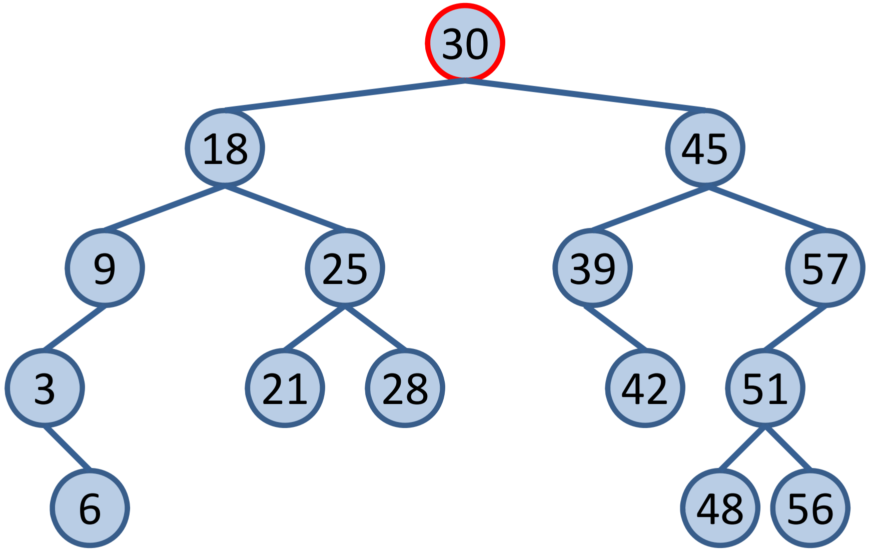
Suprimir 30



Eliminación en un ABB

Caso 4: Suprimir un nodo con dos hijos

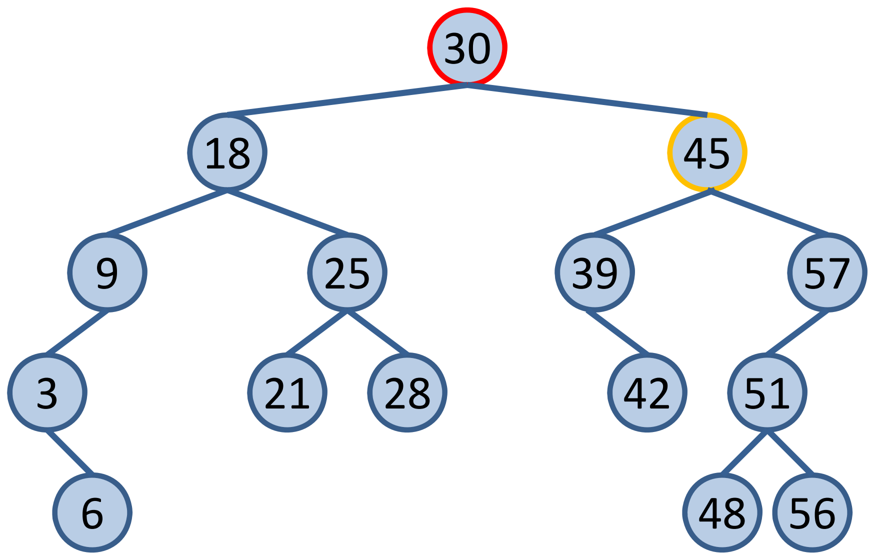
Suprimir 30



Eliminación en un ABB

Caso 4: Suprimir un nodo con dos hijos

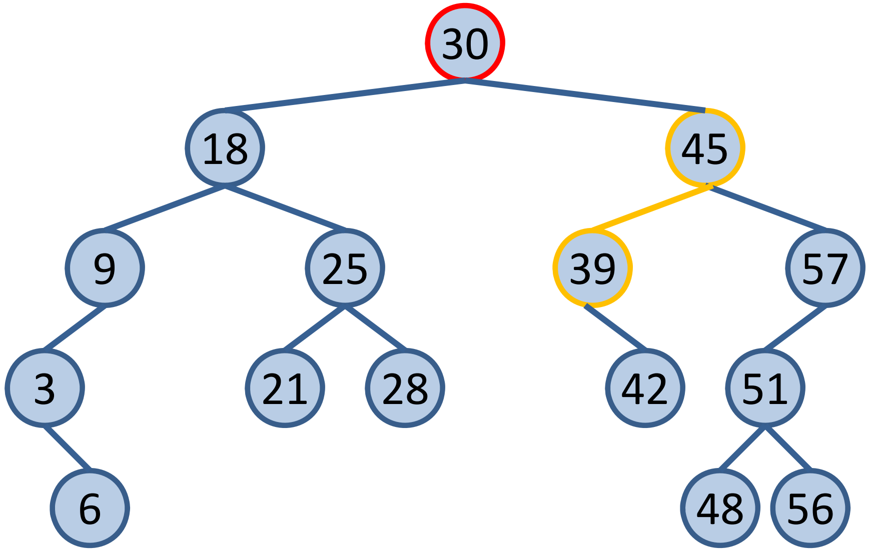
Suprimir 30



Eliminación en un ABB

Caso 4: Suprimir un nodo con dos hijos

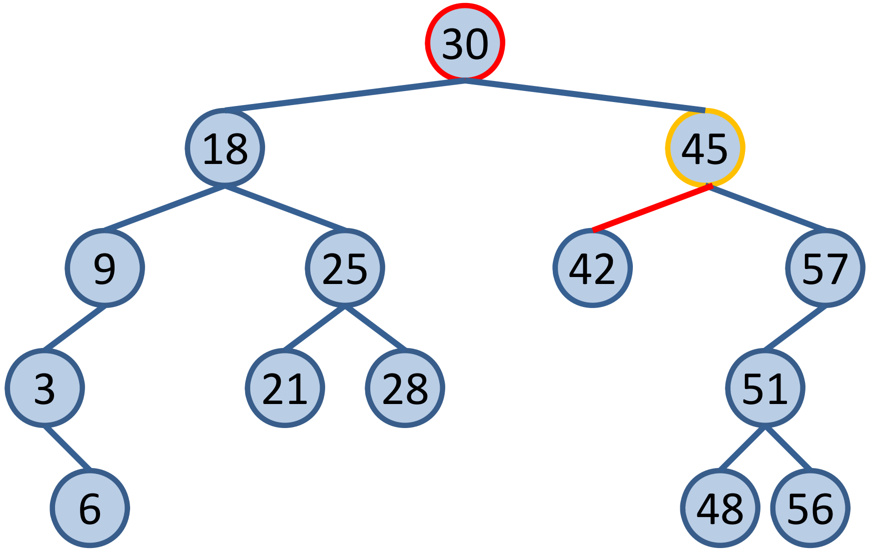
Suprimir 30



Eliminación en un ABB

Caso 4: Suprimir un nodo con dos hijos

