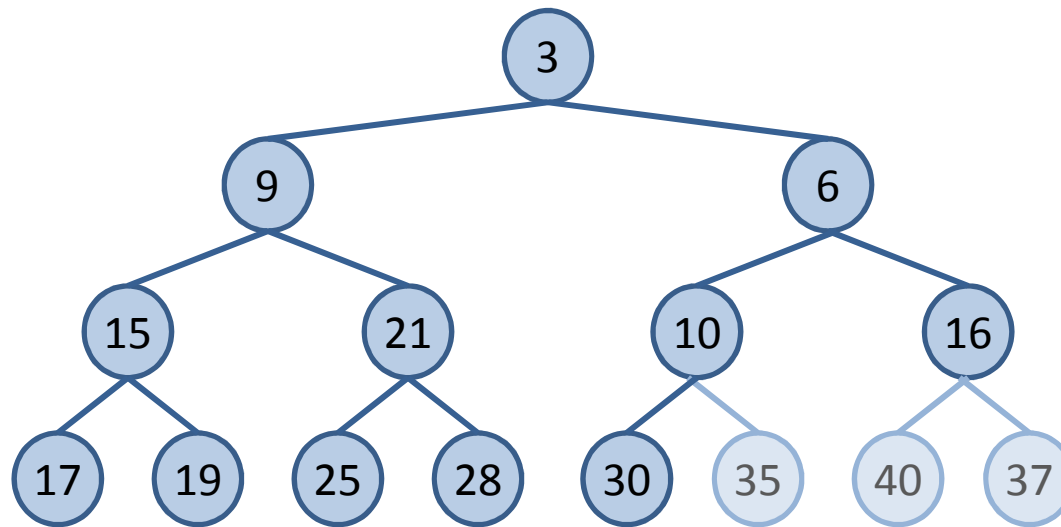


Árboles parcialmente ordenados

- Un **árbol completo** es un árbol con todos sus niveles llenos, con la posible excepción del nivel más bajo, al cuál sólo le pueden faltar nodos por la derecha.

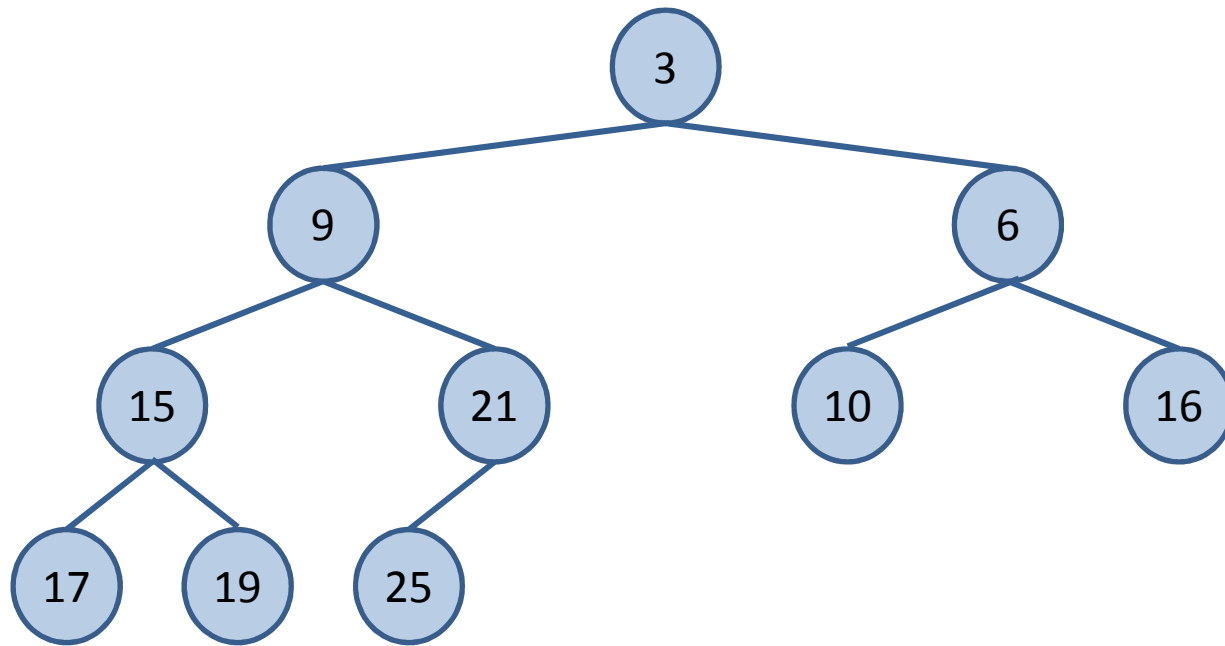


- Un árbol binario completo de altura h tiene entre $1 + \sum_{i=0}^{h-1} 2^i = 2^h$ y $\sum_{i=0}^h 2^i = 2^{h+1} - 1$ nodos. Esto implica que un árbol binario completo de n nodos tiene una altura $h = \log_2 n$.

