

Estructuras de Datos no Lineales

1.6. Árboles parcialmente ordenados (montículos)

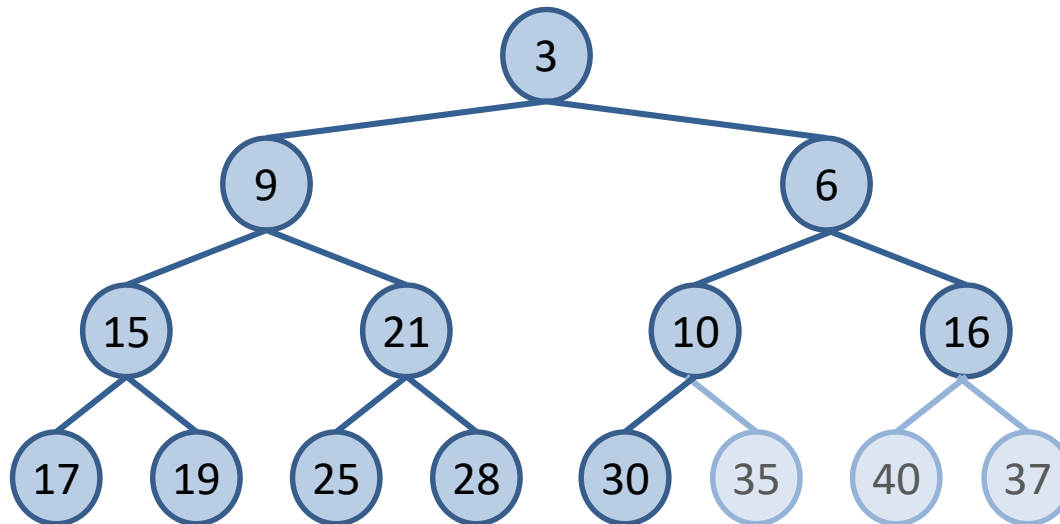
José Fidel Argudo Argudo
José Antonio Alonso de la Huerta
M^a Teresa García Horcajadas



Versión 2.1

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

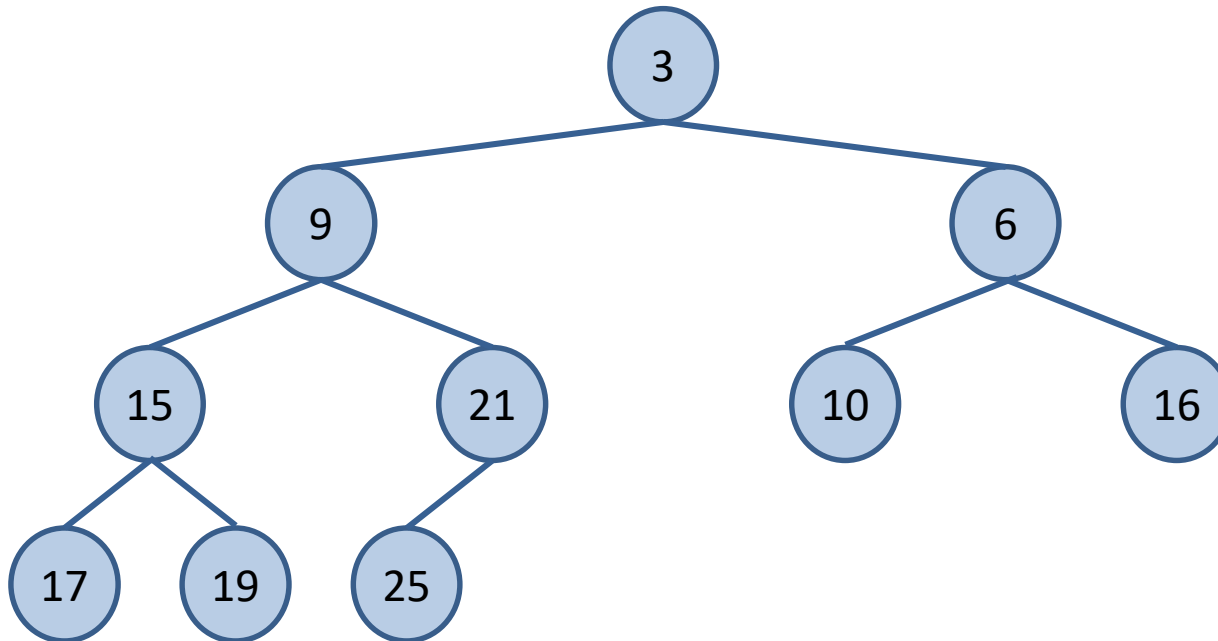
Un **árbol parcialmente ordenado** (o montículo) es un **árbol completo** en el que **el valor de cualquier nodo es menor o igual que el de todos sus descendientes**. Por tanto, para todo nodo x de un APO el valor en el padre de x es menor o igual que el valor en x (con excepción de la raíz, que no tiene padre).