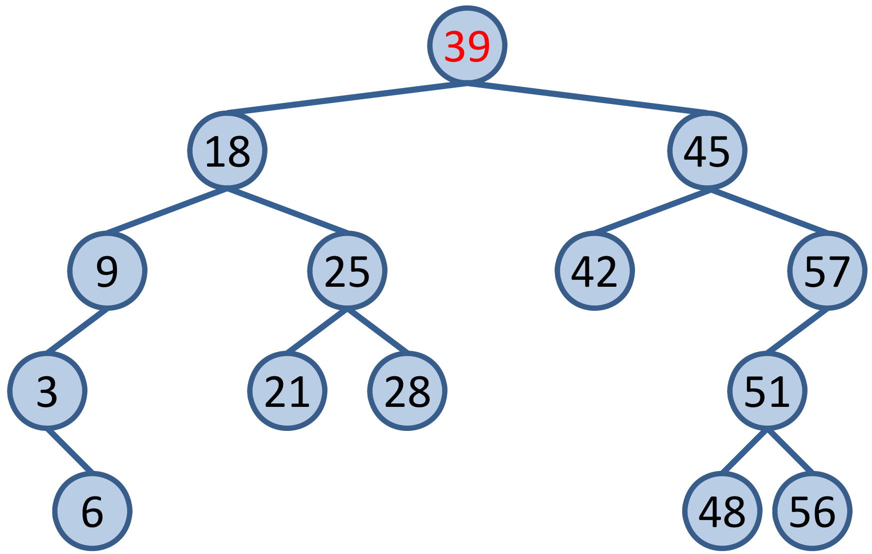
Suprimir 30



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
```

Estructuras de Datos no Lineales

1.6. Árboles binarios de búsqueda equilibrados

José Fidel Argudo Argudo
José Antonio Alonso de la Huerta
Mª Teresa García Horcajadas

Universidad de Cádiz

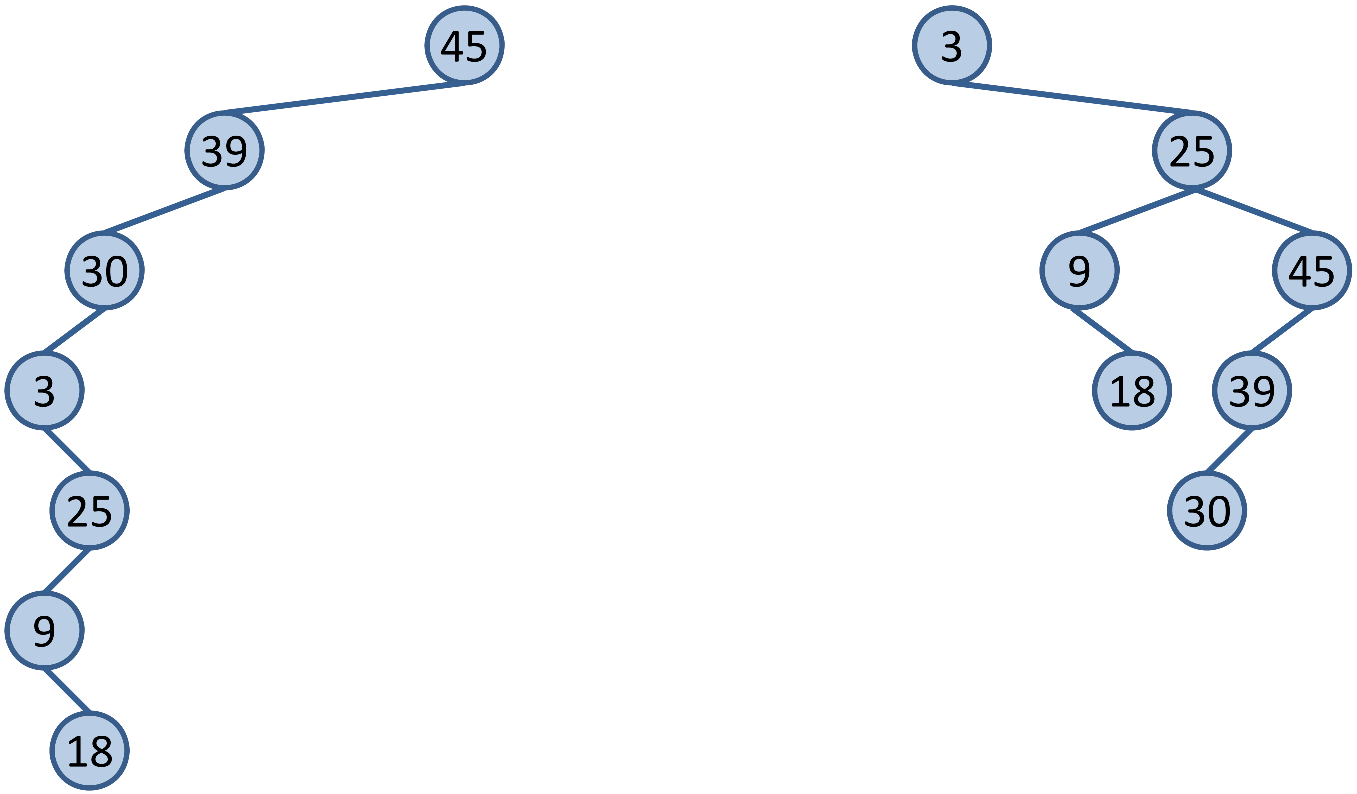
Versión 2.0

Desequilibrio de un ABB

El orden de inserción de los elementos en un ABB determina el grado de equilibrio del árbol.