Árboles parcialmente ordenados

- Un **árbol parcialmente ordenado** (montículo) es un **árbol completo** en el que **el valor en cualquier nodo es menor que el de todos sus descendientes**, o sea, en un APO, para todo nodo X , el valor en el padre de X es menor o **igual** que el valor en X (con excepción de la raíz, que no tiene padre).
- Operaciones básicas: inserción y eliminación.
- La propiedad de completitud implica que la altura menor posible de un APO de n nodos es $h = \log_2 n$.
- Las propiedad de orden de un APO permite efectuar las inserciones y eliminaciones de nodos en un tiempo $O(h)$ en el peor caso.
- En consecuencia, las inserciones y eliminaciones en un APO son $O(\log_2 n)$.
- Principal aplicación: representación de colas con prioridad.

Árboles parcialmente ordenados

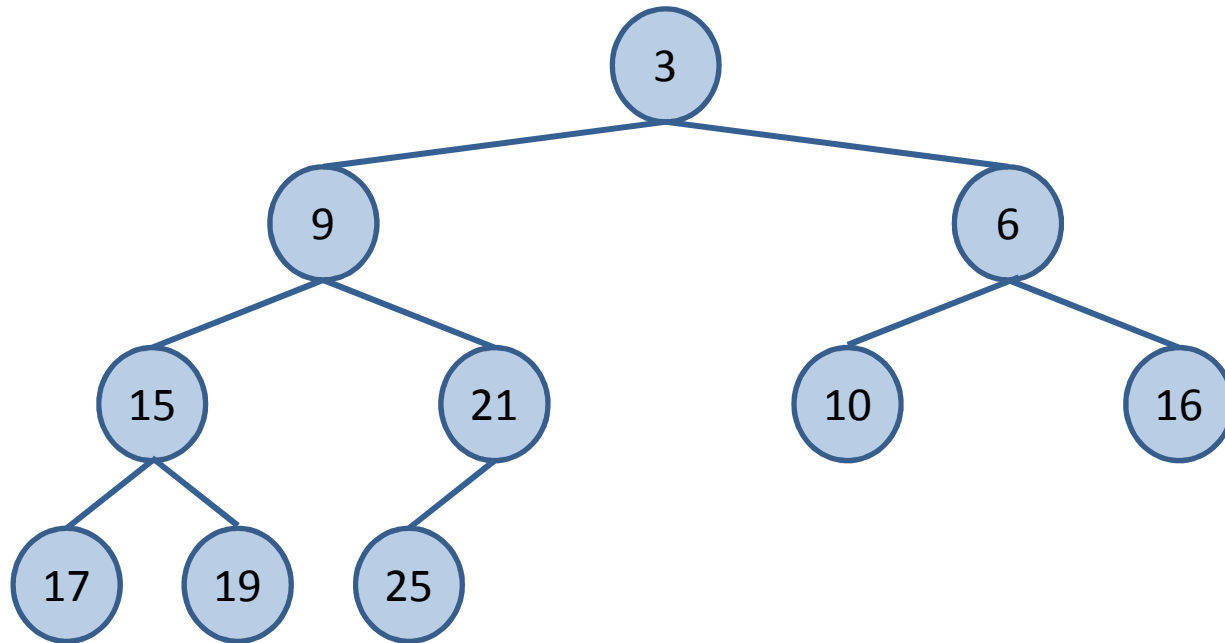


Altura = 3

Número de nodos (n)

$$2^3 \leq n \leq 2^4 - 1$$

Representación de APO



Vector de posiciones relativas

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
nodos	3	9	6	15	21	10	16	17	19	25					

maxNodos = 15

ultimo = 9

```

#ifndef APO_H
#define APO_H
#include <cassert>

template <typename T> class Apo {
public:
    explicit Apo(size_t maxNodos);           // constructor
    void insertar(const T& e);
    void suprimir();
    const T& cima() const;
    bool vacio() const;
    Apo(const Apo<T>& A);                     // ctor. de copia
    Apo<T>& operator =(const Apo<T>& A);      // asignación de apo
    ~Apo();                                   // destructor
private:
    typedef int nodo; // índice del vector
                        // entre 0 y maxNodos-1
    T* nodos;          // vector de nodos
    int maxNodos;      // tamaño del vector
    nodo ultimo;       // último nodo del árbol