Operaciones básicas: acceso al mínimo, inserción y eliminación

- **El mínimo se puede obtener en un tiempo $O(1)$** , puesto que se encuentra en la raíz.
- La propiedad de completitud implica que la altura menor posible de un APO de n nodos es $h = \log_2 n$.
- La 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**.

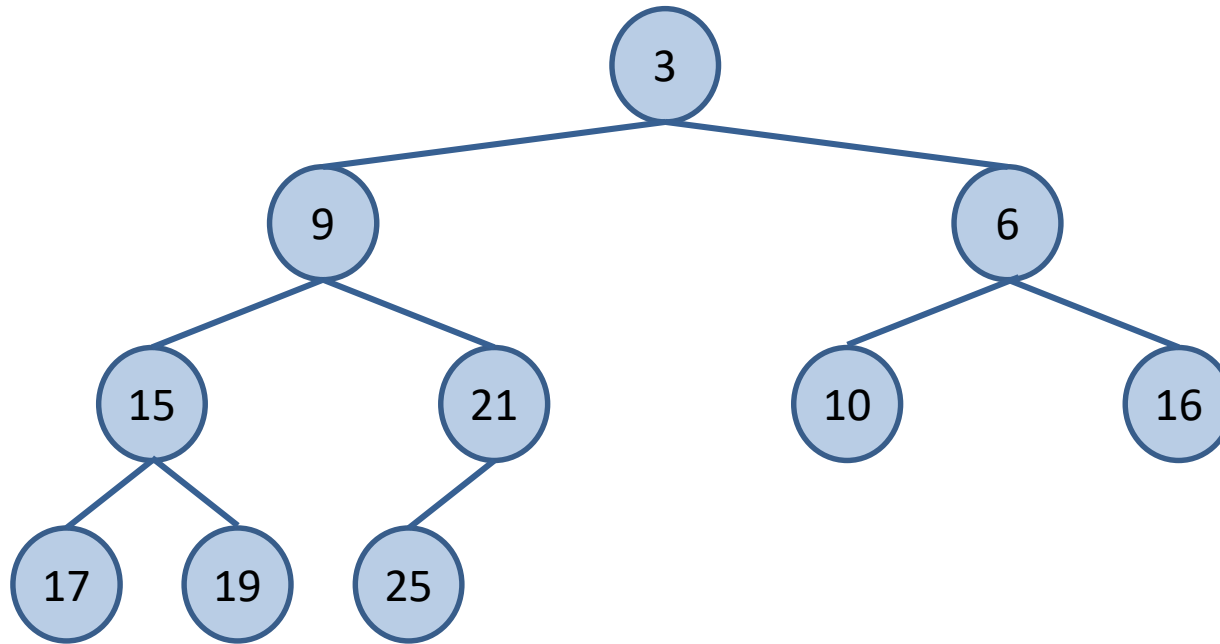
Árboles parcialmente ordenados



Altura = 3

Número de nodos (n): $2^3 \leq n \leq 2^4 - 1$

Representación de un APO mediante un 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