45, 39, 30, 3, 25, 9, 18

3, 25, 45, 39, 9, 30, 18

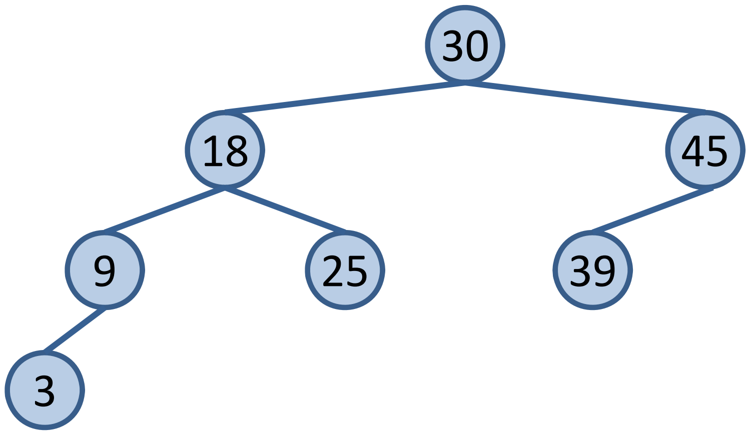


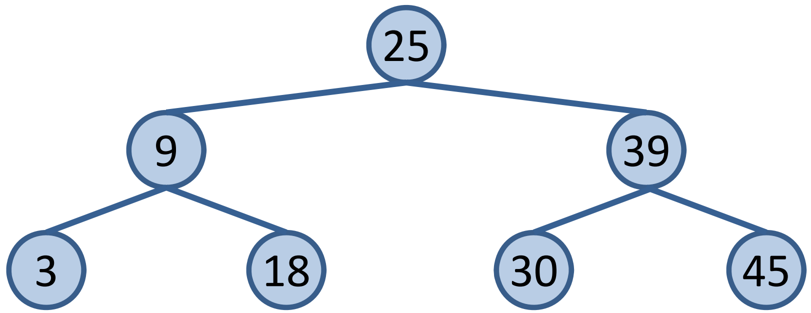
Desequilibrio de un ABB

El orden de inserción de los elementos en un ABB determina el grado de equilibrio del árbol.

30, 45, 39, 18, 9, 25, 3

25, 9, 18, 39, 30, 3, 45





Desequilibrio de un ABB

- Las sucesivas inserciones y eliminaciones en un ABB pueden alterar el grado de equilibrio del árbol.
- El tiempo de las operaciones sobre un ABB (búsqueda, inserción y eliminación) depende del grado de equilibrio del árbol y puede llegar a ser *O(n)* en el caso más desfavorable (árbol degenerado en una lista).
- Para garantizar un tiempo proporcional a la mínima altura posible, o sea $O(\log_2 n)$, es necesario, después de cada operación modificadora, mantener el árbol tan equilibrado como sea posible.