```

```
nodo padre(nodo i) const { return (i-1)/2; }  
nodo hIzq(nodo i)  const { return 2*i+1; }  
nodo hDer(nodo i)  const { return 2*i+2; }  
void flotar(nodo i);  
void hundir(nodo i);  
};
```

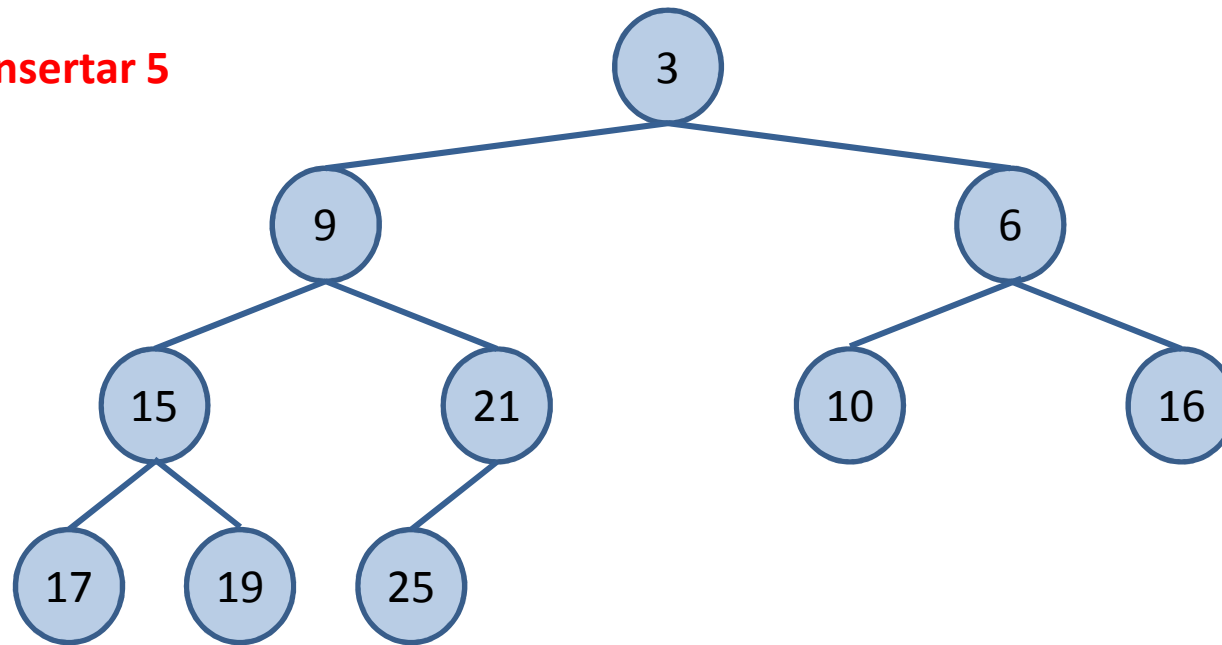
```
template <typename T>
inline Apo<T>::Apo(size_t maxNodos) :
    nodos(new T[maxNodos]),
    maxNodos(maxNodos),
    ultimo(-1)    // Apo vacío
{}

template <typename T>
inline const T& Apo<T>::cima() const
{
    assert(ultimo > -1);    // Apo no vacío
    return nodos[0];
}

template <typename T>
inline bool Apo<T>::vacio() const
{
    return (ultimo == -1);
}
```