numNodos = 10

```

#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 size_t nodo; // índice del vector
                          // entre 0 y maxNodos-1
    T* nodos;             // vector de nodos
    size_t maxNodos;      // tamaño del vector
    size_t numNodos;      // 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),
    numNodos(0)    // Apo vacío.
{}

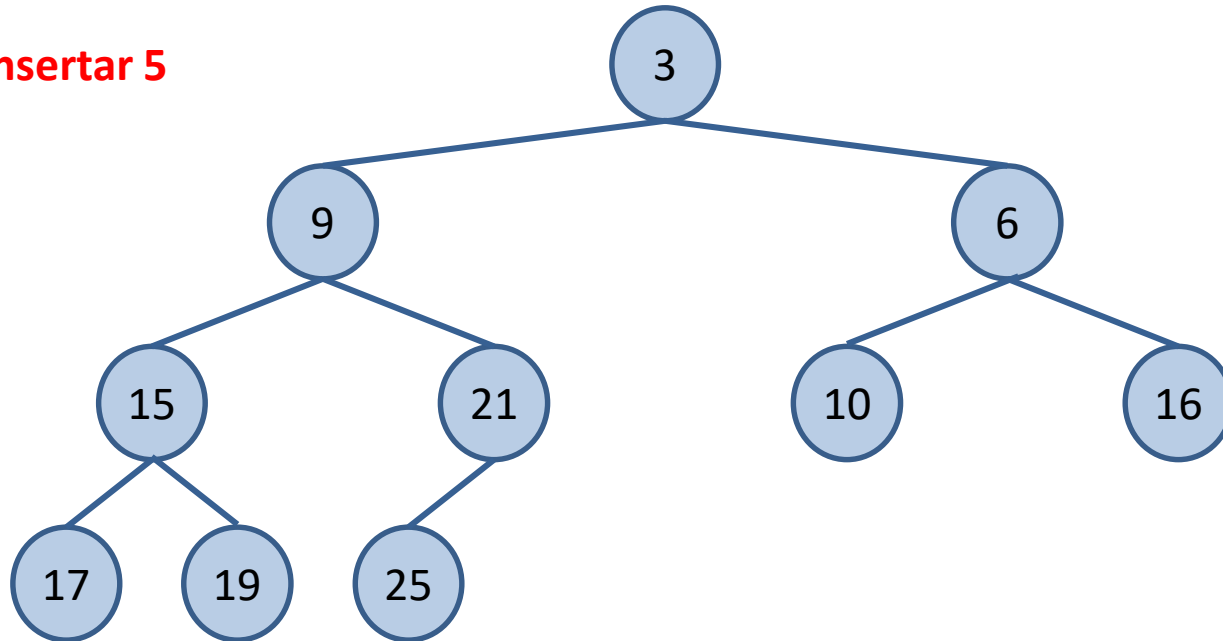
template <typename T>
inline const T& Apo<T>::cima() const
{
    assert(numNodos > 0);    // Apo no vacío.
    return nodos[0];
}

template <typename T>
inline bool Apo<T>::vacio() const
{
    return (numNodos == 0);
}