ABB equilibrado (AVL)

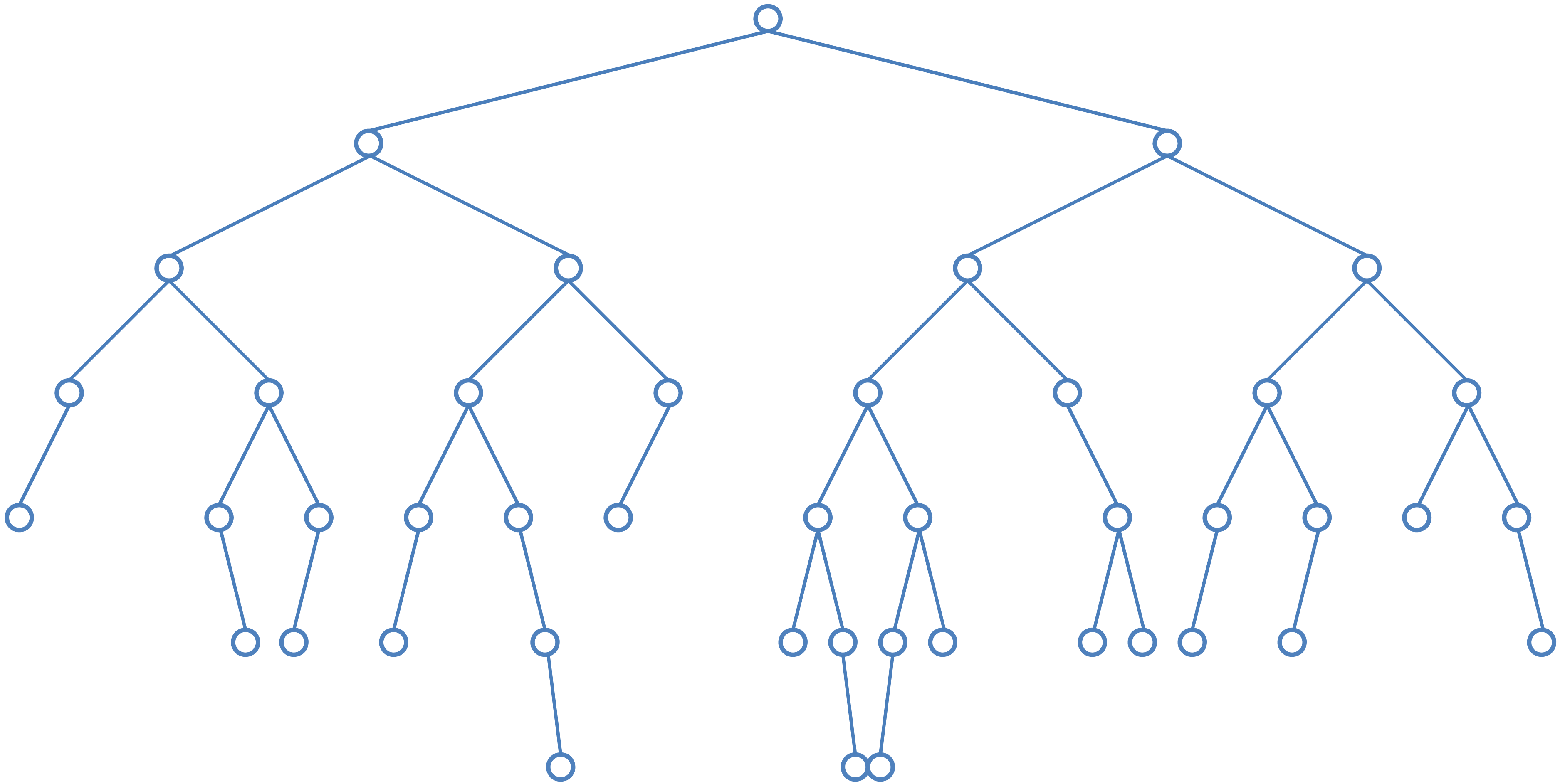
Factor de equilibrio de un nodo: Altura del subárbol derecho menos altura del subárbol izquierdo del nodo.

Árbol binario equilibrado: Aquél en el que el factor de equilibrio de todos los nodos es -1, 0 ó 1.

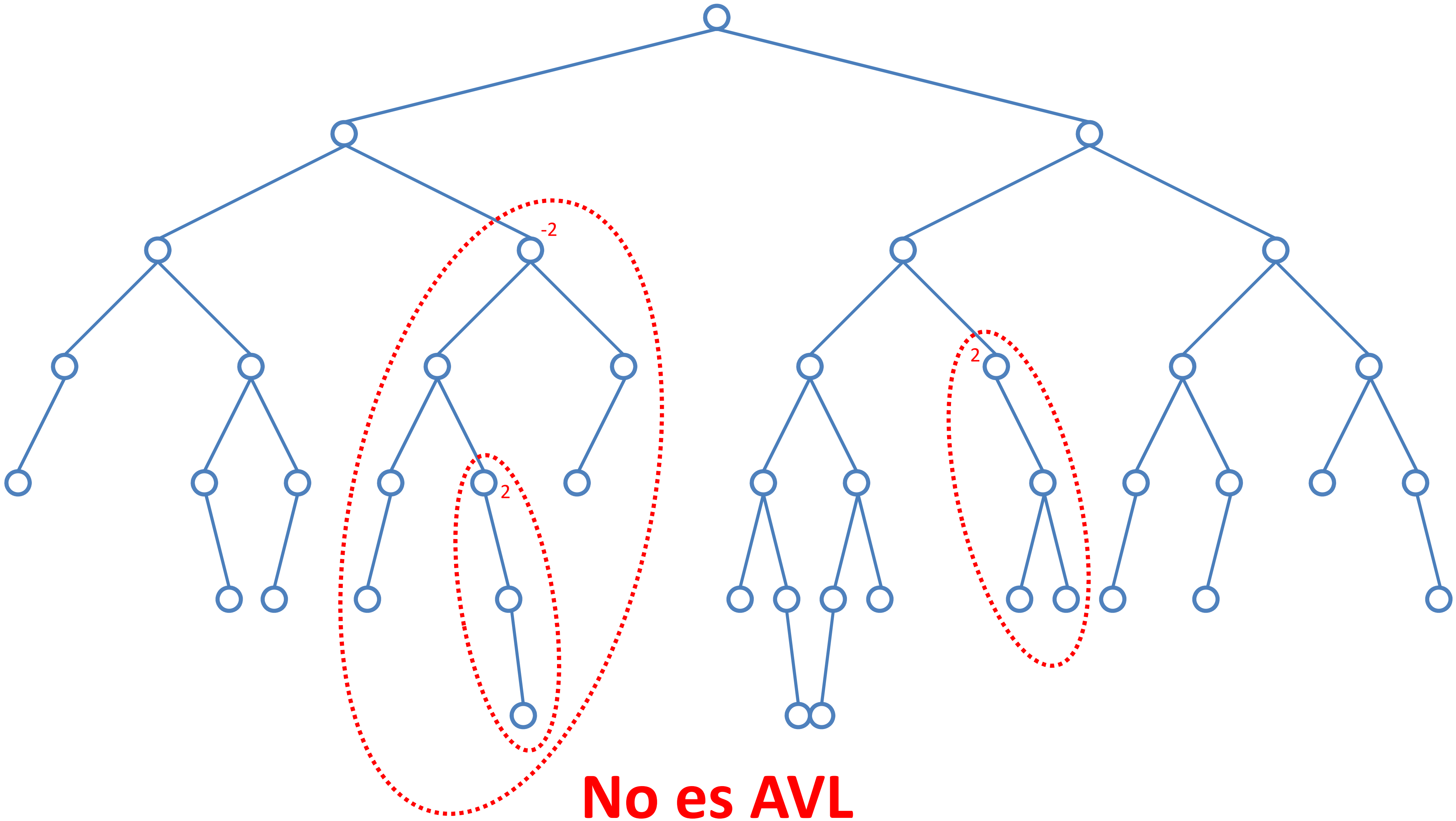
Árbol AVL (**A**delson-**V**elskii & **L**andis, 1962): Árbol binario de búsqueda equilibrado.