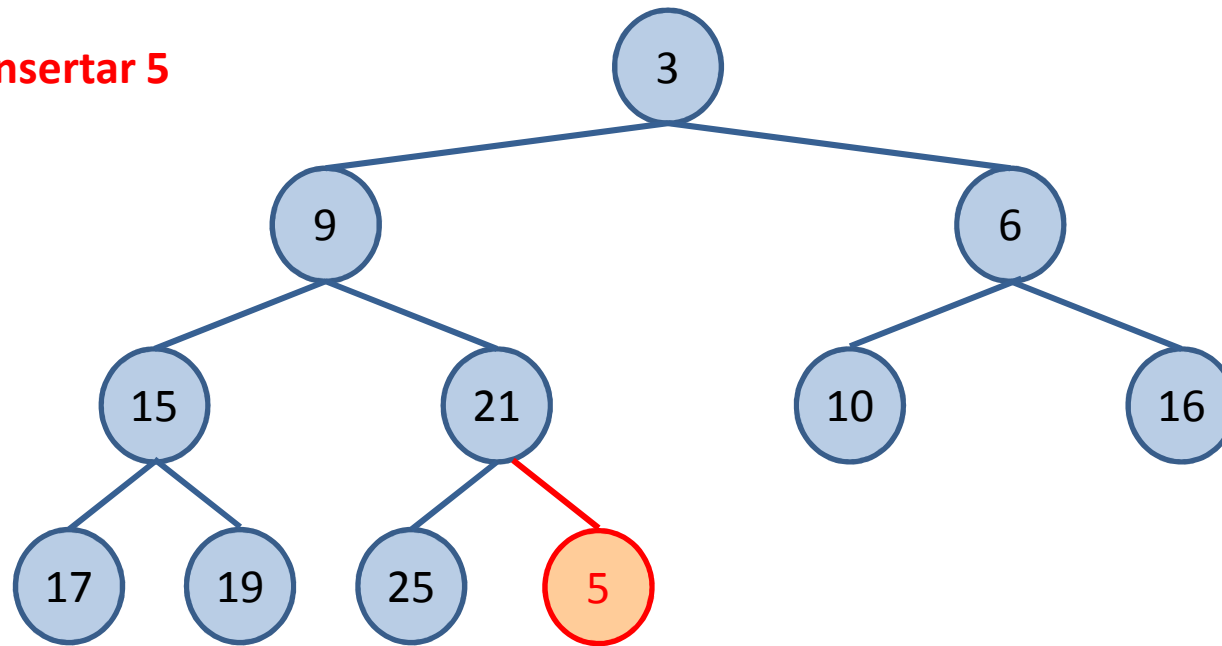
Insertión en un APO

Insertar 5



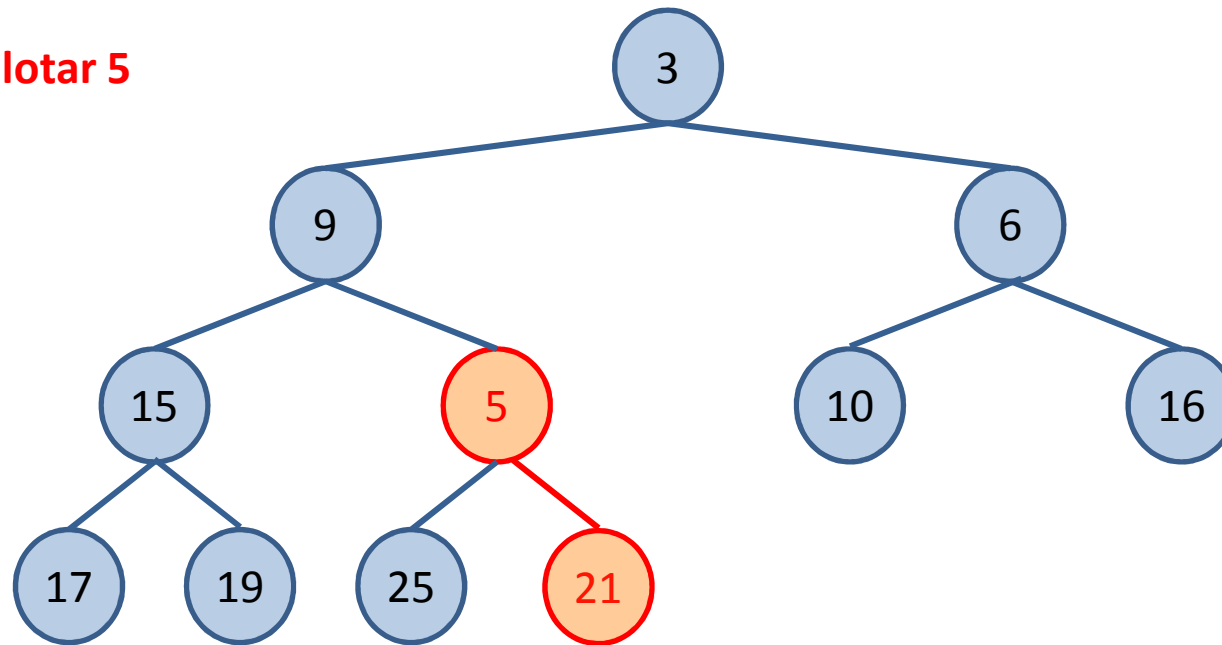
Insertión en un APO

Insertar 5



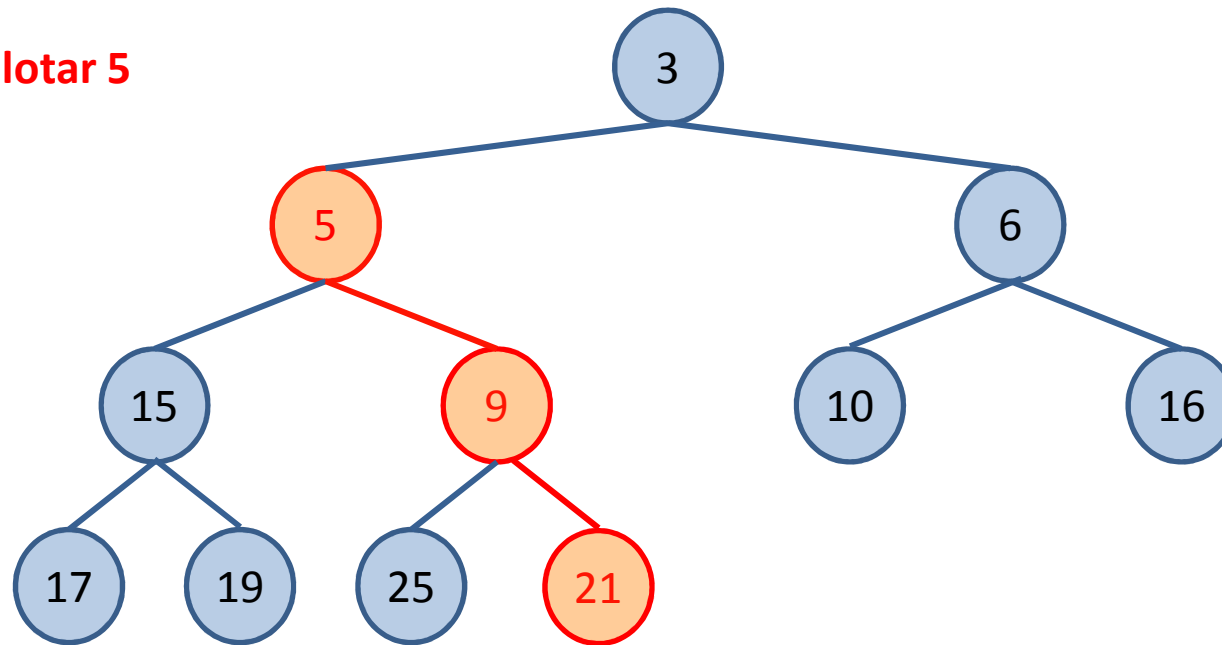
Inserción en un APO

Flotar 5



Inserción en un APO

Flotar 5



```
template <typename T>
inline void Apo<T>::insertar(const T& e)
{
    assert(ultimo < maxNodos-1);    // Apo no lleno
    nodos[++ultimo] = e;
    if (ultimo > 0)
        flotar(ultimo);    // Reordenar
}
```

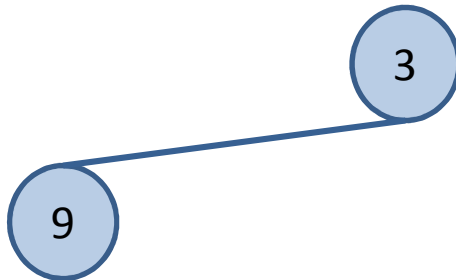
```
template <typename T>
void Apo<T>::flotar(nodo i)
{
    T e = nodos[i];
    while (i > 0 && e < nodos[padre(i)])
    {
        nodos[i] = nodos[padre(i)];
        i = padre(i);
    }
    nodos[i] = e;
}
```

