

# Estructuras de Datos no Lineales

## 1.1. Árboles binarios

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

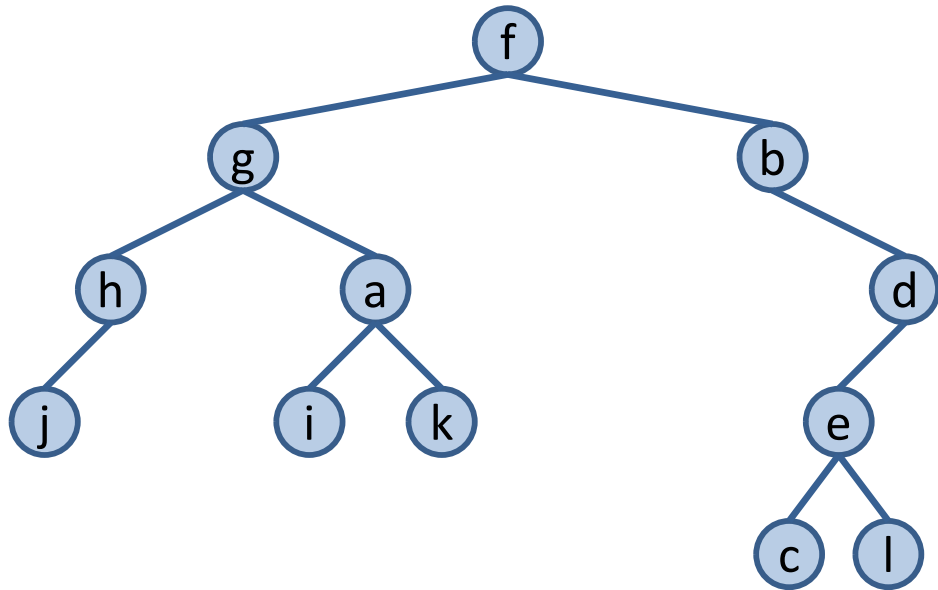


Versión 2.2

# TAD Árbol binario

## Definición:

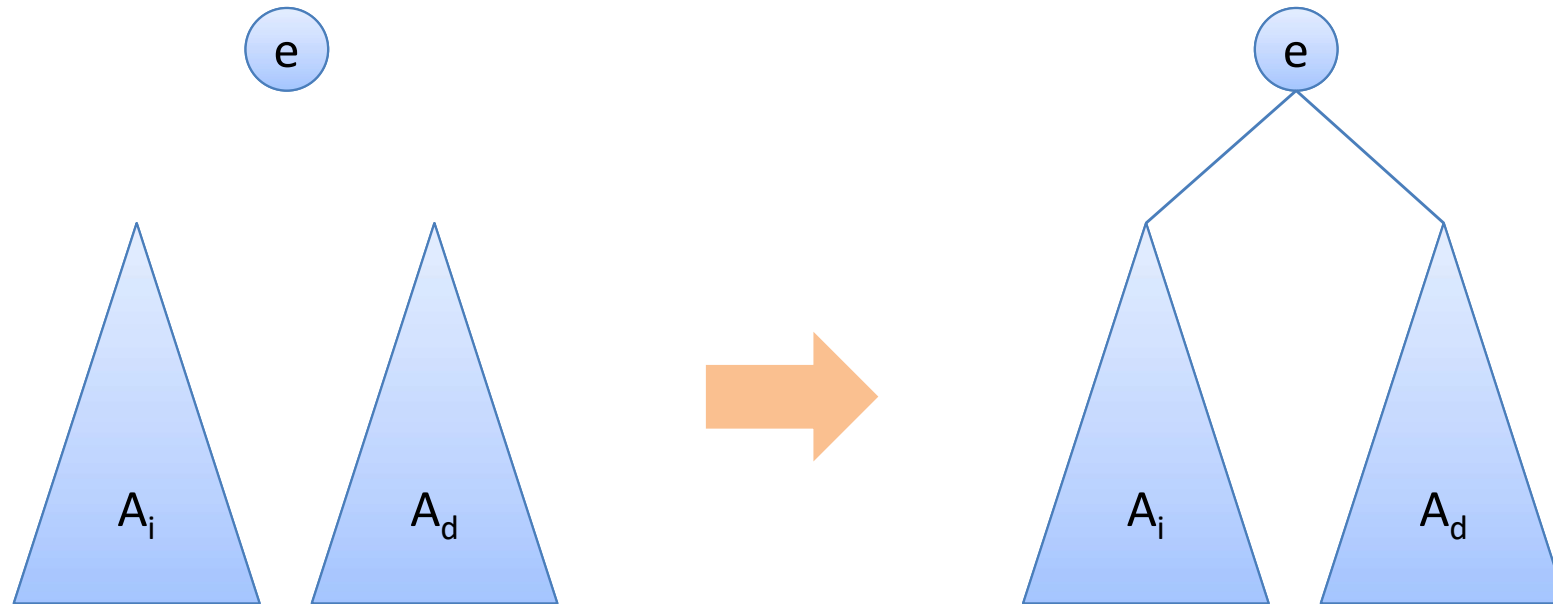
Un árbol binario se define como un árbol cuyos nodos son, a lo sumo, de grado 2, es decir, tienen 0, 1 ó 2 hijos. Éstos se llaman *hijo izquierdo* e *hijo derecho*.



## Operaciones:

- Construcción
- Inserción
- Eliminación
- Recuperación
- Modificación
- Acceso
- Destrucción

# Construcción de un árbol binario (I)

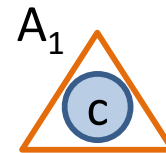


**Construcción a partir de los componentes de un árbol binario (un nodo y dos subárboles):**

```
Abin(); // Árbol vacío.  
void insertarRaiz(const T& e);  
void insertarIzqdo (Abin& Ai);  
void insertarDrcho(Abin& Ad);
```

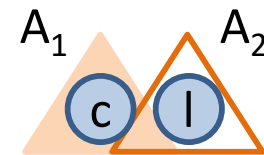
# Construcción de un árbol binario (I)

```
Abin A1;  
A1.insertarRaiz('c');
```



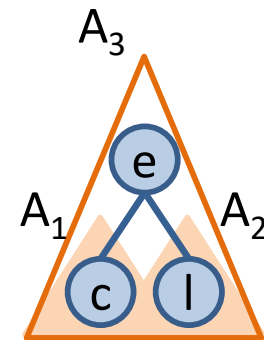
# Construcción de un árbol binario (I)

Abin  $A_2$ ;  
 $A_2$ .insertarRaiz('l');

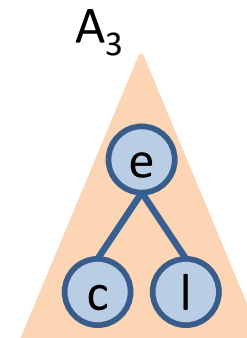
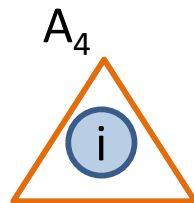


# Construcción de un árbol binario (I)

```
Abin A3; A3.insertarRaiz('e');  
A3.insertarIzqdo(A1);  
A3.insertarDrcho(A2);
```

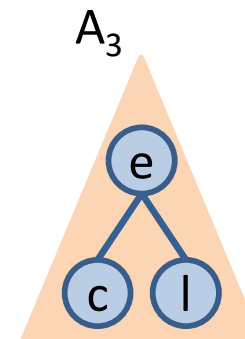
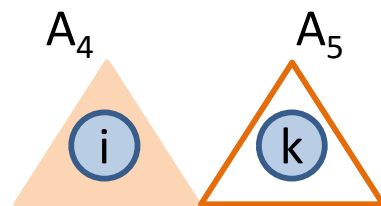


# Construcción de un árbol binario (I)



Abin  $A_4$ ;  
 $A_4$ .insertarRaiz('i');

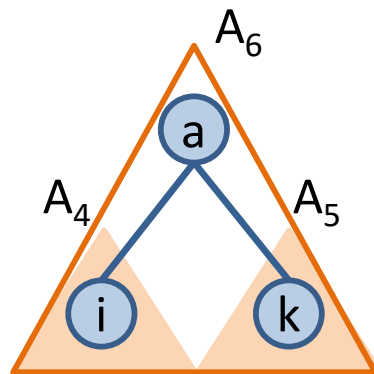
# Construcción de un árbol binario (I)



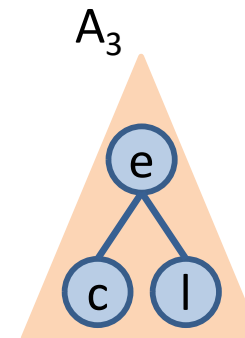
Abin  $A_5$ ;  
 $A_5.\text{insertarRaiz}('k');$



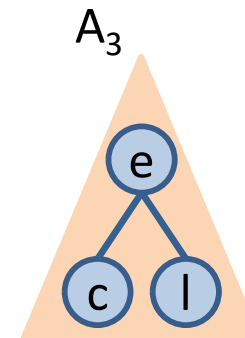
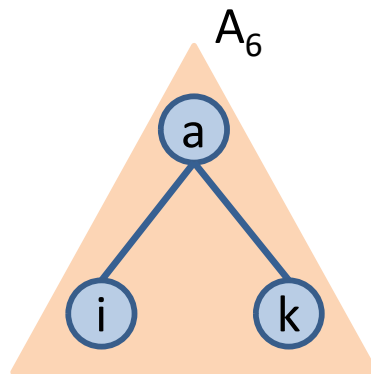
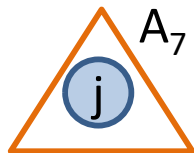
# Construcción de un árbol binario (I)



```
Abin A6; A6.insertarRaiz('a');  
A6.insertarIzqdo(A4);  
A6.insertarDrcho(A5);
```

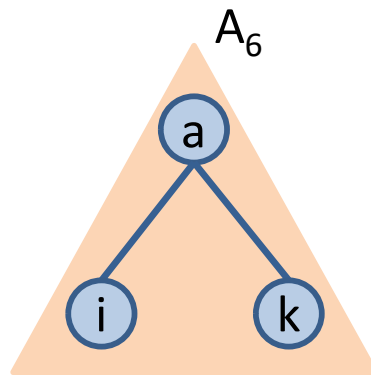
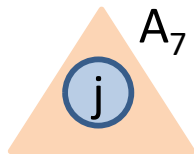


# Construcción de un árbol binario (I)

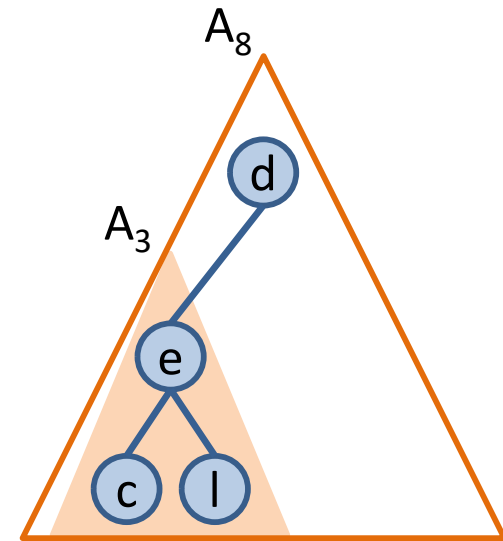


Abin  $A_7$ ;  
 $A_7$ .insertarRaiz('j');