- La condición de equilibrio garantiza que la altura de un AVL es de orden logarítmico.
- Los algoritmos de inserción y eliminación en un AVL pueden verificar la propiedad de equilibrio del árbol (y reequilibrarlo, si es necesario, mediante rotaciones de sus nodos que conservan la propiedad de orden/búsqueda) en un tiempo proporcional a su altura.
- En consecuencia los tiempos de búsqueda, inserción y eliminación en un árbol AVL están en *O(log n)*, en el peor caso.

Suponiendo que los valores almacenados en los nodos cumplen la propiedad de orden de búsqueda, ¿este árbol es un AVL?

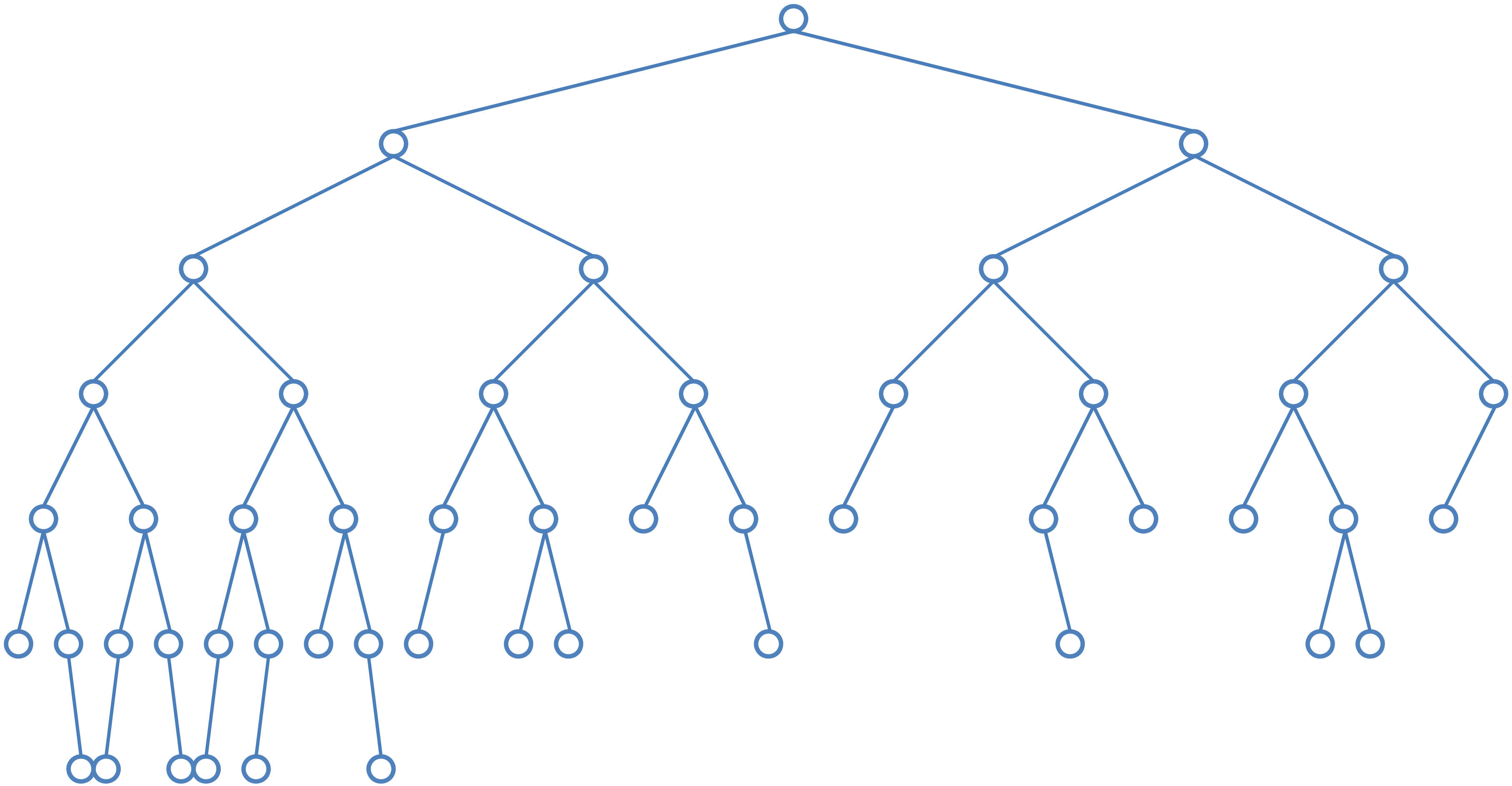


Suponiendo que los valores almacenados en los nodos cumplen la propiedad de orden de búsqueda, ¿este árbol es un AVL?