Suprimir en un APO

Caso 1: Un nodo

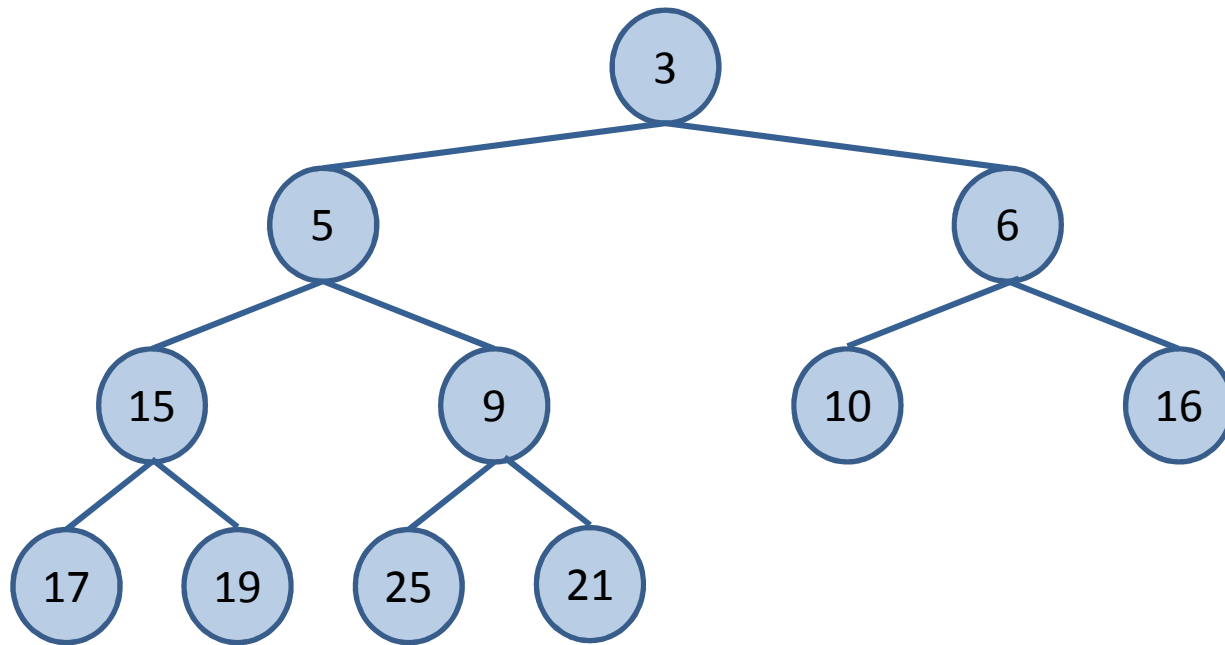


Caso 2: Dos nodos



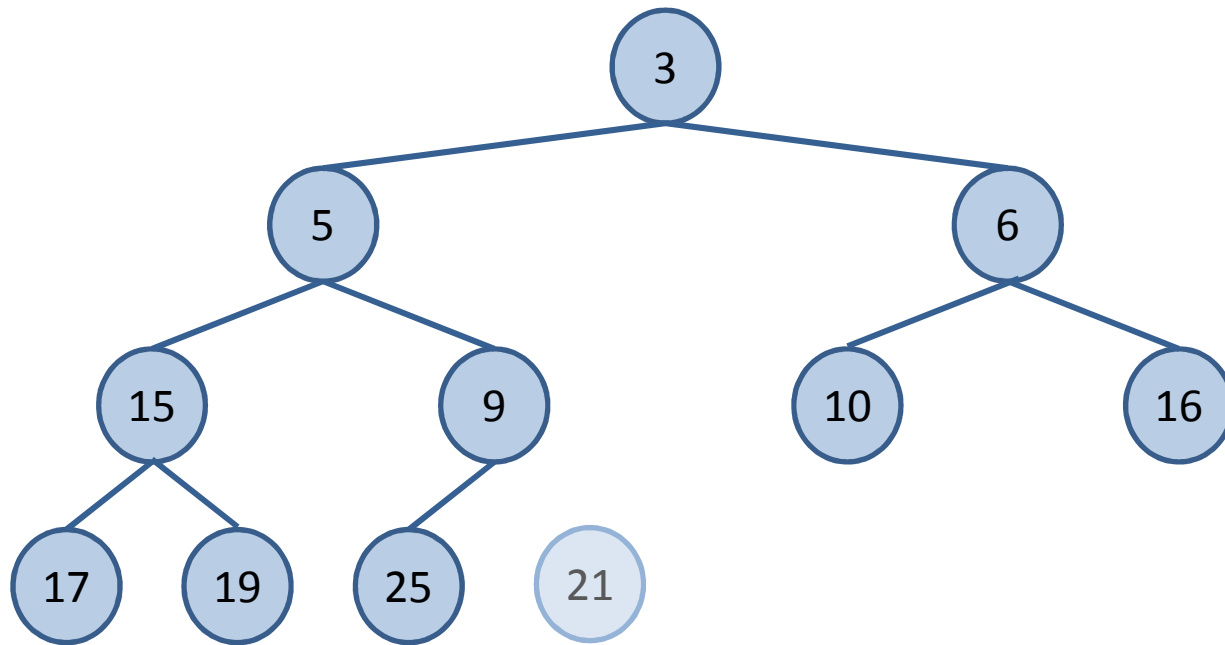
Suprimir en un APO

Caso 3: Más de dos nodos



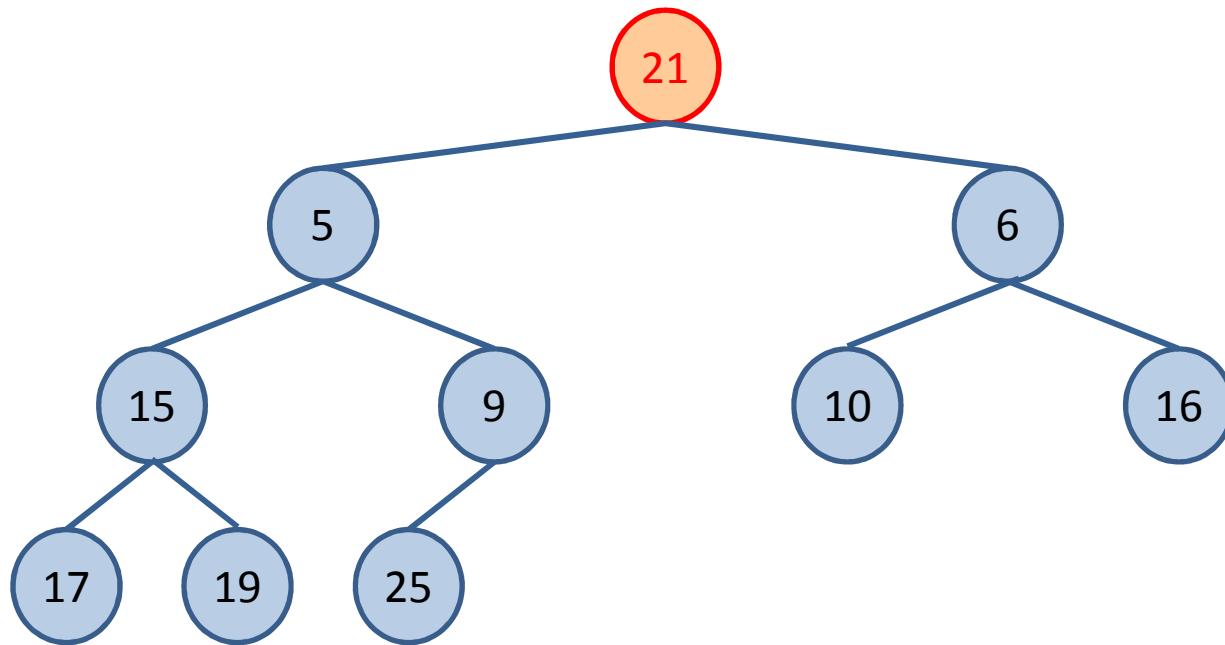
Suprimir en un APO

Caso 3: Más de dos nodos



Suprimir en un APO

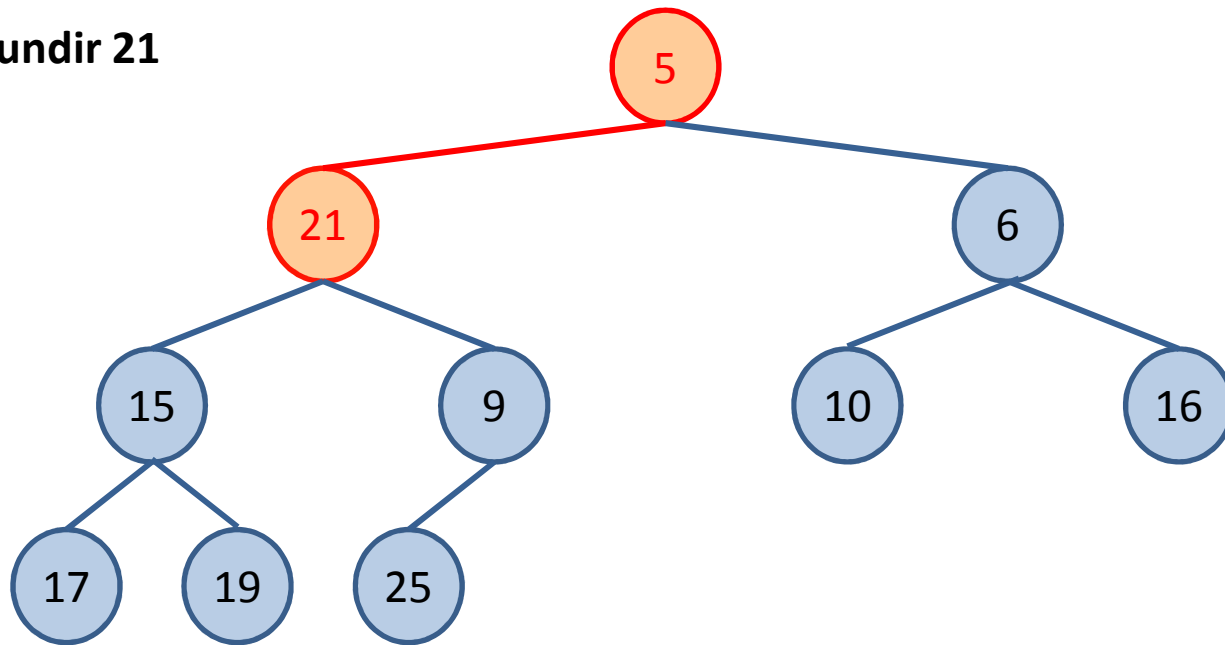
Caso 3: Más de dos nodos



Suprimir en un APO

Caso 3: Más de dos nodos

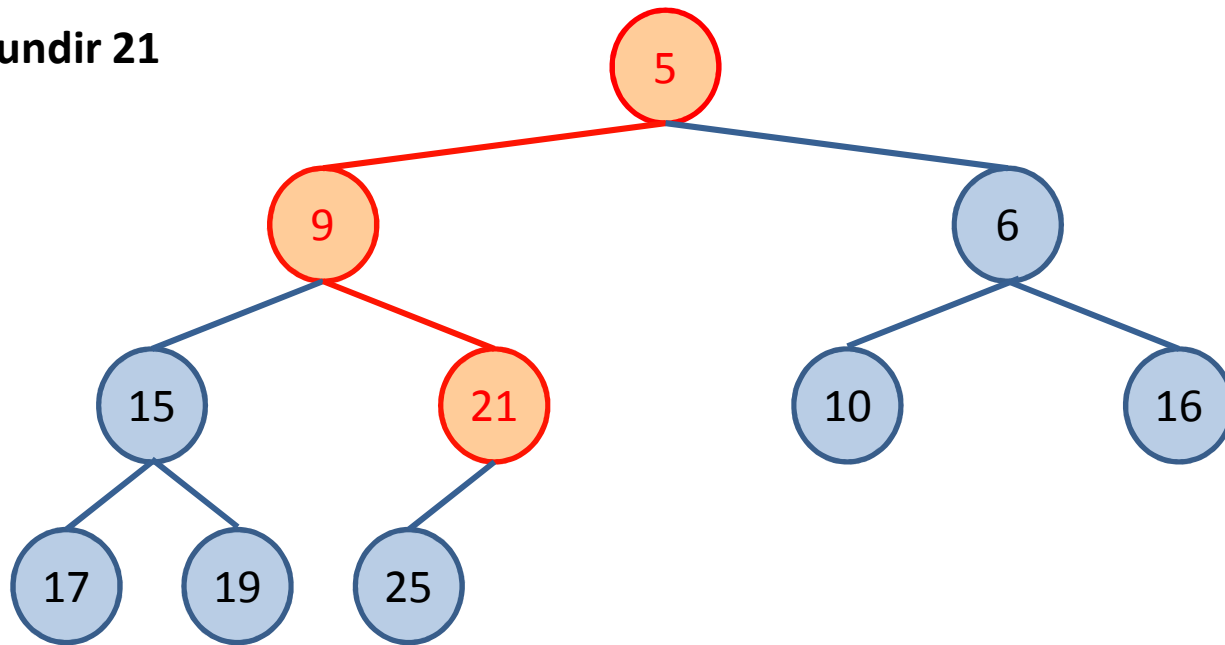
Hundir 21



Suprimir en un APO

Caso 3: Más de dos nodos

Hundir 21



```
template <typename T>
inline void Apo<T>::suprimir()
{
    assert(ultimo > -1);    // Apo no vacío
    if (--ultimo > -1)      // Apo no queda vacío
    {
        nodos[0] = nodos[ultimo+1];