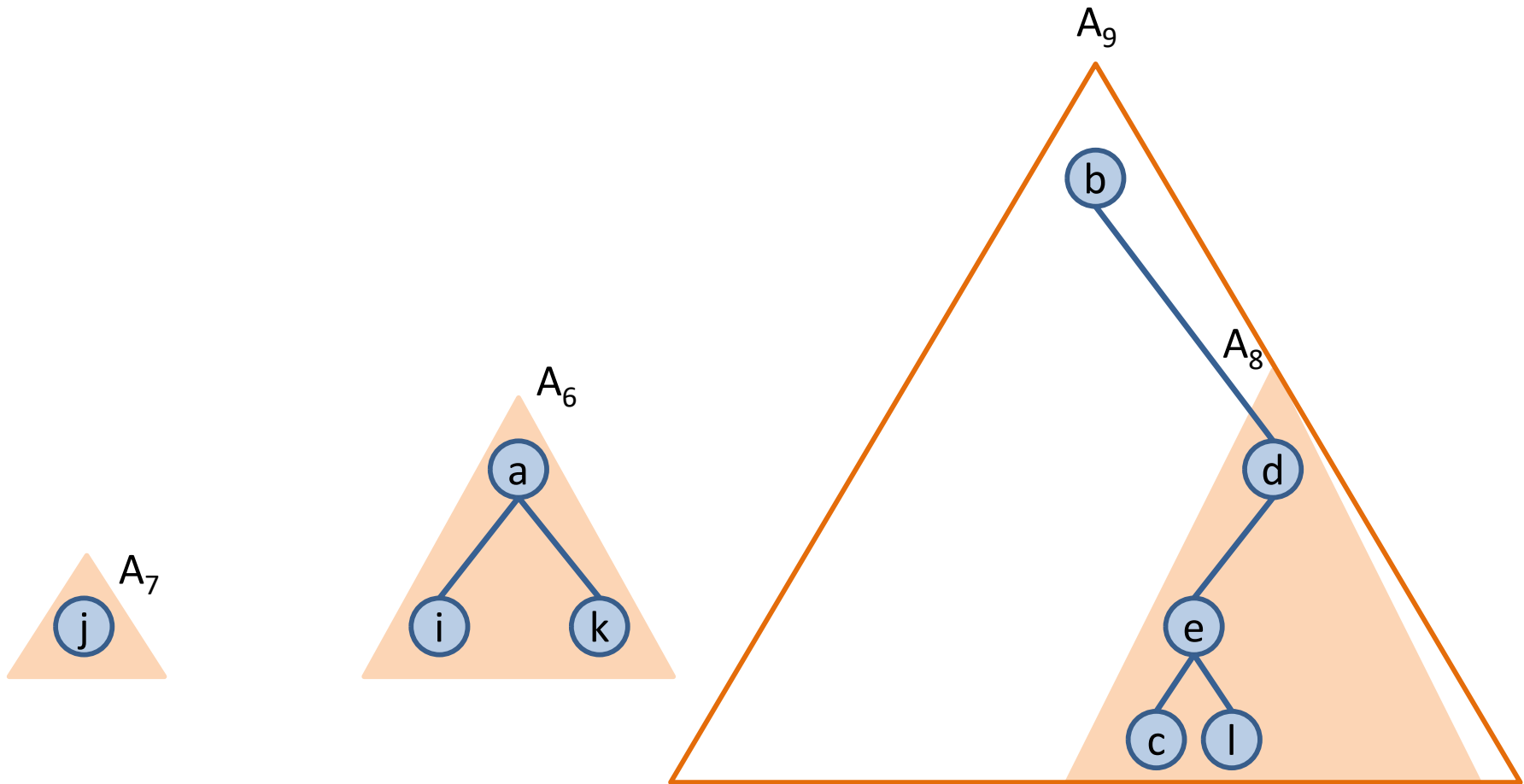
# Construcción de un árbol binario (I)



Abin A<sub>8</sub>;  
A<sub>8</sub>.insertarRaiz('d');  
A<sub>8</sub>.insertarIzqdo(A<sub>3</sub>);

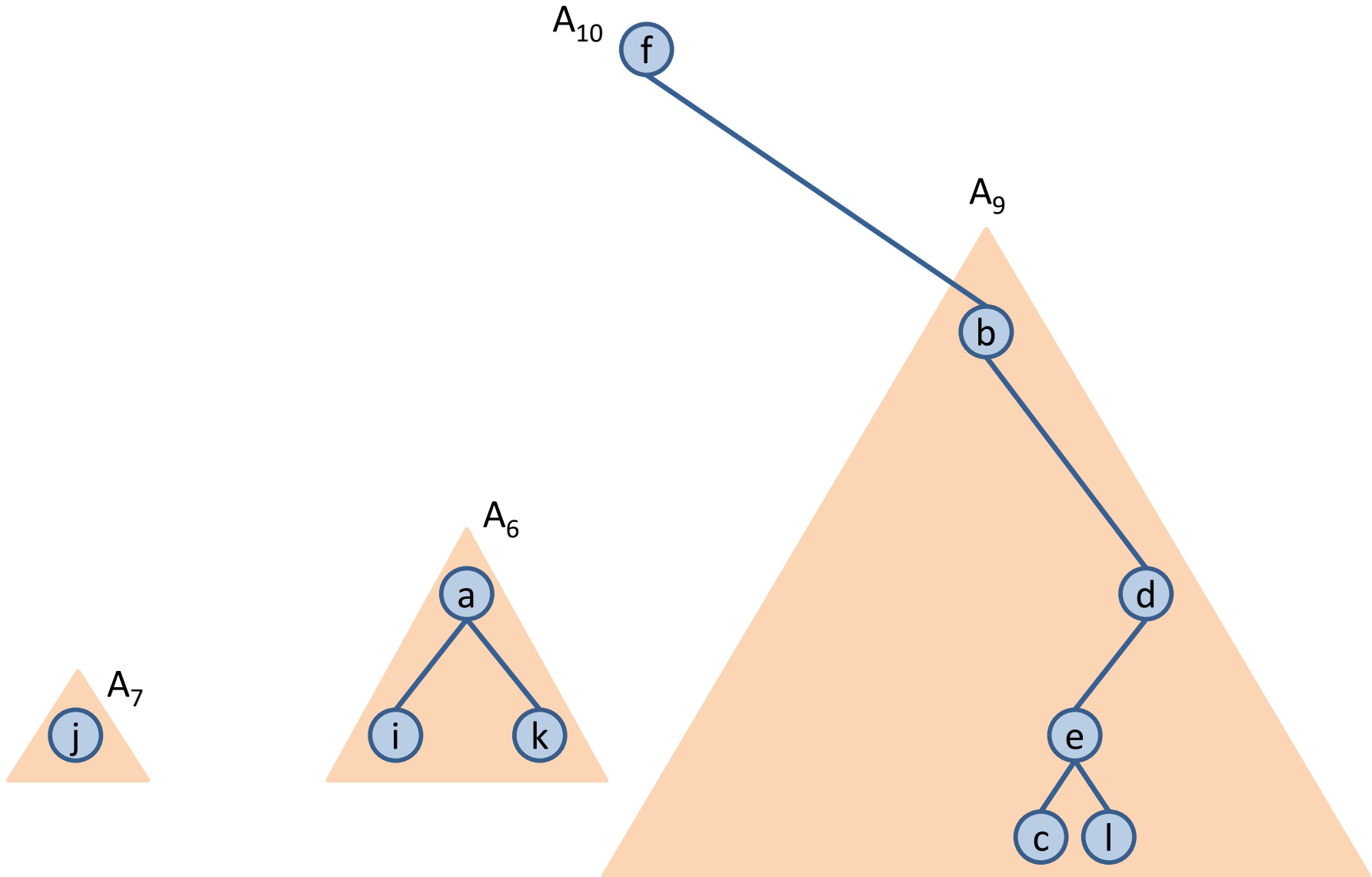


# Construcción de un árbol binario (I)



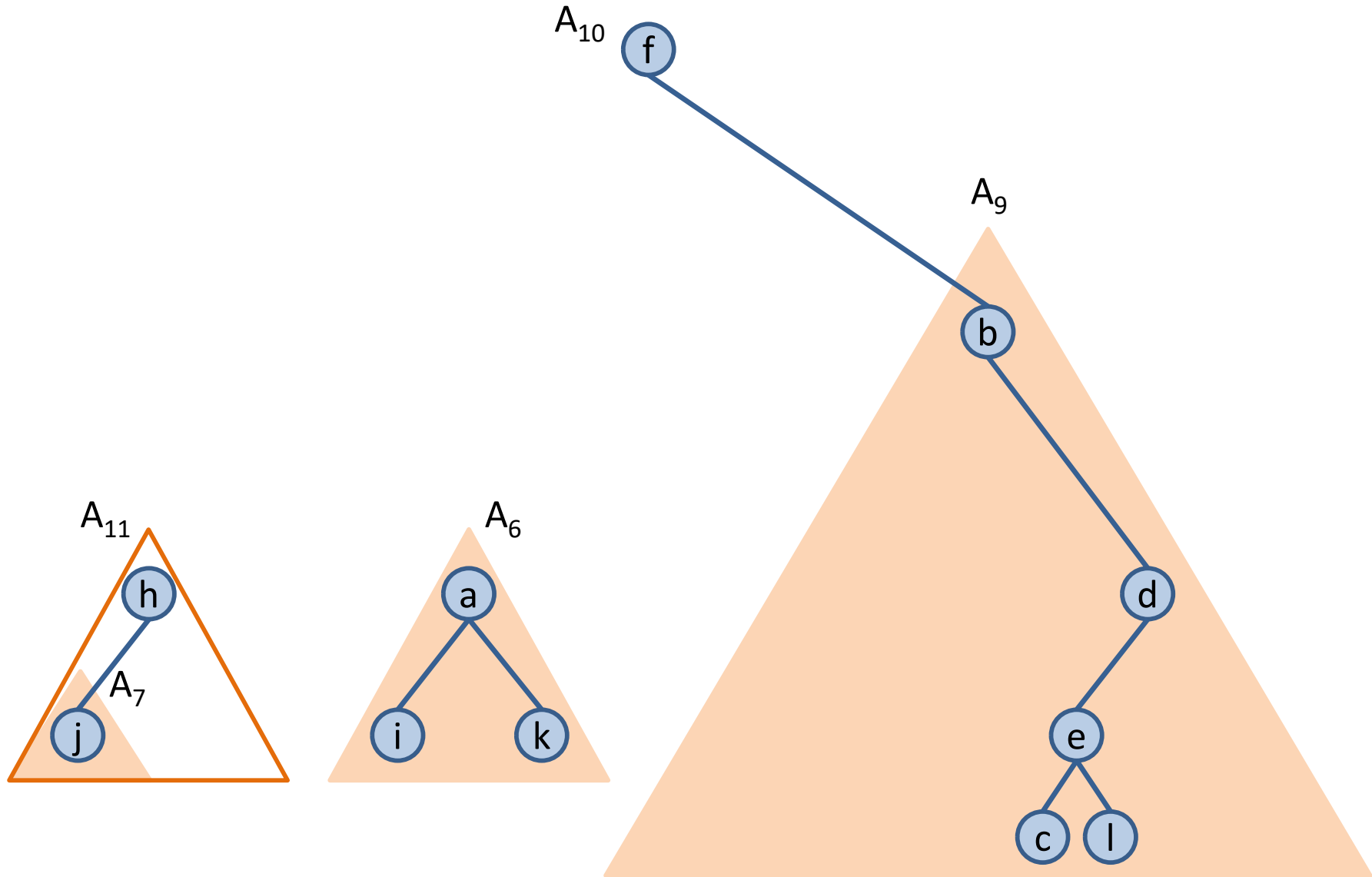
```
Abin A9; A9.insertarRaiz('b');  
A9.insertarDrcho(A8);
```

# Construcción de un árbol binario (I)



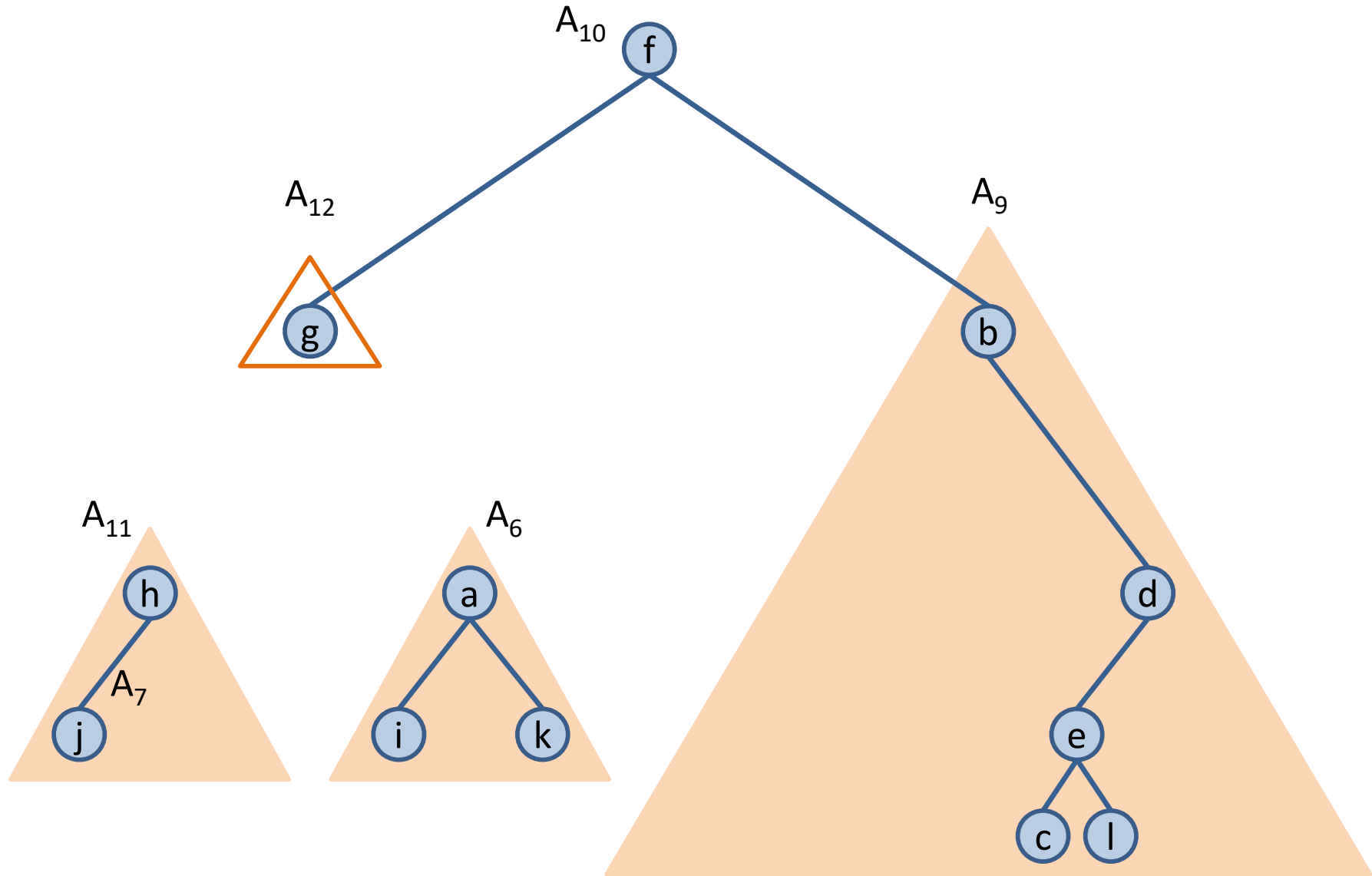
```
Abin A10; A10.insertarRaiz('f');  
A10.insertarDrcho(A9);
```