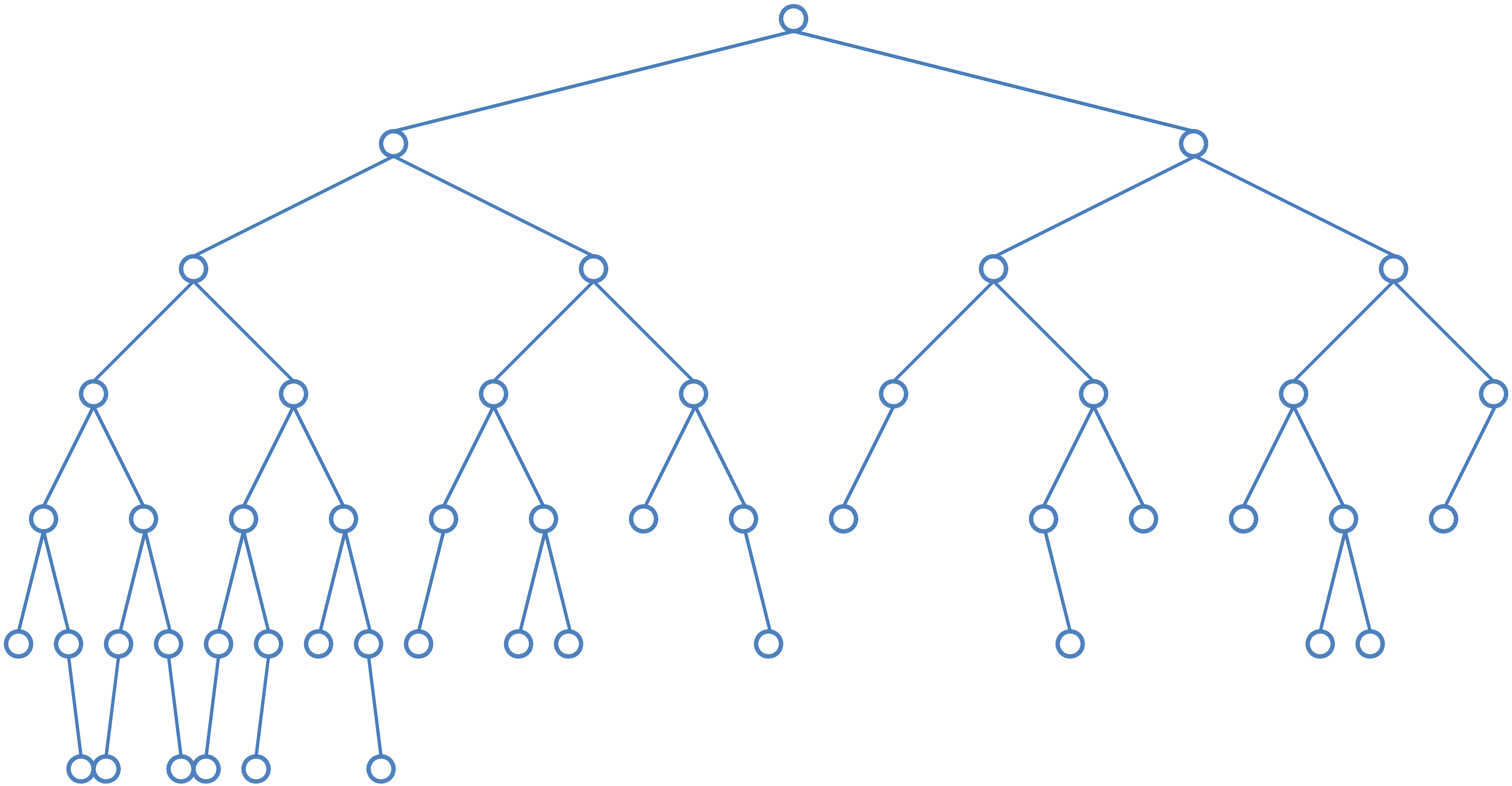


No es AVL

Suponiendo que los valores almacenados en los nodos cumplen la propiedad de orden de búsqueda, ¿este árbol es un AVL?