```


Inserción en un APO

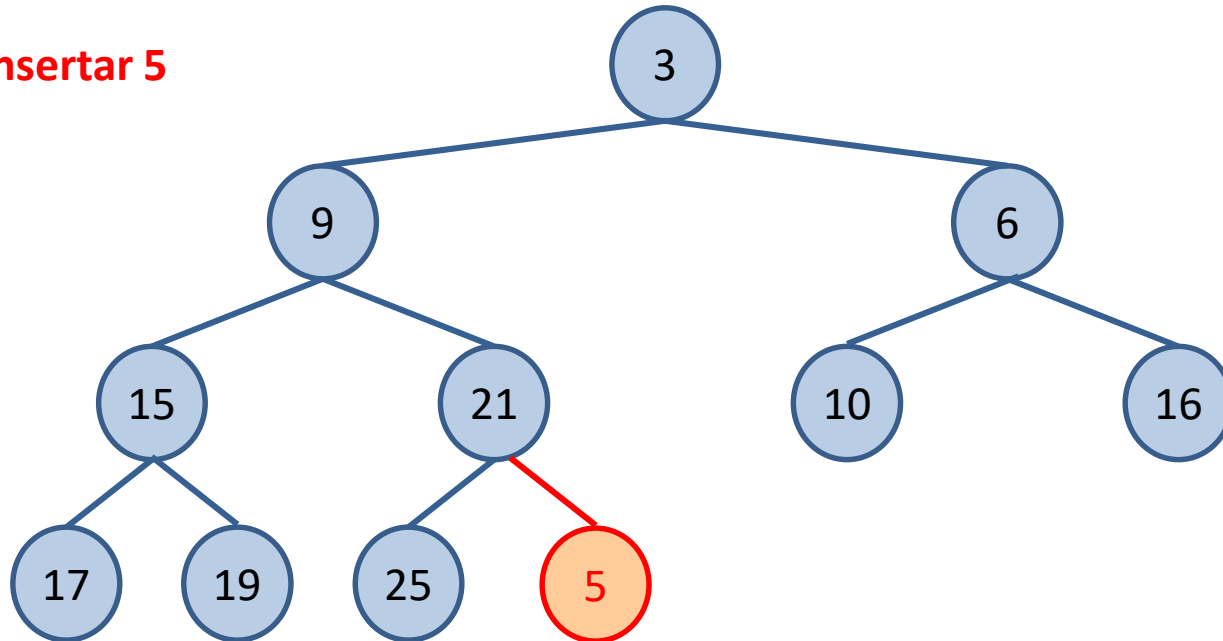
Insertar 5



numNodos = 10

Inserción en un APO

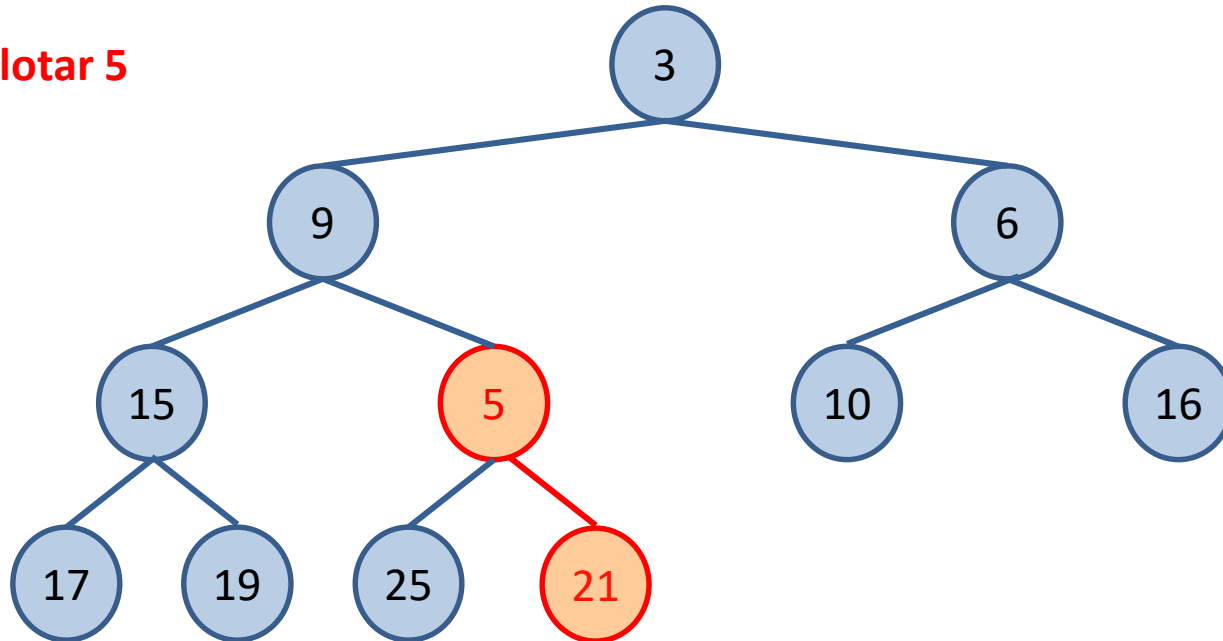
Insertar 5



numNodos = 11

Inserción en un APO

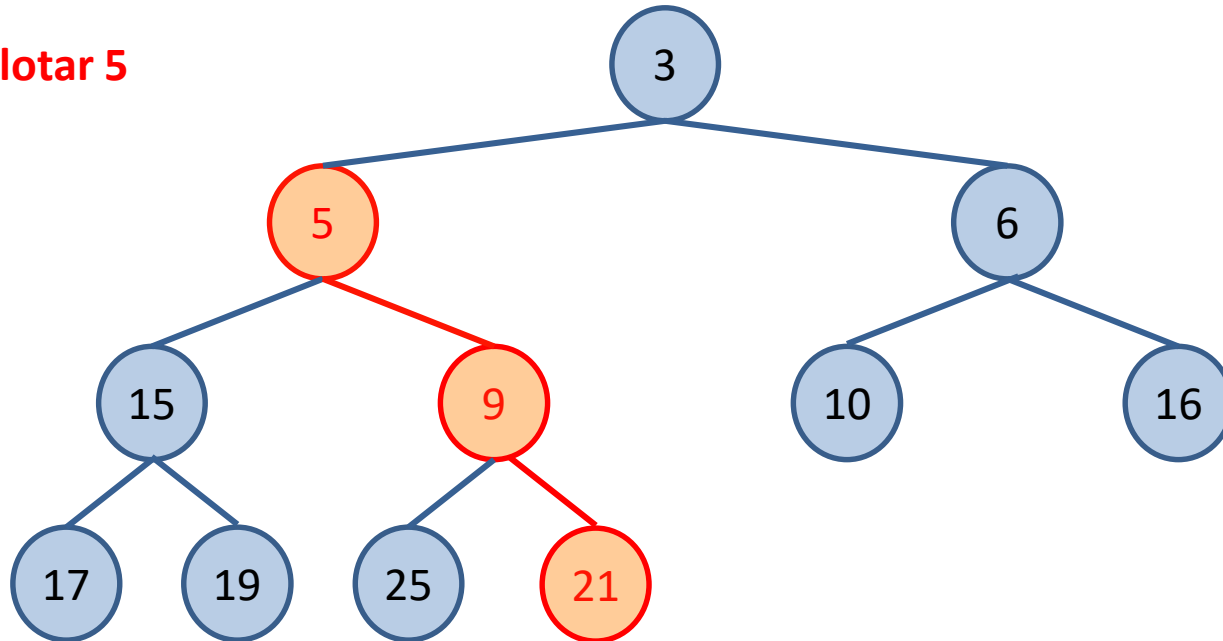
Flotar 5



numNodos = 11

Inserción en un APO

Flotar 5



numNodos = 11

```

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