# Construcción de un árbol binario (I)



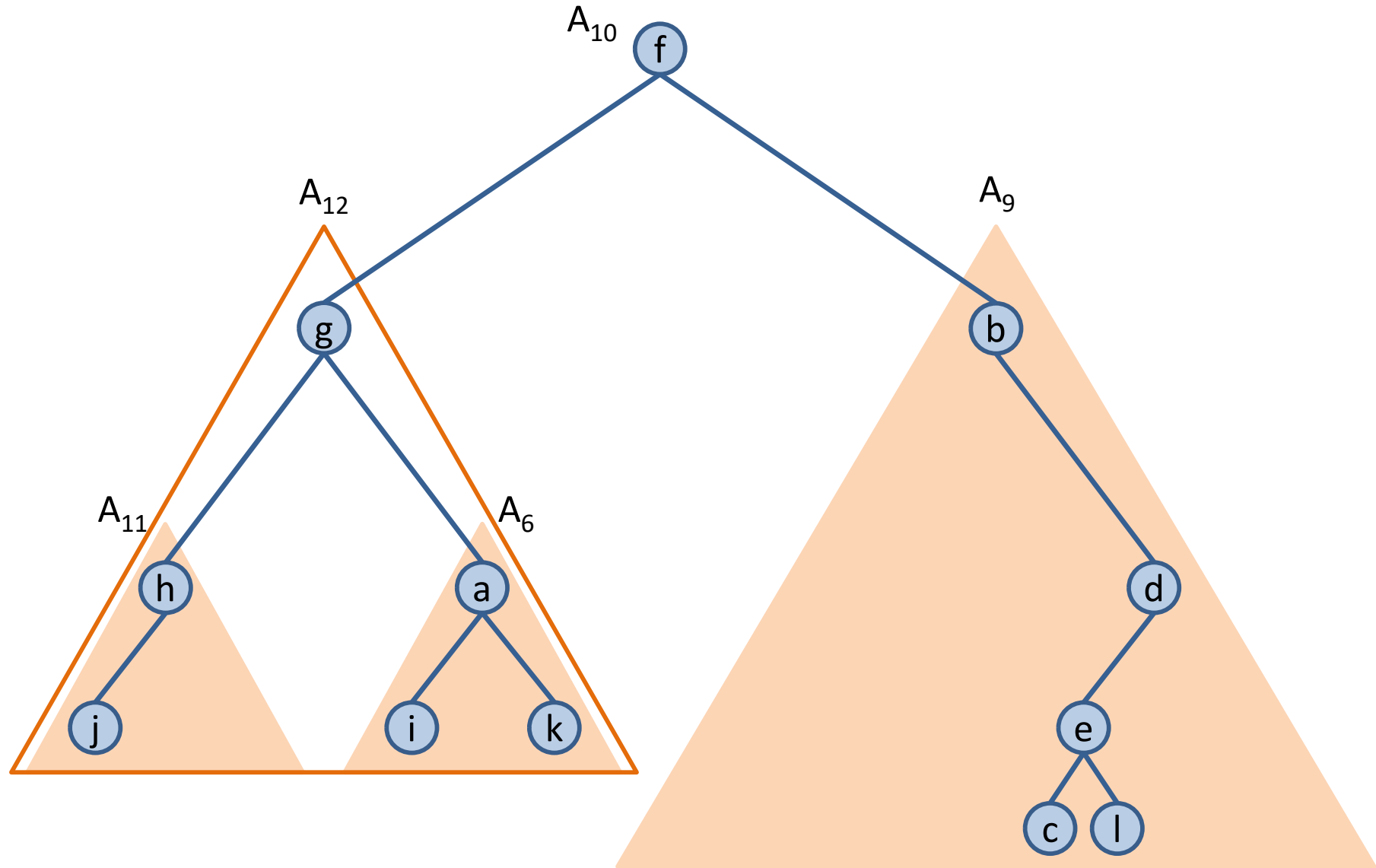
```
Abin A11; A11.insertarRaiz('h');  
A11.insertarIzqdo(A7);
```

# Construcción de un árbol binario (I)



Abin  $A_{12}$ ;  $A_{12}$ .insertarRaiz('g');  
 $A_{10}$ .insertarIzqdo( $A_{12}$ );

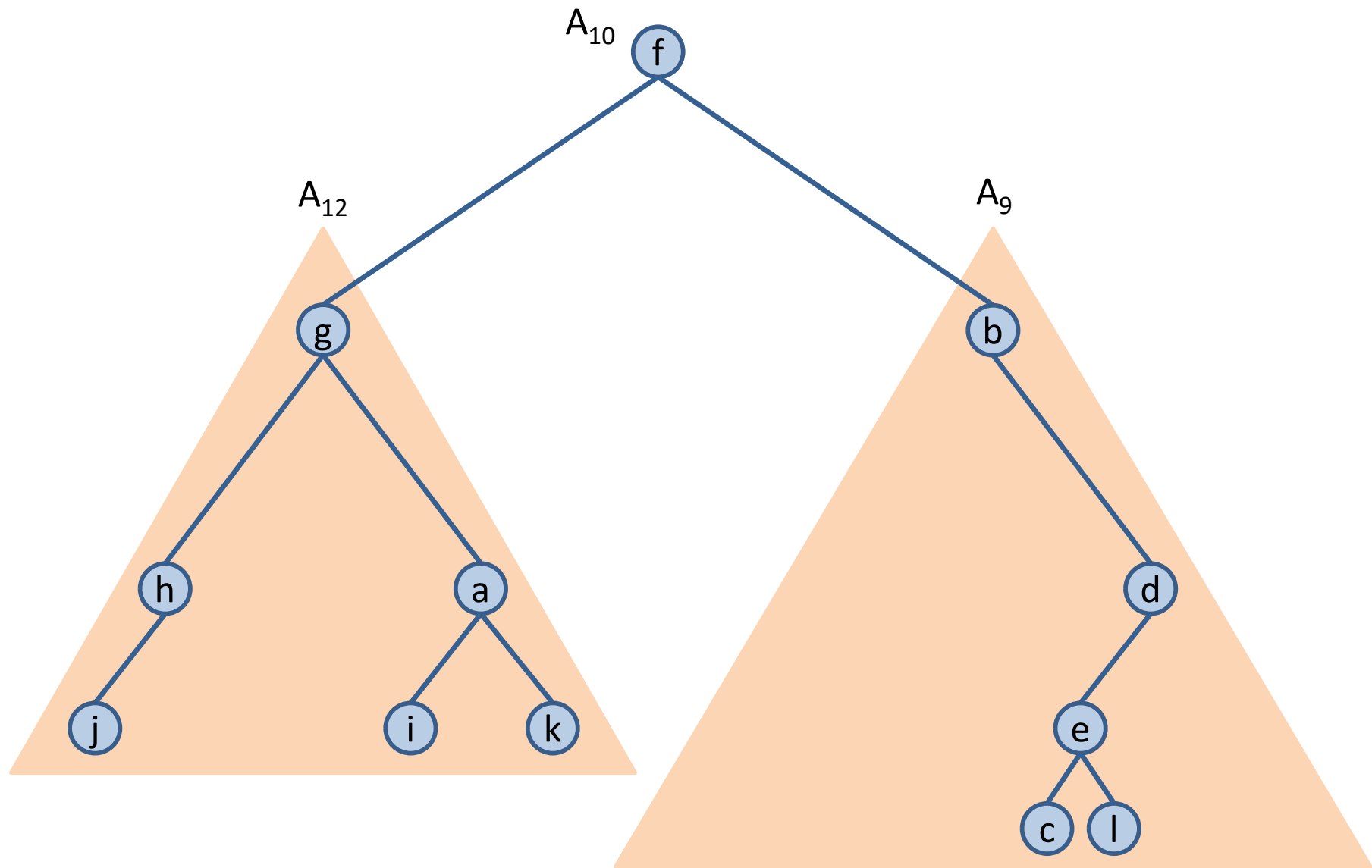
# Construcción de un árbol binario (I)



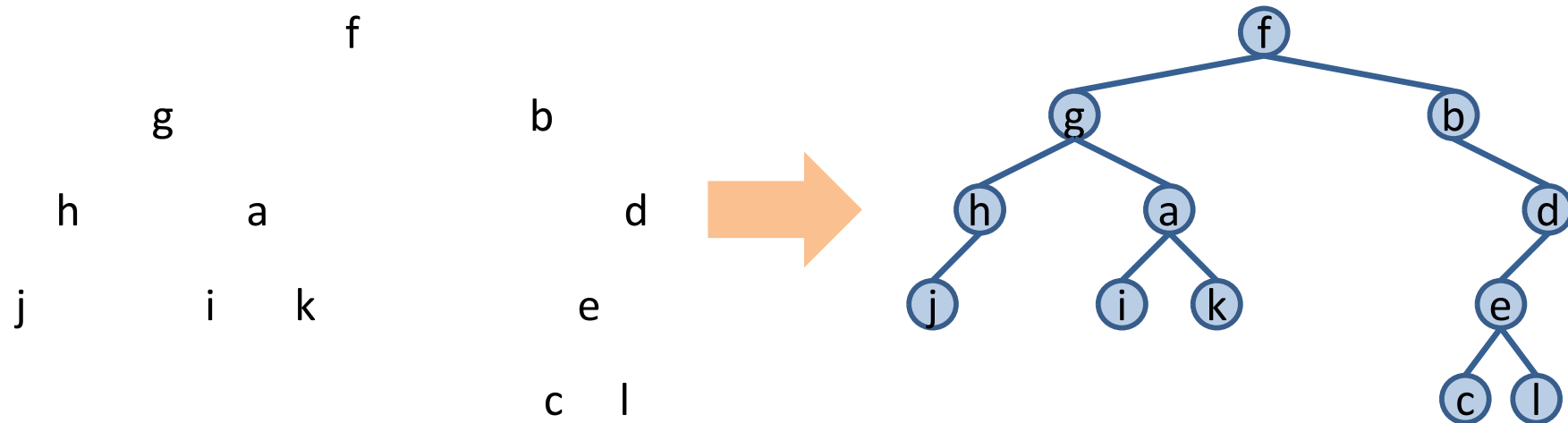
$A_{12}.insertarIzqdo(A_{11});$   
 $A_{12}.insertarDrcho(A_6);$