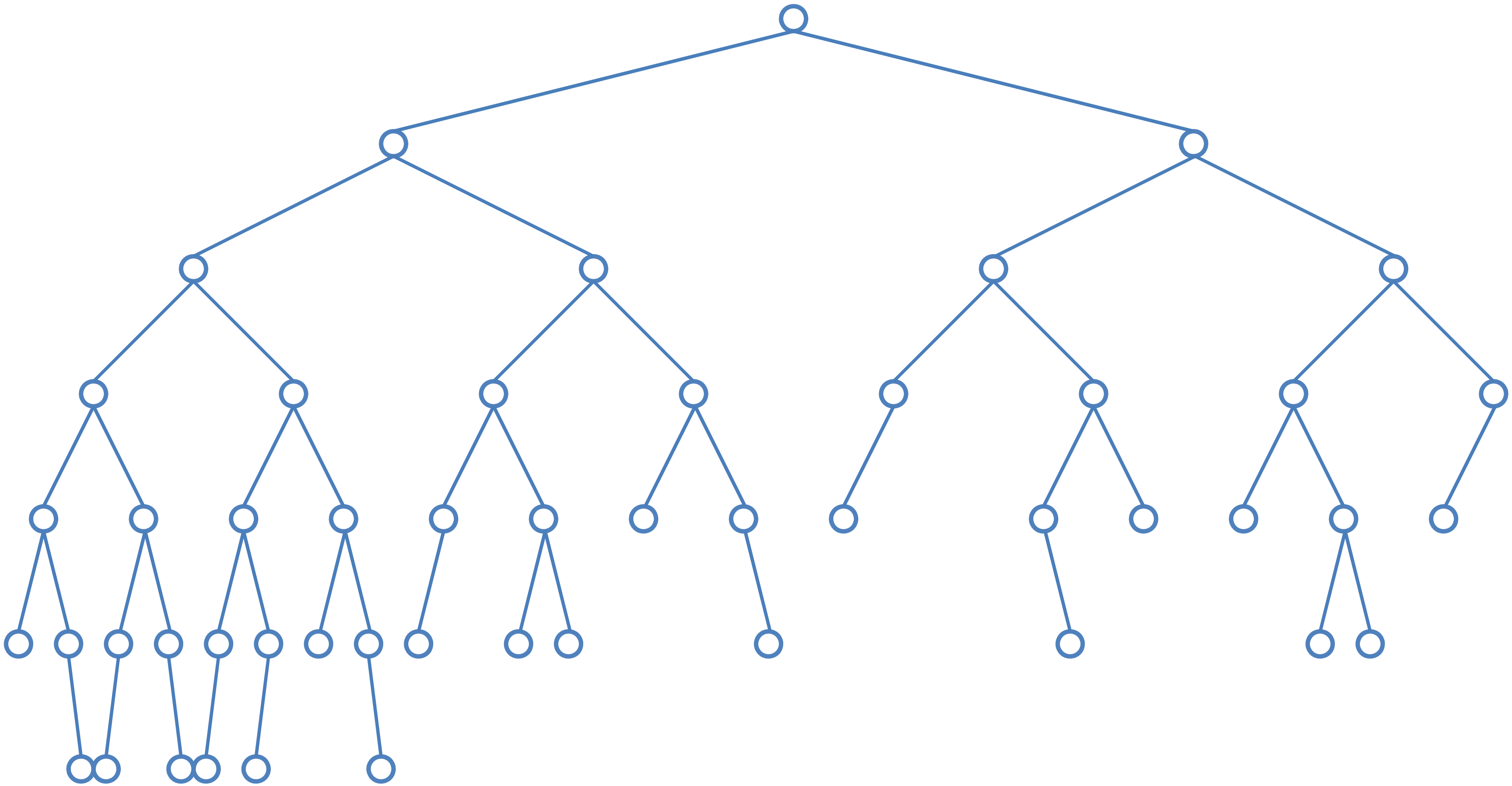


Suponiendo que los valores almacenados en los nodos cumplen la propiedad de orden de búsqueda, ¿este árbol es un AVL?



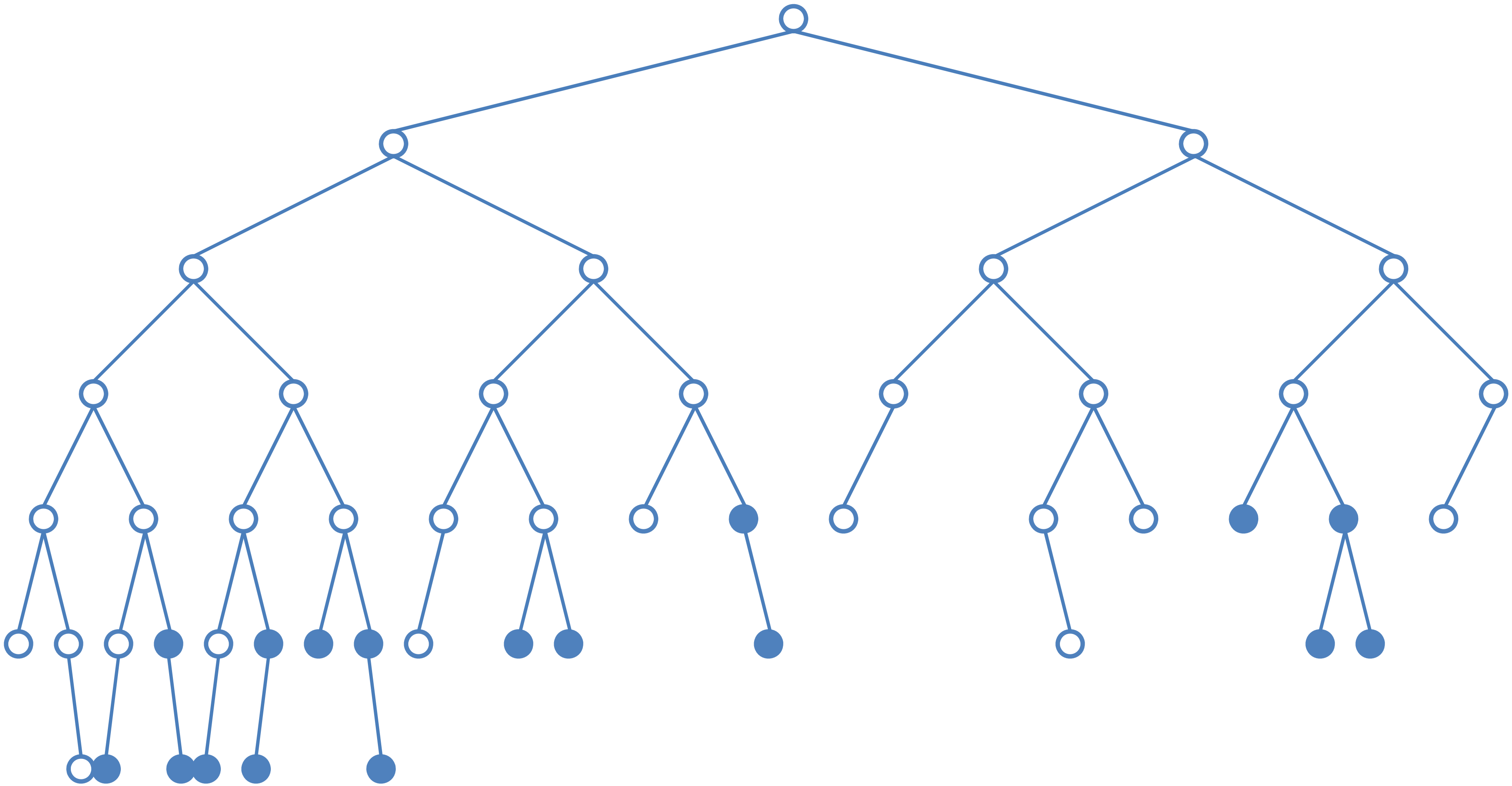
Es AVL

¿Cuáles son los nodos que se pueden suprimir sin que el árbol pierda el equilibrio y manteniendo su altura?