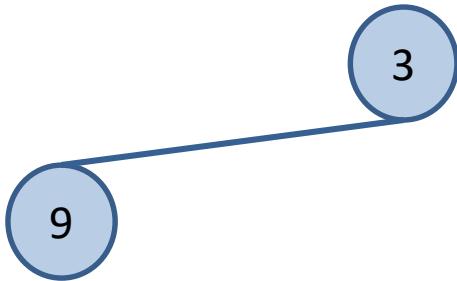


numNodos = 1



numNodos = 0

Caso 2: Dos nodos



numNodos = 2



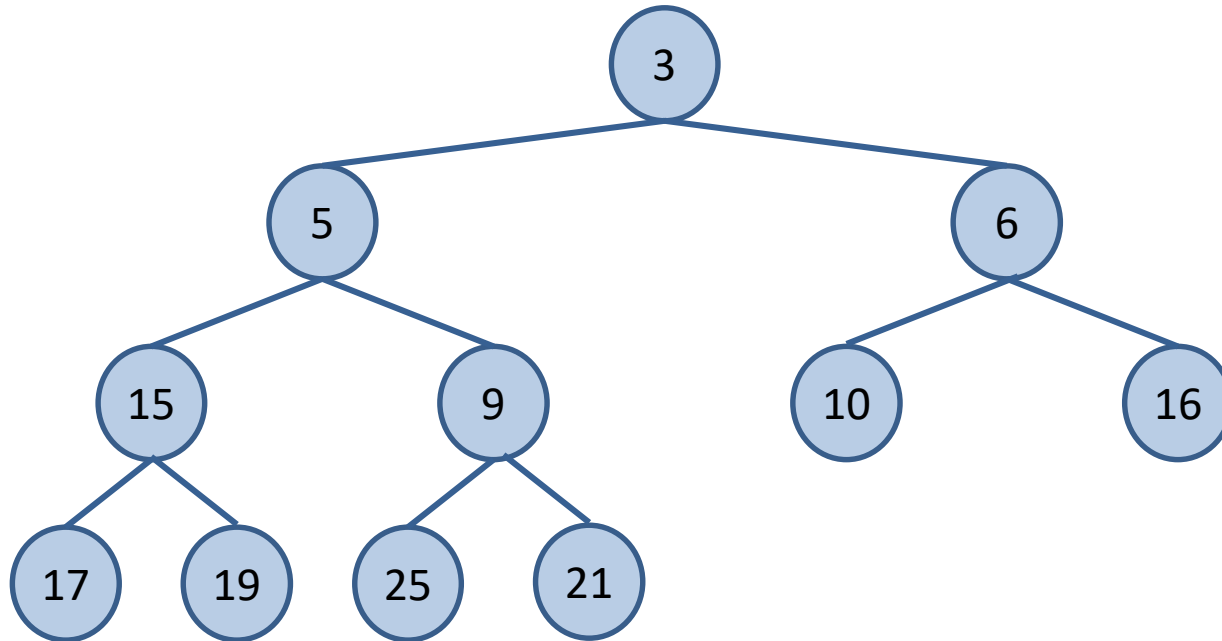
numNodos = 1



numNodos = 1

Suprimir en un APO

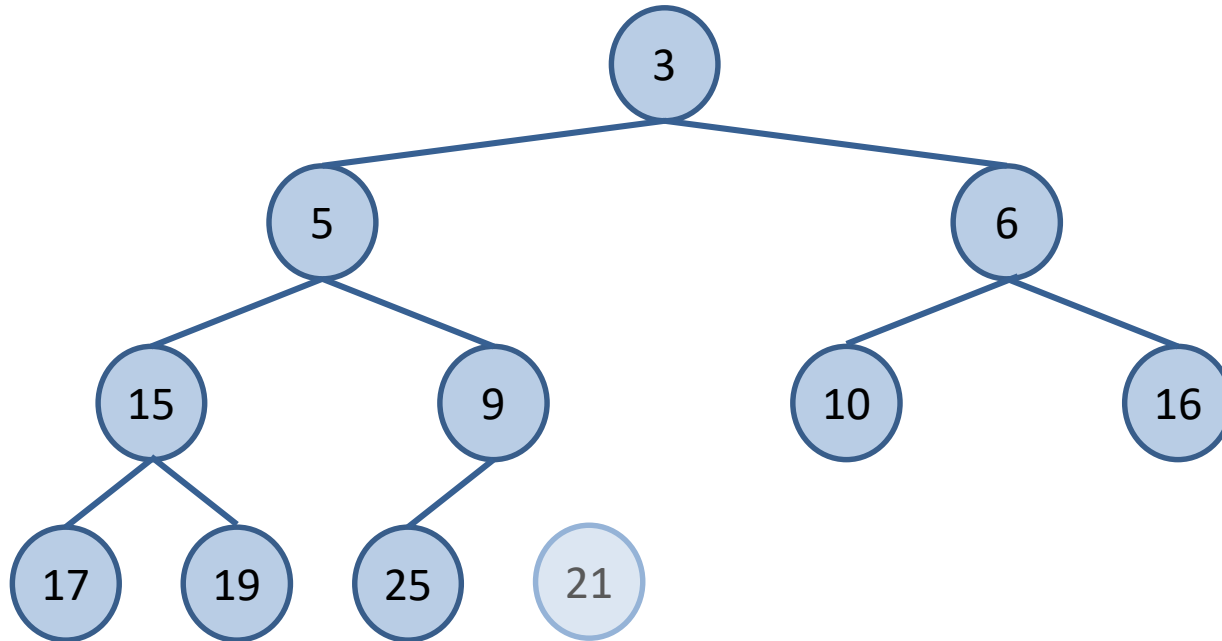
Caso 3: Más de dos nodos



numNodos = 11

Suprimir en un APO

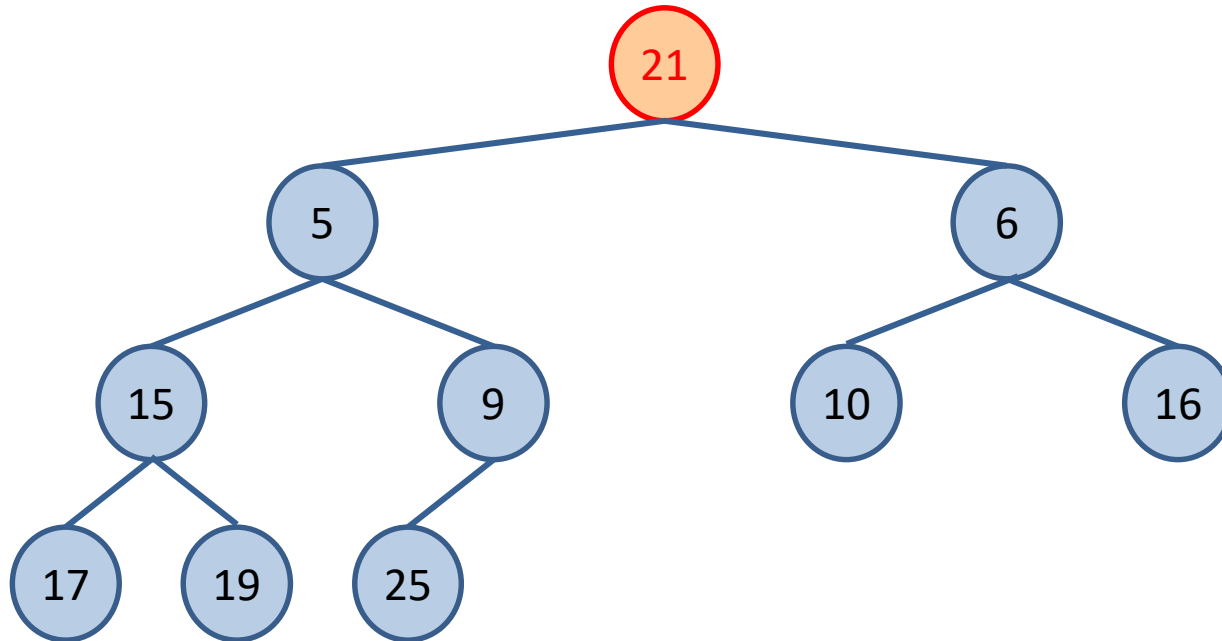
Caso 3: Más de dos nodos



numNodos = 10