# Construcción de un árbol binario (I)



## Construcción de un árbol binario (II)

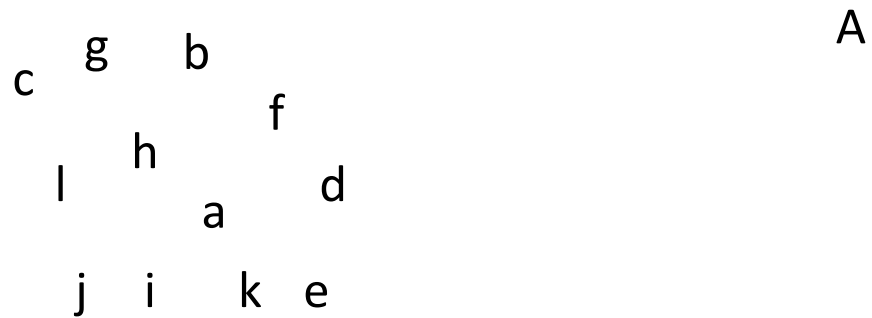


**Construcción añadiendo los nodos uno a uno desde la raíz hacia las hojas:**

```
Abin(); // Árbol vacío.  
void insertarRaiz(const T& e);  
void insertarHijoIzqdo(nodo n, const T& e);  
void insertarHijoDrcho(nodo n, const T& e);
```

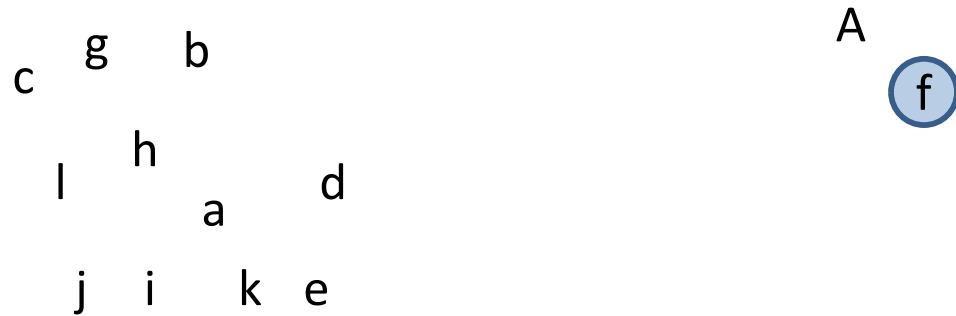
## Construcción de un árbol binario (II)

Creación del árbol binario A como un contenedor vacío.



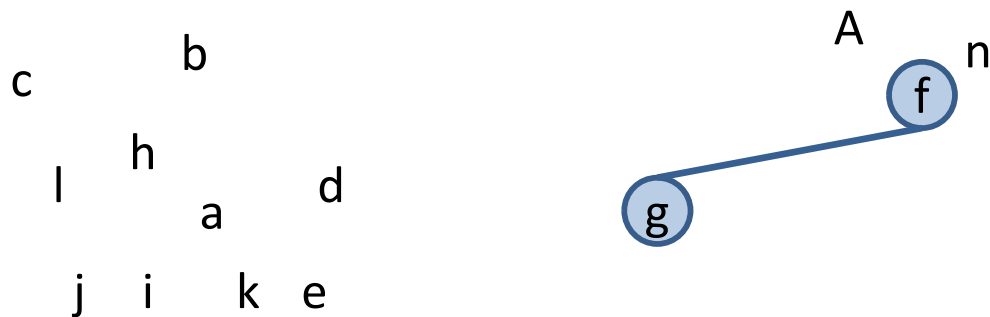
Abin A;

## Construcción de un árbol binario (II)



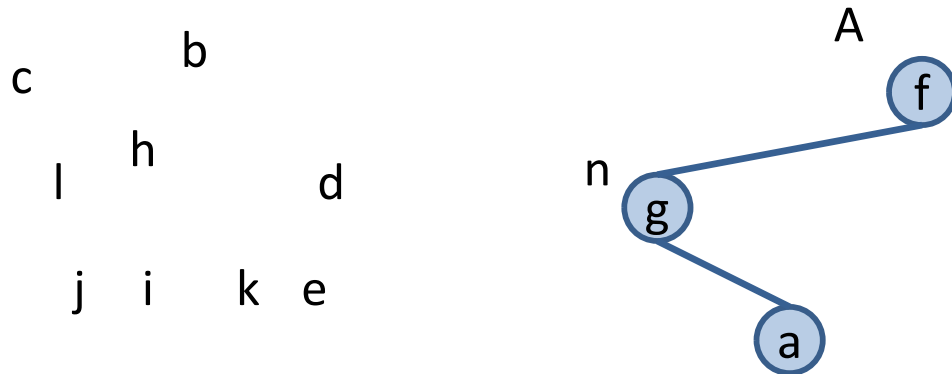
A.insertarRaiz('f');

## Construcción de un árbol binario (II)



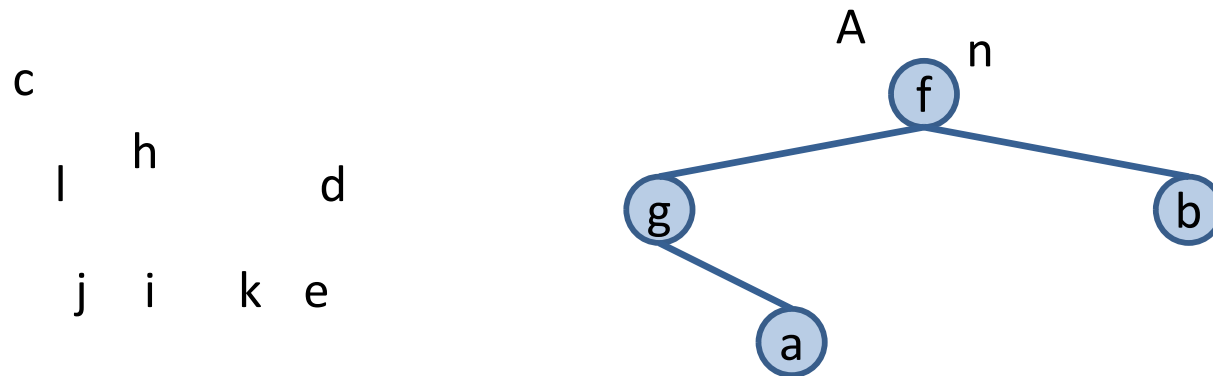
A.insertarHijoIzqdo(n, 'g');

## Construcción de un árbol binario (II)



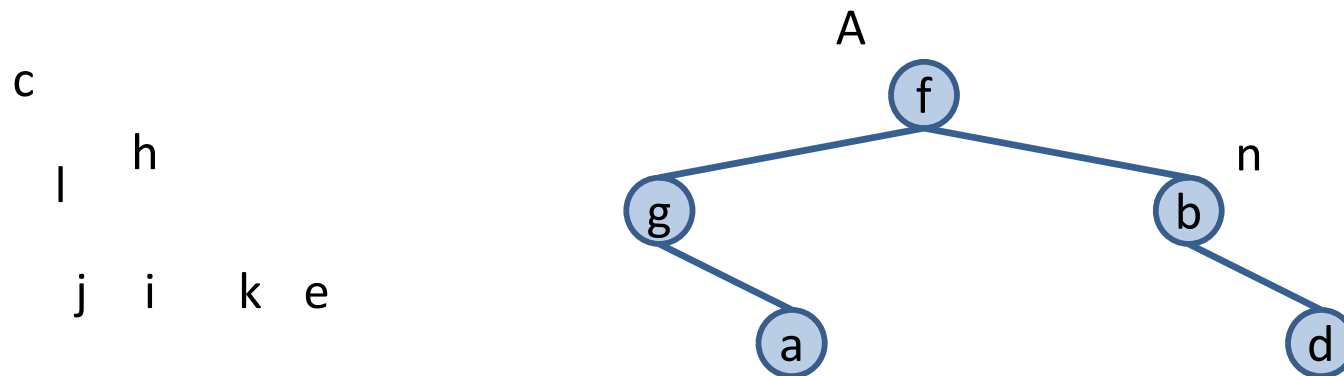
A.insertarHijoDrcho(n, 'a');

## Construcción de un árbol binario (II)



A.insertarHijoDrcho(n, 'b');

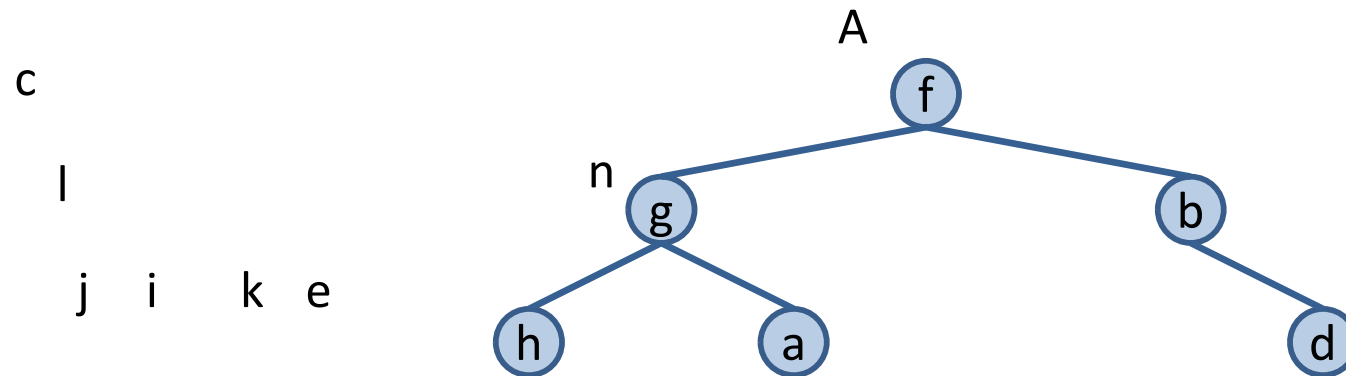
## Construcción de un árbol binario (II)



A.insertarHijoDrcho(n, 'd');

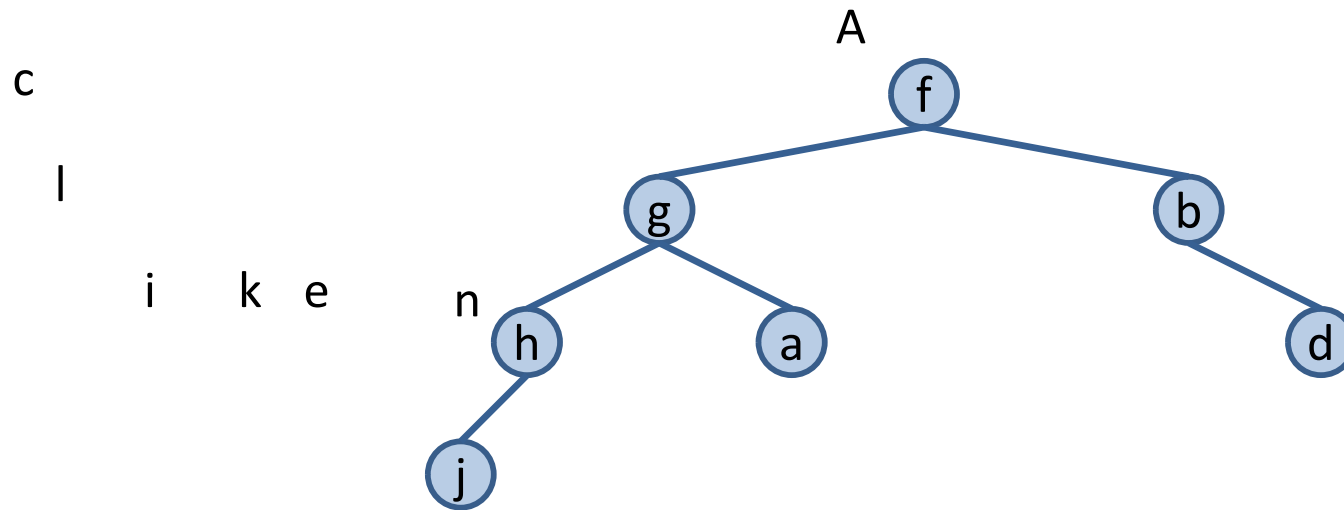


## Construcción de un árbol binario (II)



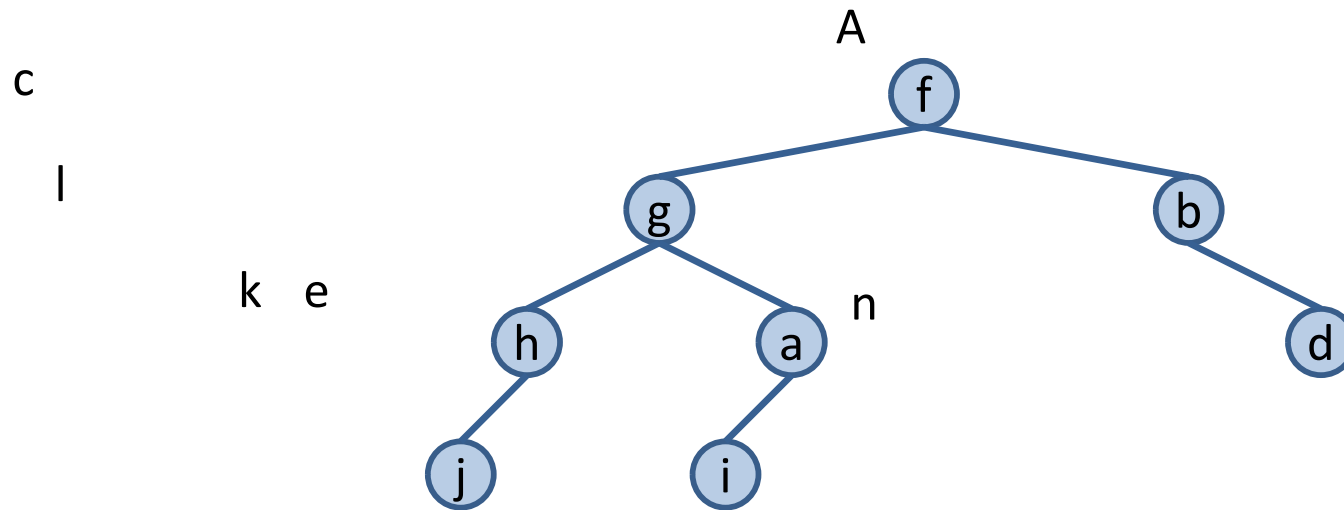
A.insertarHijoIzqdo(n, 'h');