        if (ultimo > 0)     // Quedan dos o más elementos.
            hundir(0);      // Reordenar
    }
}
```

```

template <typename T>
void Apo<T>::hundir(nodo i)
{
    bool fin = false;
    T e = nodos[i];
    while (hIzq(i) <= ultimo && !fin)    // hundir e
    {
        nodo hMin;    // hijo menor del nodo i
        if (hIzq(i) < ultimo && nodos[hDer(i)] < nodos[hIzq(i)])
            hMin = hDer(i);
        else
            hMin = hIzq(i);
        if (nodos[hMin] < e) { // subir el hijo menor
            nodos[i] = nodos[hMin];
            i = hMin;
        }
        else    // e <= hijos
            fin = true;
    }
    nodos[i] = e;    // colocar e
}

```

```
template <typename T>
inline Apo<T>::~~Apo()
{
    delete[] nodos;
}
```

```
template <typename T>
Apo<T>::Apo(const Apo<T>& A) :
    nodos(new T[A.maxNodos]),
    maxNodos(A.maxNodos),
    ultimo(A.ultimo)
{
    // copiar el vector
    for (nodo n = 0; n <= ultimo; n++)
        nodos[n] = A.nodos[n];
}
```

```

template <typename T>
Apo<T>& Apo<T>::operator =(const Apo<T>& A)
{
    if (this != &A) {    // evitar autoasignación
        // Destruir el vector y crear uno nuevo si es necesario
        if (maxNodos != A.maxNodos) {
            delete[] nodos;
            maxNodos = A.maxNodos;
            nodos = new T[maxNodos];
        }
        ultimo = A.ultimo;
        // Copiar el vector
        for (nodo n = 0; n <= ultimo; n++)
            nodos[n] = a.nodos[n];
    }
    return *this;
}

#endif // APO_H

```