Suprimir en un APO

Caso 3: Más de dos nodos

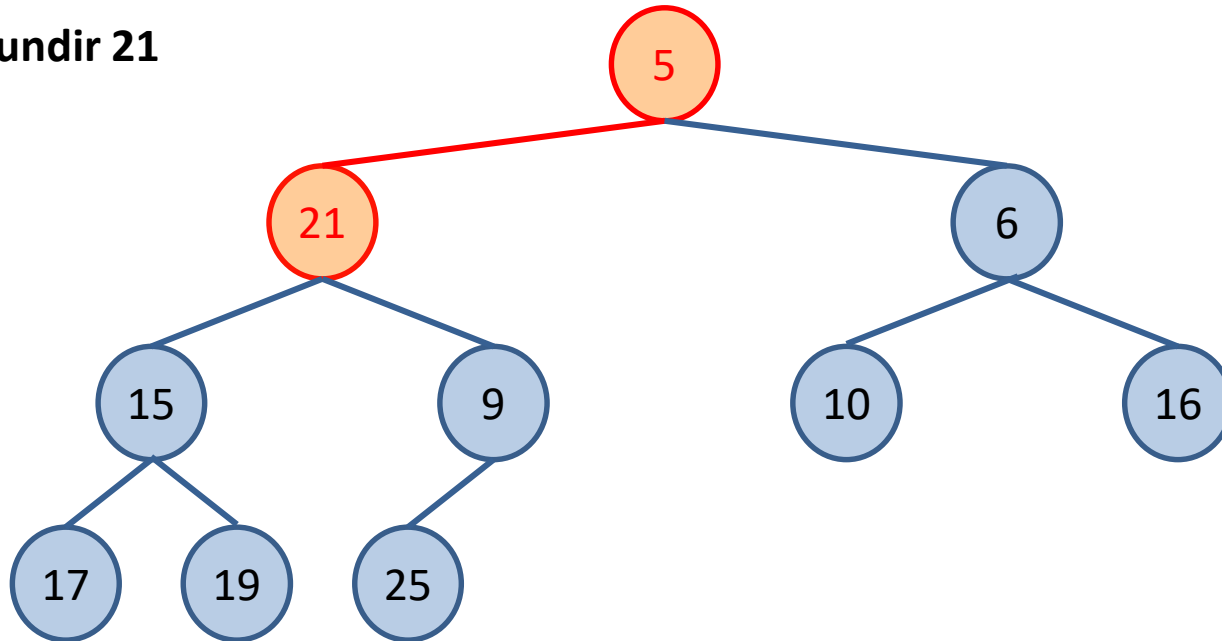


numNodos = 10

Suprimir en un APO

Caso 3: Más de dos nodos

Hundir 21

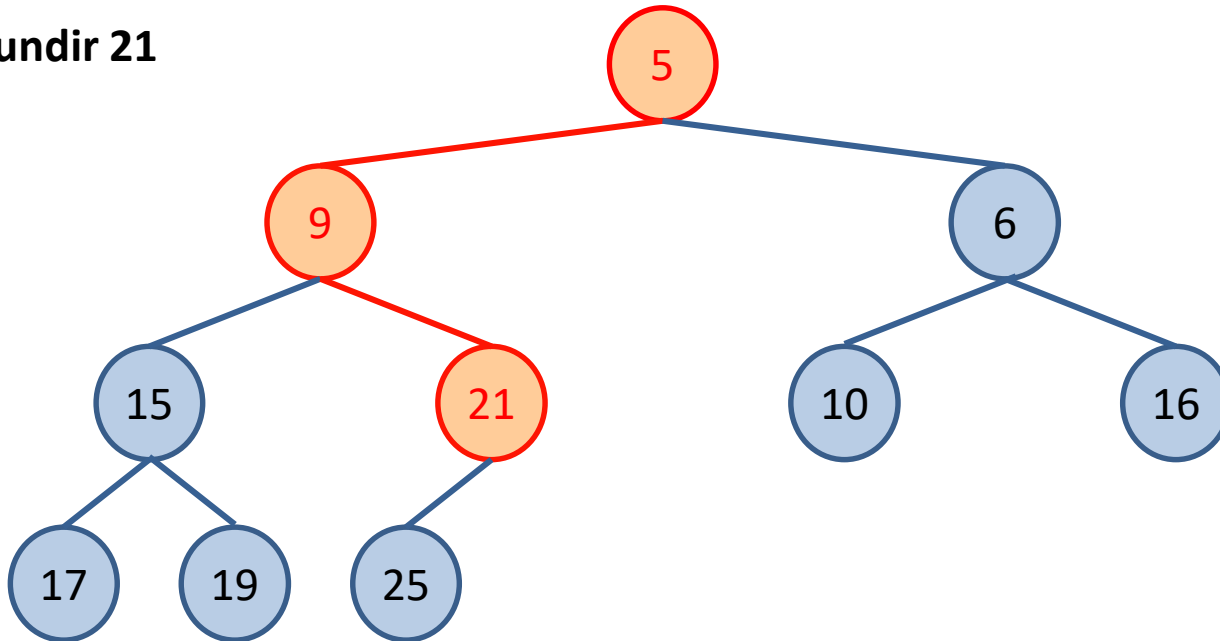


numNodos = 10

Suprimir en un APO

Caso 3: Más de dos nodos

Hundir 21



numNodos = 10

```

template <typename T>
inline void Apo<T>::suprimir()
{
    assert(numNodos > 0);    // Apo no vacío.
    if (--numNodos > 0)      // Apo no queda vacío.
    {
        nodos[0] = nodos[numNodos];
        if (numNodos > 1)    // 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) < numNodos && !fin)    // Hundir e.
    {
        nodo hMin;    // Hijo menor del nodo i.
        if (hDer(i) < numNodos && 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 <= nodos[hMin]
            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),
    numNodos(A.numNodos)
{
    // Copiar el vector.
    for (nodo n = 0; n < numNodos; 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];
        }
        numNodos = A.numNodos;
        // Copiar el vector
        for (nodo n = 0; n < numNodos; n++)
            nodos[n] = a.nodos[n];
    }
    return *this;
}

#endif // APO_H

```