## Construcción de un árbol binario (II)



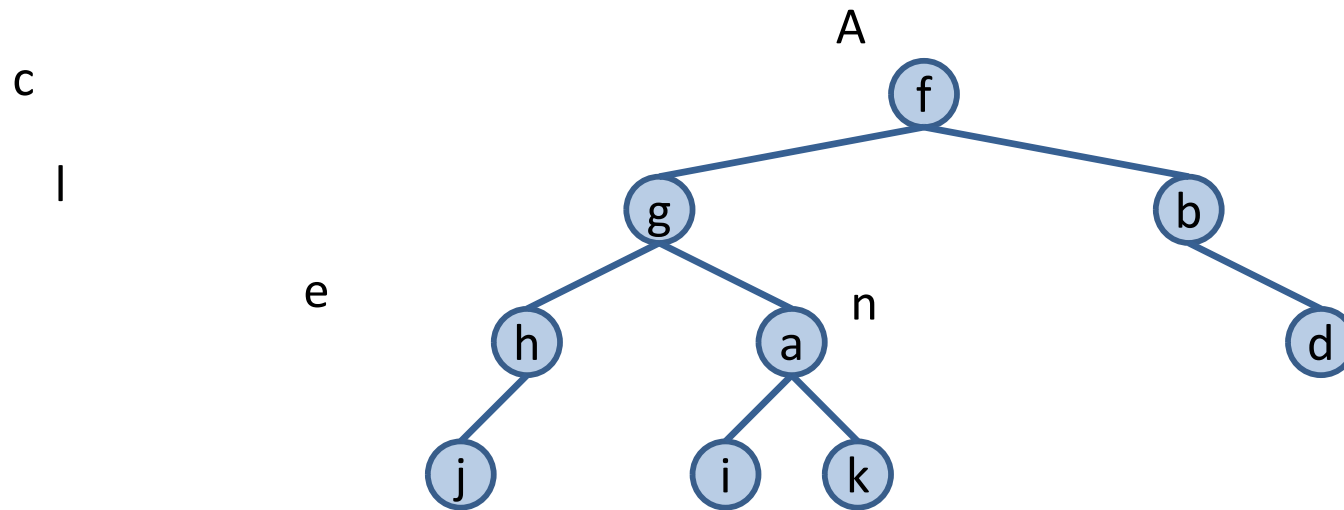
A.insertarHijolzqdo(n, 'j');

## Construcción de un árbol binario (II)



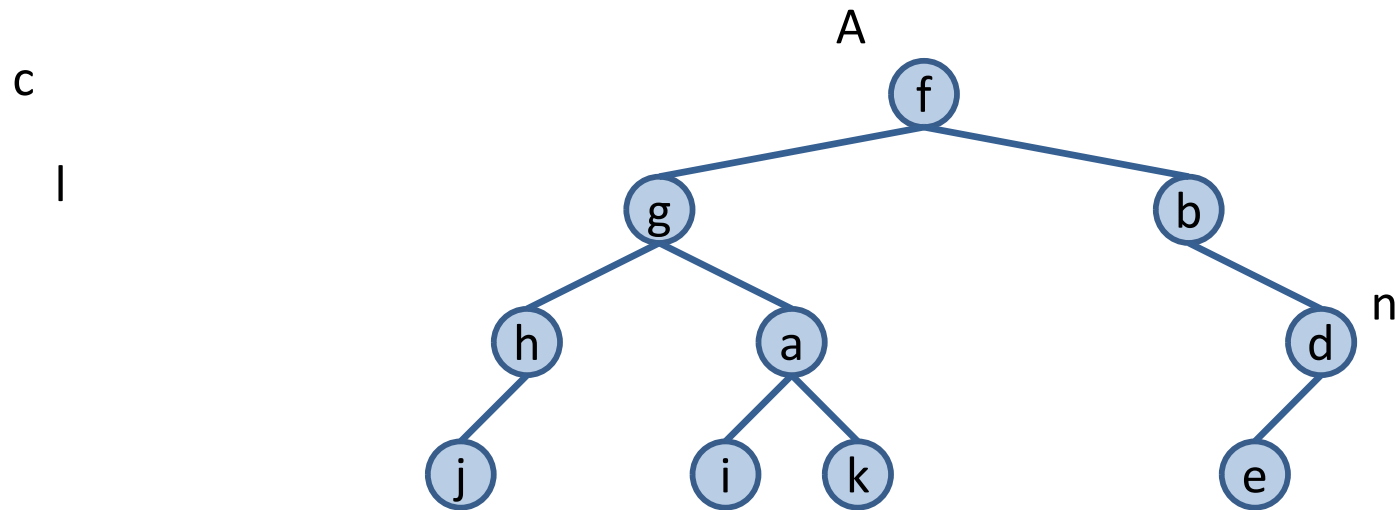
A.insertarHijolzqdo(n, 'i');

## Construcción de un árbol binario (II)



A.insertarHijoDrcho(n, 'k');

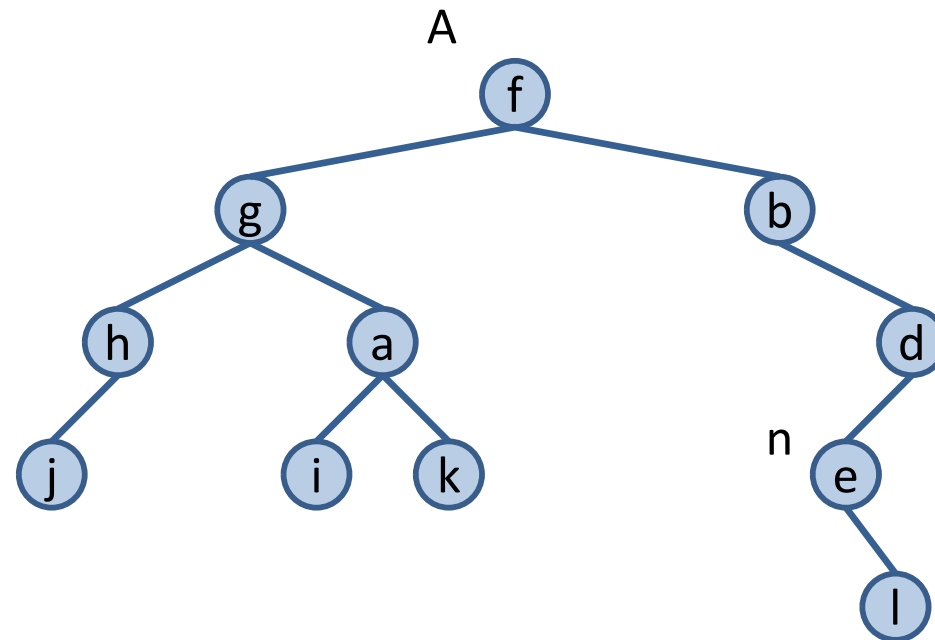
## Construcción de un árbol binario (II)



A.insertarHijoIzqdo(n, 'e');

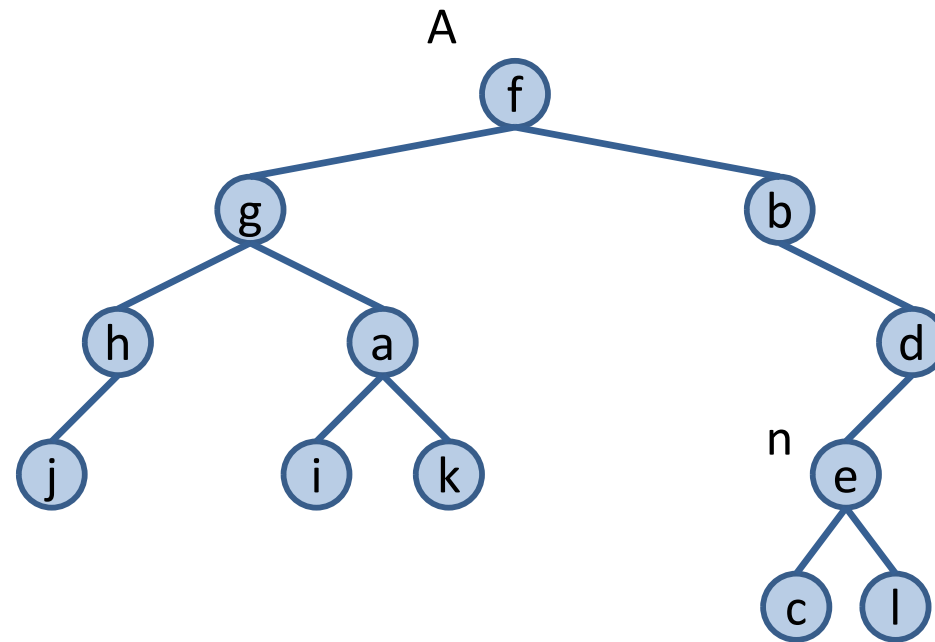
## Construcción de un árbol binario (II)

c



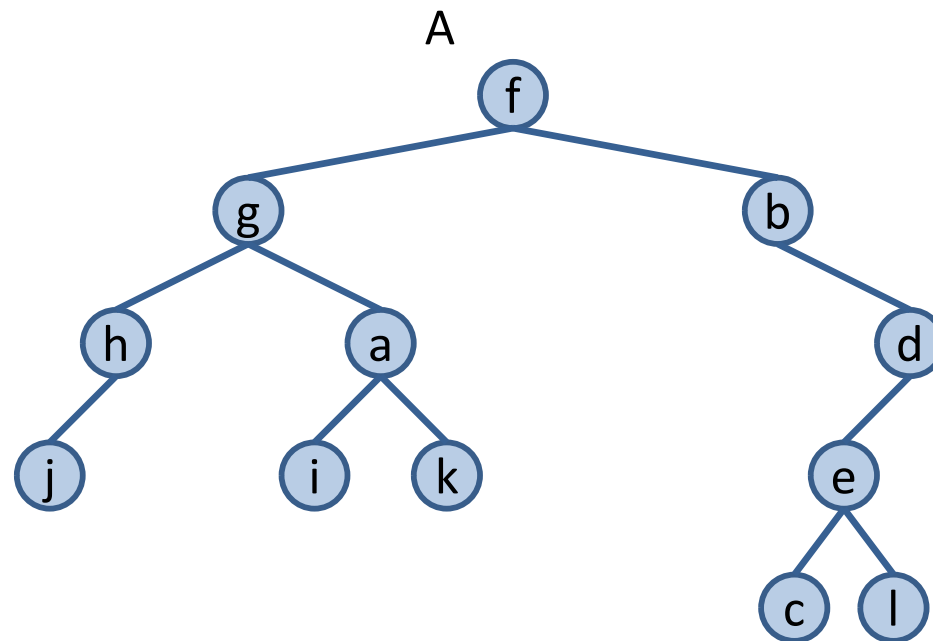
A.insertarHijoDrcho(n, 'l');

## Construcción de un árbol binario (II)



A.insertarHijoIzqdo(n, 'c');

## Construcción de un árbol binario (II)





# Especificación de operaciones:

*Abin ()*

Post: Crea y devuelve un árbol vacío.

*void insertarRaiz (const T& e)*

Pre: El árbol está vacío.

Post: Inserta el nodo raíz cuyo contenido será  $e$ .