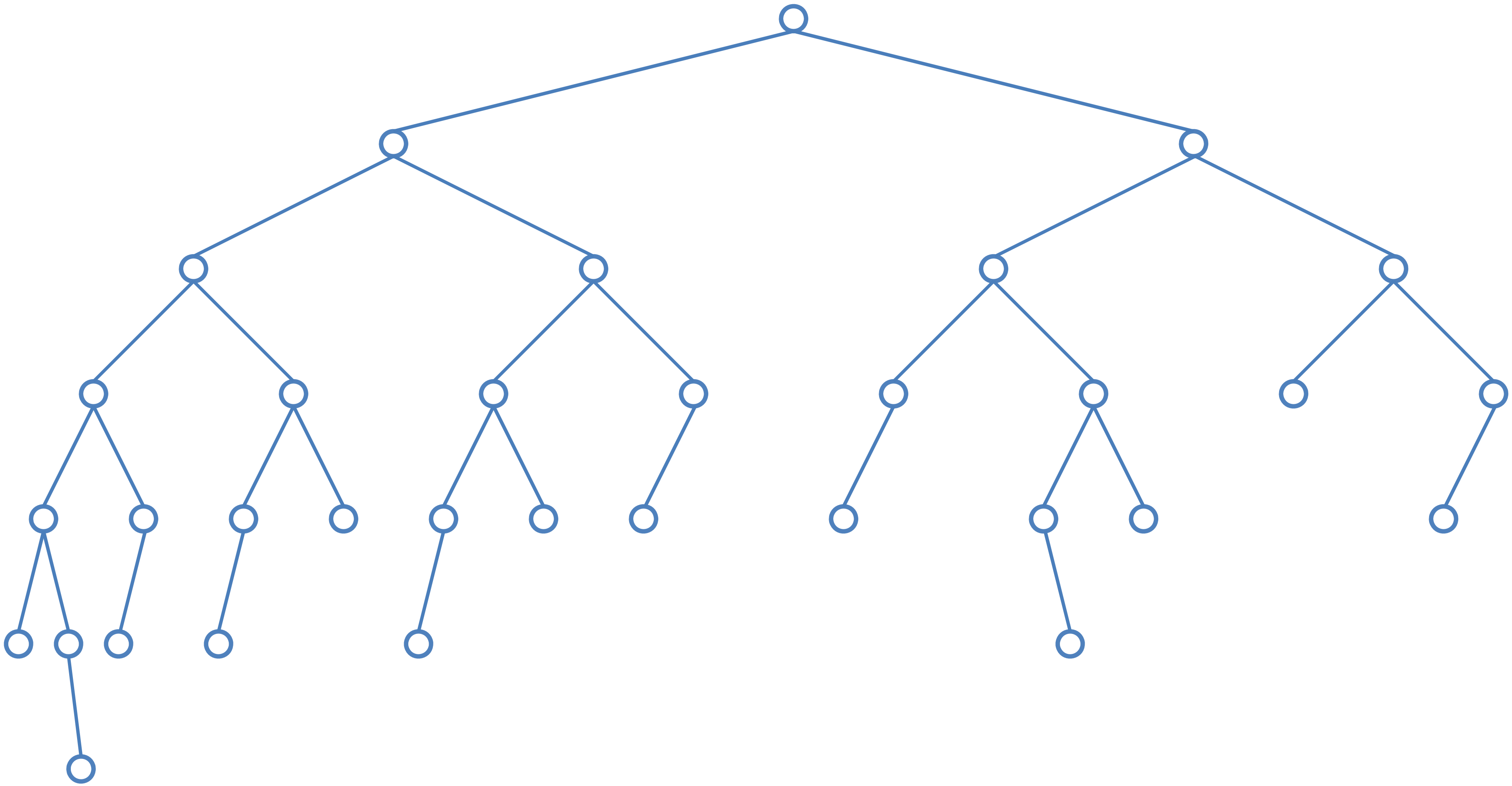
Es AVL

¿Cuáles son los nodos que se pueden suprimir sin que el árbol pierda el equilibrio y manteniendo su altura?



Es AVL

¿Cuáles son los nodos que se pueden suprimir sin que el árbol pierda el equilibrio y manteniendo su altura?



Es AVL