*void insertarHijoIzqdo (nodo  $n$ , const T&  $e$ )*

Pre:  $n$  es un nodo del árbol que no tiene hijo izquierdo.

Post: Inserta el elemento  $e$  como hijo izquierdo del nodo  $n$ .

*void insertarHijoDrcho (nodo  $n$ , const T&  $e$ )*

Pre:  $n$  es un nodo del árbol que no tiene hijo derecho.

Post: Inserta el elemento  $e$  como hijo derecho del nodo  $n$ .

*void eliminarHijoIzqdo (nodo n)*

Pre:  $n$  es un nodo del árbol.

Existe  $hijoIzqdoB(n)$  y es una hoja.

Post: Destruye el hijo izquierdo del nodo  $n$ .

*void eliminarHijoDrcho (nodo n)*

Pre:  $n$  es un nodo del árbol.

Existe  $hijoDrchoB(n)$  y es una hoja.

Post: Destruye el hijo derecho del nodo  $n$ .

*void eliminarRaiz ()*

Pre: El árbol no está vacío y  $raiz()$  es una hoja.

Post: Destruye el nodo raíz. El árbol queda vacío

No podemos eliminar un nodo que no es una hoja, es decir, que no tiene nodo hijo derecho, izquierdo ó ambos.

Si no hacemos eso, nos estamos saltando la Especificación del TAD cosa que no puede ser.

*bool arbolVacio () const*

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

*const T& elemento(nodo n) const*

*T& elemento(nodo n)*

Pre: *n* es un nodo del árbol.

Post: Devuelve el elemento del nodo *n*.

*nodo raíz () const*

Post: Devuelve el nodo raíz del árbol. Si el árbol está vacío, devuelve *NODO\_NULO*.

*nodo padre (nodo n) const*

Pre: *n* es un nodo del árbol.

Post: Devuelve el padre del nodo *n*. Si *n* es el nodo raíz, devuelve *NODO\_NULO*.

*nodo hijoIzqdo (nodo n) const*

Pre: *n* es un nodo del árbol.

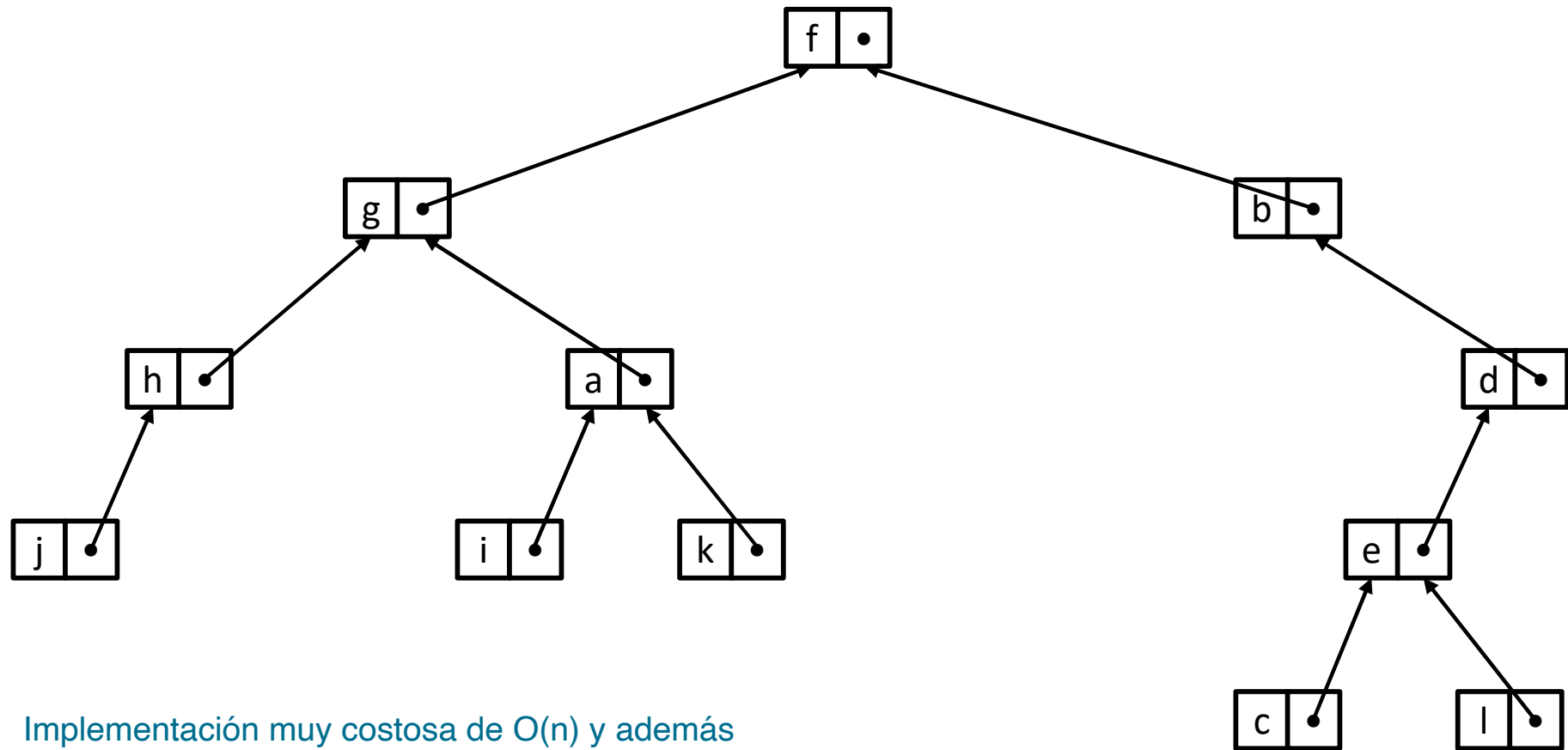
Post: Devuelve el nodo hijo izquierdo del nodo *n*. Si no existe, devuelve *NODO\_NULO*.

*nodo hijoDrcho (nodo n) const*

Pre: *n* es un nodo de *A*.

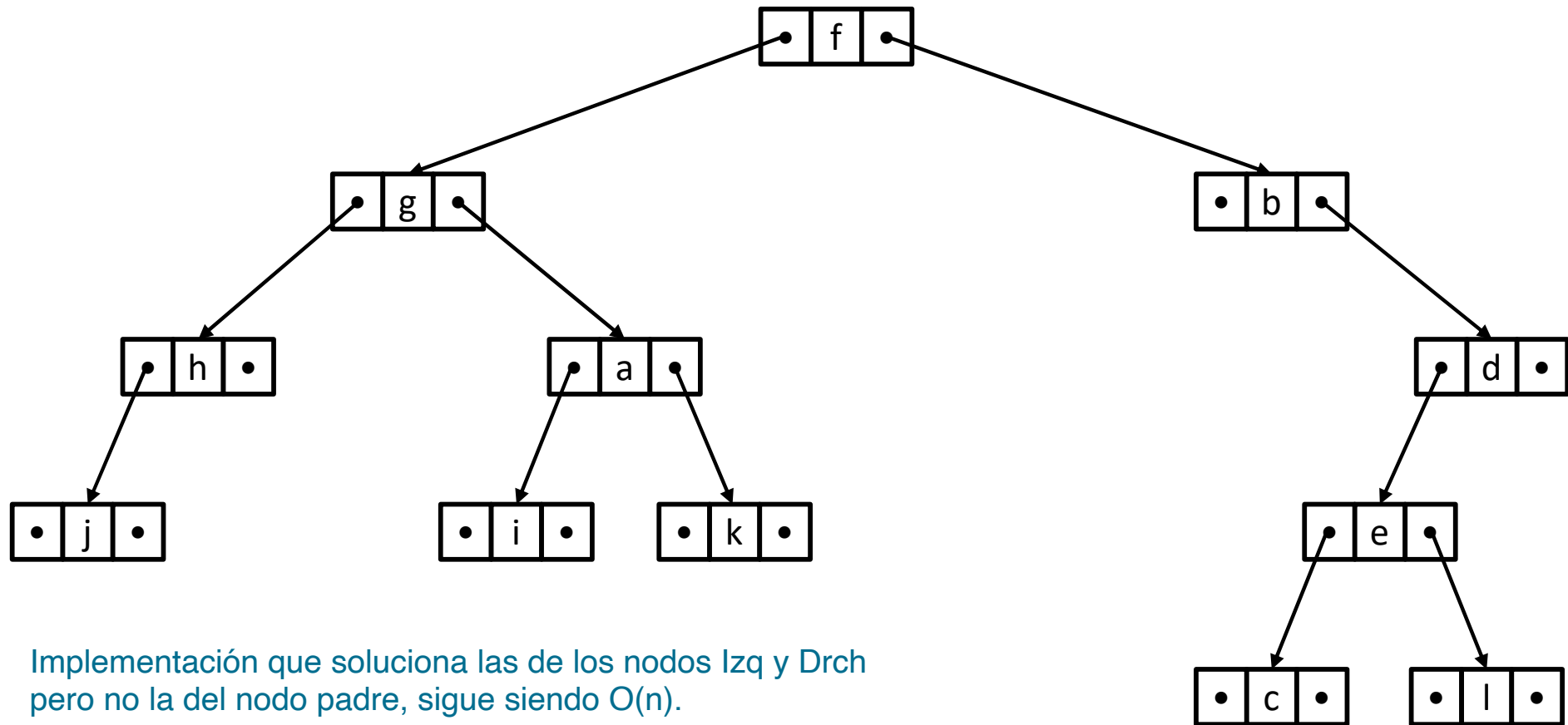
Post: Devuelve el nodo hijo derecho del nodo *n*. Si no existe, devuelve *NODO\_NULO*.

# Implementación de un árbol binario usando celdas enlazadas

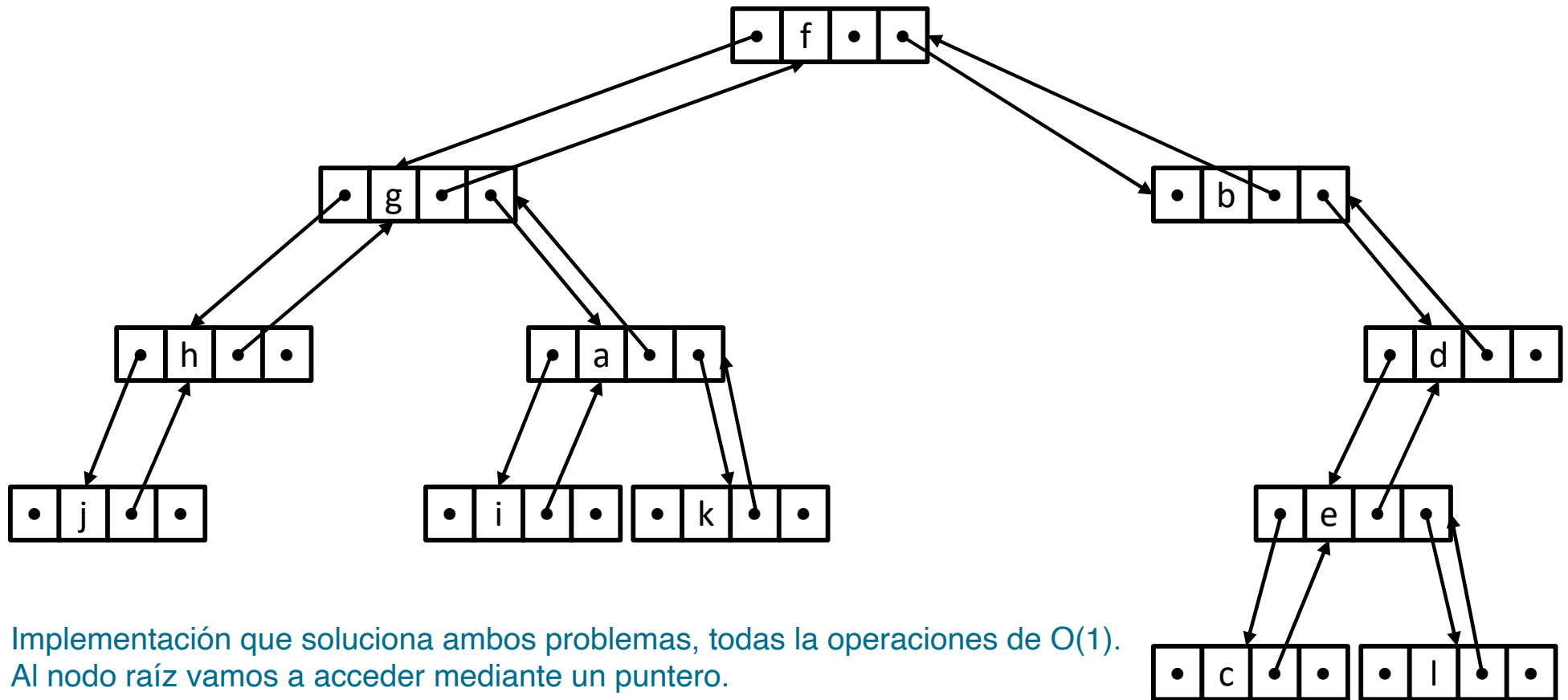


Implementación muy costosa de  $O(n)$  y además tenemos que acceder desde las hojas cosa que es muy compleja

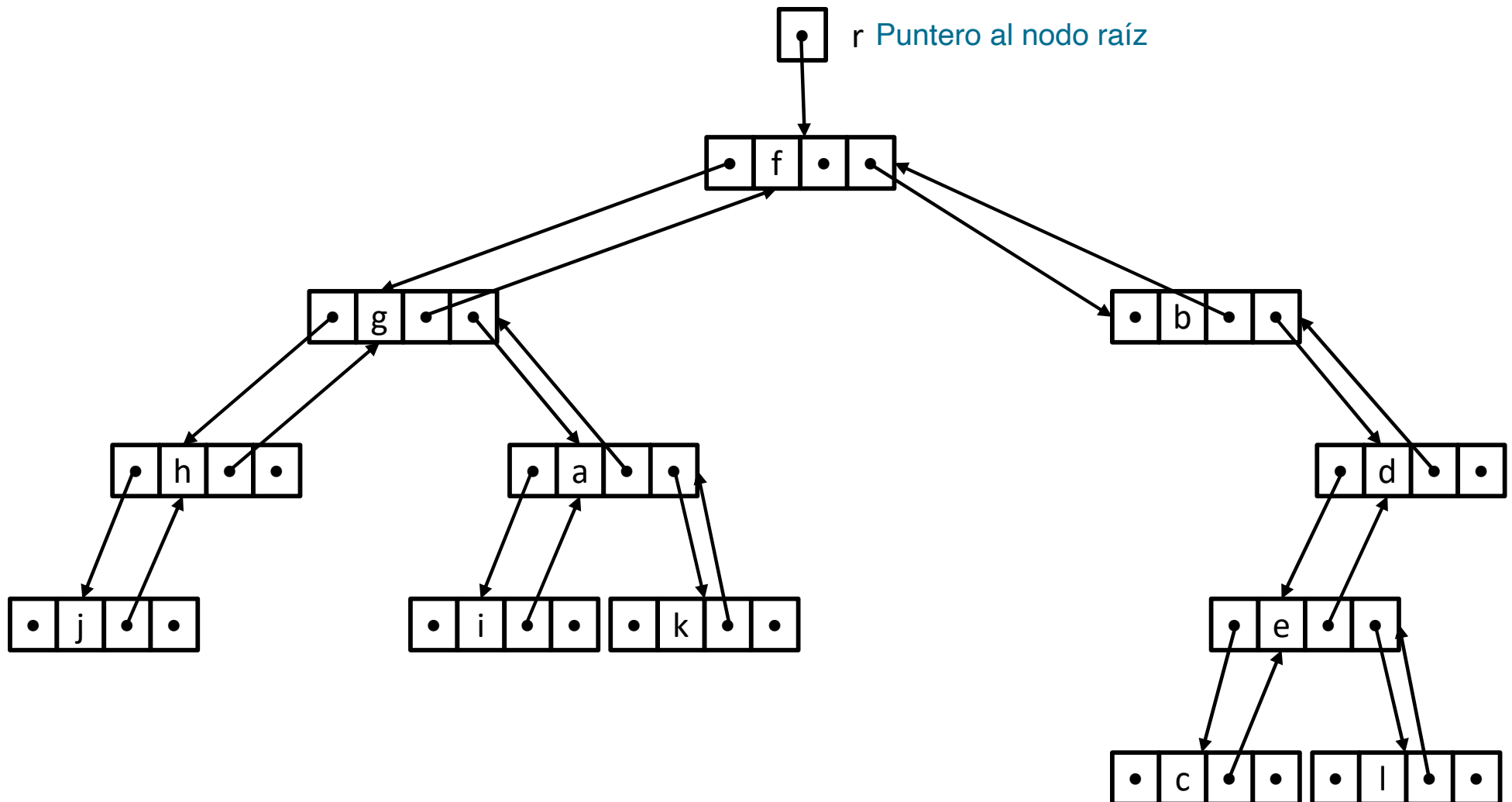
# Implementación de un árbol binario usando celdas enlazadas



# Implementación de un árbol binario usando celdas enlazadas



# Implementación de un árbol binario usando celdas enlazadas





```

#ifndef ABIN_H
#define ABIN_H
#include <cassert>

template <typename T> class Abin {
    struct celda;    // Declaración adelantada privada
public:
    typedef celda* nodo;
    static const nodo NODO_NULO; Indica que no hay un nodo

    Abin();           // Constructor
    void insertarRaiz(const T& e);
    void insertarHijoIzqdo(nodo n, const T& e);
    void insertarHijoDrcho(nodo n, const T& e);
    void eliminarHijoIzqdo(nodo n);
    void eliminarHijoDrcho(nodo n);
    void eliminarRaiz();
    bool arbolVacio() const;
    const T& elemento(nodo n) const; // Lec. en Abin const
    T& elemento(nodo n);             // Lec/Esc. en Abin no-const
    nodo raiz() const;

```

```

nodo padre(nodo n) const;
nodo hijoIzqdo(nodo n) const;
nodo hijoDrcho(nodo n) const;
Abin(const Abin<T>& a); // Ctor. de copia
Abin<T>& operator =(const Abin<T>& A); // Asig. de árboles
~Abin(); // Destructor

private:
    struct celda {
        T elto;
        nodo padre, hizq, hder;
        celda(const T& e, nodo p = NODO_NULO): elto(e),
            padre(p), hizq(NODO_NULO), hder(NODO_NULO) {}
    };
    nodo r; // nodo raíz del árbol

    void destruirNodos(nodo& n);
    nodo copiar(nodo n);
};

/* Definición del nodo nulo */
template <typename T>
const typename Abin<T>::nodo Abin<T>::NODO_NULO(nullptr);

```

Un árbol en sí es un puntero a una celda

```

/*-----*/
/* Métodos públicos */
/*-----*/
    Al crearse un árbol vacío se crea un árbol vacío = nodo raíz nulo

```

```

template <typename T>

```

```

inline Abin<T>::Abin() : r(NODO_NULO) {}

```

```

template <typename T>

```

```

inline void Abin<T>::insertarRaiz(const T& e)

```

```

{

```

```

    assert(r == NODO_NULO); // Árbol vacío

```

```

    r = new celda(e); // Al crearse un nodo raíz se inicializan ambos hijos (I,D) a nulo

```

```

}

```

Como el nodo raíz no tiene nodo padre,  
solo se le pasa el elto

```

template <typename T>

```

```

inline void Abin<T>::insertarHijoIzqdo(nodo n, const T& e)

```

Nodo padre del que queremos insertar

```

{

```

```

    assert(n != NODO_NULO); // Asumimos que n es un nodo del árbol y que hizq no exista

```

```

    assert(n->hizq == NODO_NULO); // No existe hijo izqdo.

```

```

    n->hizq = new celda(e, n);

```

```

}

```

```

template <typename T> Análogo al Izquierdo
inline void Abin<T>::insertarHijoDrcho(nodo n, const T& e)
{
    assert(n != NODO_NULO);
    assert(n->hder == NODO_NULO); // No existe hijo drcho.
    n->hder = new celda(e, n);
}

```

```

template <typename T>
inline void Abin<T>::eliminarHijoIzqdo(nodo n)
{
    assert(n != NODO_NULO);
    assert(n->hizq != NODO_NULO); // Existe hijo izqdo.
    assert(n->hizq->hizq == NODO_NULO && // Hijo izqdo.
           n->hizq->hder == NODO_NULO); // es hoja.
    delete n->hizq;
    n->hizq = NODO_NULO;
}

```

Comprobamos que existe el nodo a eliminar,  
lo eliminamos y lo volvemos NULO, para hacer que  
esté en un estado válido y esa dirección de memoria  
se libere correctamente

```

template <typename T>    Análogo al Izquierdo
inline void Abin<T>::eliminarHijoDrcho(nodo n)
{
    assert(n != NODO_NULO);
    assert(n->hder != NODO_NULO);    // Existe hijo drcho.
    assert(n->hder->hizq == NODO_NULO &&    // Hijo drcho.
           n->hder->hder == NODO_NULO);    // es hoja
    delete n->hder;
    n->hder = NODO_NULO;
}

```

```

template <typename T>
inline void Abin<T>::eliminarRaiz()
{
    assert(r != NODO_NULO);    // Árbol no vacío.
    assert(r->hizq == NODO_NULO &&
           r->hder == NODO_NULO);    // La raíz es hoja.
    delete r;
    r = NODO_NULO;
}

```

```
template <typename T> inline bool Abin<T>::arbolVacio() const
{ return (r == NODO_NULO); }
```

```
template <typename T>
inline const T& Abin<T>::elemento(nodo n) const
{
    assert(n != NODO_NULO);
    return n->elto;
}
```

```
template <typename T>
inline T& Abin<T>::elemento(nodo n)
{
    assert(n != NODO_NULO);
    return n->elto;
}
```

```
template <typename T>
inline typename Abin<T>::nodo Abin<T>::raiz() const
{ return r; }
```

```

template <typename T> inline
typename Abin<T>::nodo Abin<T>::padre(nodo n) const
{
    assert(n != NODO_NULO);
    return n->padre;
}

template <typename T> inline
typename Abin<T>::nodo Abin<T>::hijoIzqdo(nodo n) const
{
    assert(n != NODO_NULO);
    return n->hizq;
}

template <typename T> inline
typename Abin<T>::nodo Abin<T>::hijoDrcho(nodo n) const
{
    assert(n != NODO_NULO);
    return n->hder;
}

```

```

template <typename T>
inline Abin<T>::Abin(const Abin<T>& A)
{
    r = copiar(A.r); // Copiar raíz y descendientes.
}

template <typename T>
Abin<T>& Abin<T>::operator =(const Abin<T>& A)
{
    if (this != &A) // Evitar autoasignación.
    {
        destruirNodos(r); // Vaciar el árbol.
        r = copiar(A.r); // Copiar raíz y descendientes.
    }
    return *this;
}

```



Los arboles ofrecen dos ventajas, representa jerarquias y permiten hacer busquedas en orden log. <- Importante.

```
template <typename T>
inline Abin<T>::~~Abin()
{
    destruirNodos(r);    // Vaciar el árbol.
}

/*-----*/
/* Métodos privados */
/*-----*/
// Destruye un nodo y todos sus descendientes
template <typename T>
void Abin<T>::destruirNodos(nodo& n)
{
    if (n != NODO_NULO) Condición de parada de la recursiva, para cuando n es NULO
    {
        destruirNodos(n->hizq);
        destruirNodos(n->hder);
        delete n;
        n = NODO_NULO;
    }
}
```

→ Mata primero a los hijos y luego al nodo padre

```

// Devuelve una copia de un nodo y todos sus descendientes
template <typename T>
typename Abin<T>::nodo Abin<T>::copiar(nodo n)
{
    nodo m = NODO_NULO;

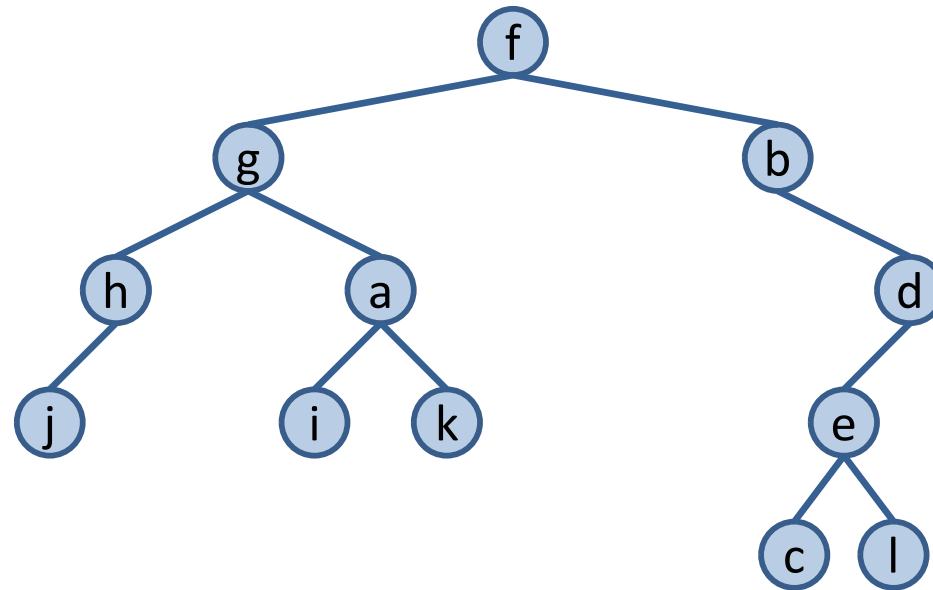
    if (n != NODO_NULO) {
        m = new celda(n->elto);    // Copiar n.
        m->hizq = copiar(n->hizq); // Copiar subárbol izqdo.
        if (m->hizq != NODO_NULO) m->hizq->padre = m;
        m->hder = copiar(n->hder); // Copiar subárbol drcho.
        if (m->hder != NODO_NULO) m->hder->padre = m;
    }
    return m;
}

#endif // ABIN_H

```

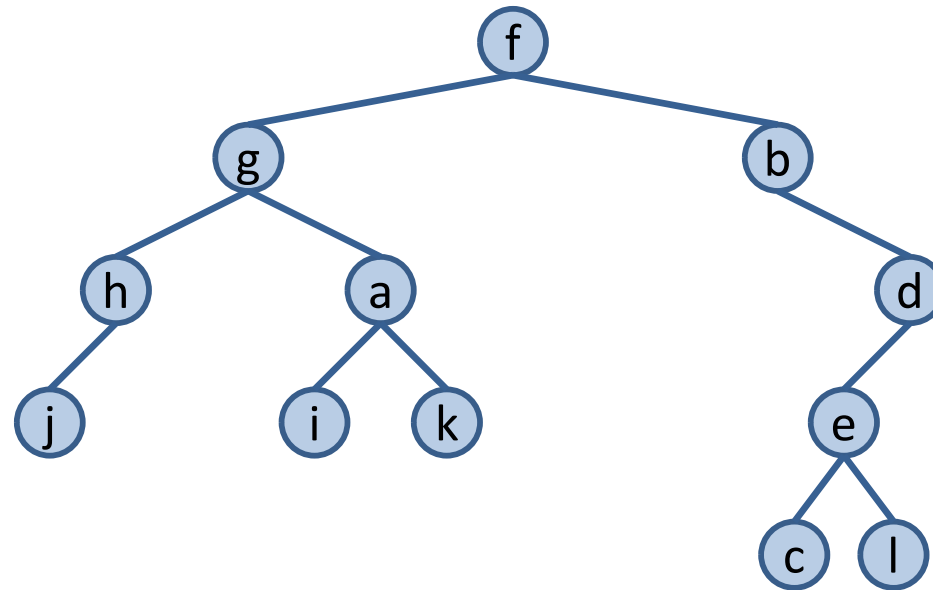
Hacerse una traza para ver como funciona la copia

# Implementación vectorial de árboles binarios



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	maxNodos-1
elto	f	g	a	b	d	h	j	i	k	e	l	c			
padre	*	0	1	0	3	1	5	2	2	4	9	9			

# Implementación vectorial de árboles binarios



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
elto	f	g	a	b	d	h	j	i	k	e	l	c			
padre	*	0	1	0	3	1	5	2	2	4	9	9			
hizq	1	5	7	*	9	6	*	*	*	11	*	*			
hder	3	2	8	4	*	*	*	*	*	10	*	*			

maxNodos-1

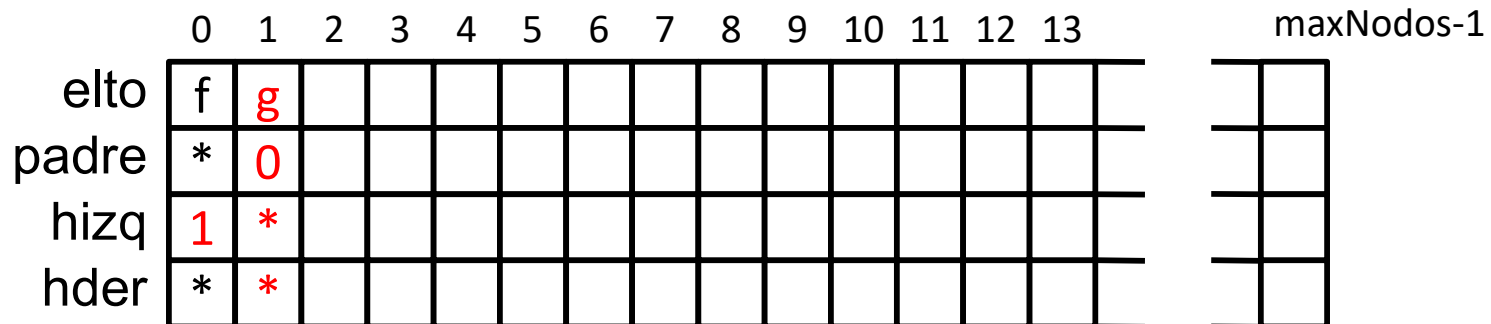

# Implementación vectorial de árboles binarios

## Inserción y eliminación



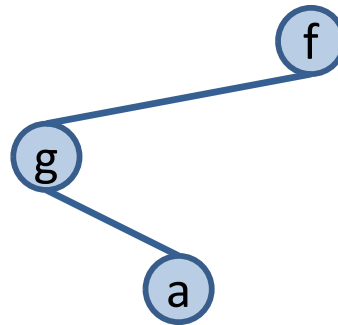
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
elto	f														
padre	*														
hizq	*														
hder	*														


# Inserción y eliminación



# Implementación vectorial de árboles binarios

## Inserción y eliminación

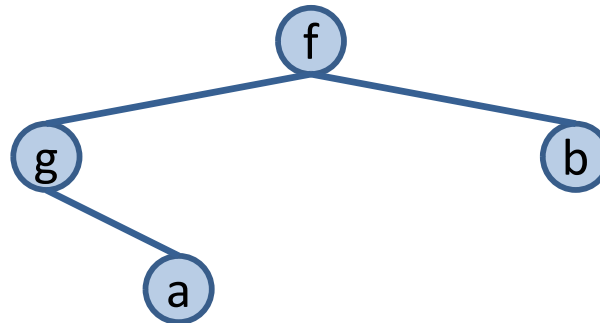


	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
elto	f	g	a												
padre	*	0	1												
hizq	1	*	*												
hder	*	2	*												


maxNodos-1

# Implementación vectorial de árboles binarios

## Inserción y eliminación

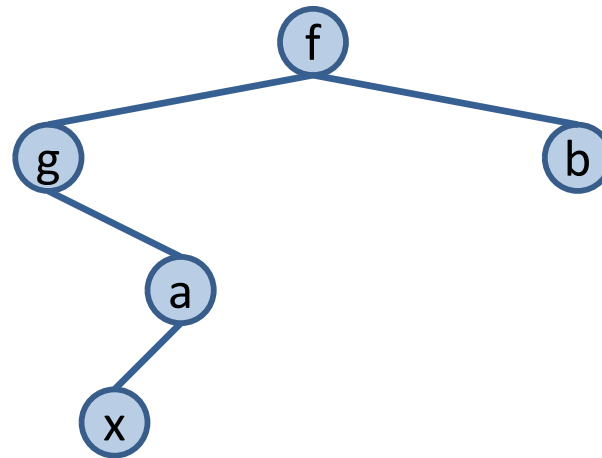


	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
elto	f	g	a	b											maxNodos-1
padre	*	0	1	0											
hizq	1	*	*	*											
hder	3	2	*	*											



# Implementación vectorial de árboles binarios

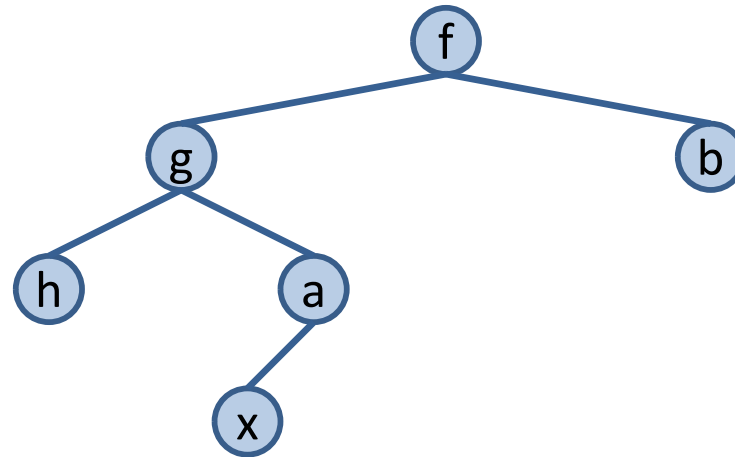
## Inserción y eliminación



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	maxNodos-1
elto	f	g	a	b	x										
padre	*	0	1	0	2										
hizq	1	*	4	*	*										
hder	3	2	*	*	*										

# Implementación vectorial de árboles binarios

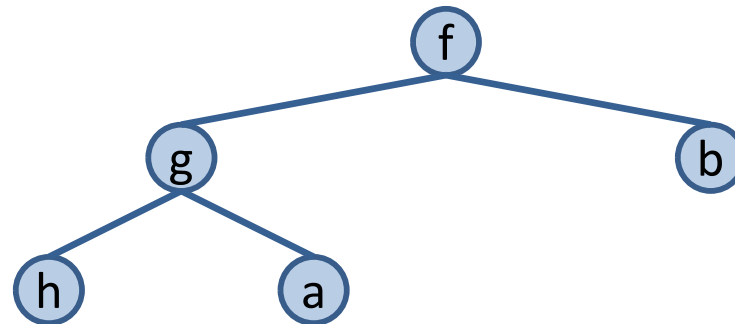
## Inserción y eliminación



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	maxNodos-1
elto	f	g	a	b	x	h									
padre	*	0	1	0	2	1									
hizq	1	5	4	*	*	*									
hder	3	2	*	*	*	*									

# Implementación vectorial de árboles binarios

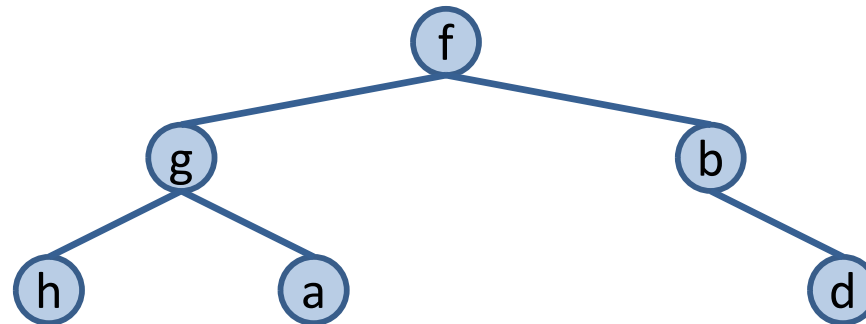
## Inserción y eliminación



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
elto	f	g	a	b		h									maxNodos-1
padre	*	0	1	0		1									
hizq	1	5	*	*		*									
hder	3	2	*	*		*									

# Implementación vectorial de árboles binarios

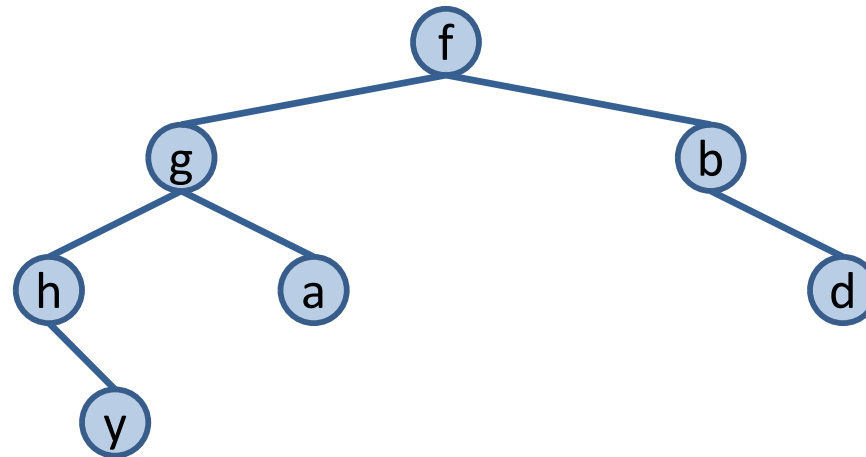
## Inserción y eliminación



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
elto	f	g	a	b	d	h									maxNodos-1
padre	*	0	1	0	3	1									
hizq	1	5	*	*	*	*									
hder	3	2	*	4	*	*									

# Implementación vectorial de árboles binarios

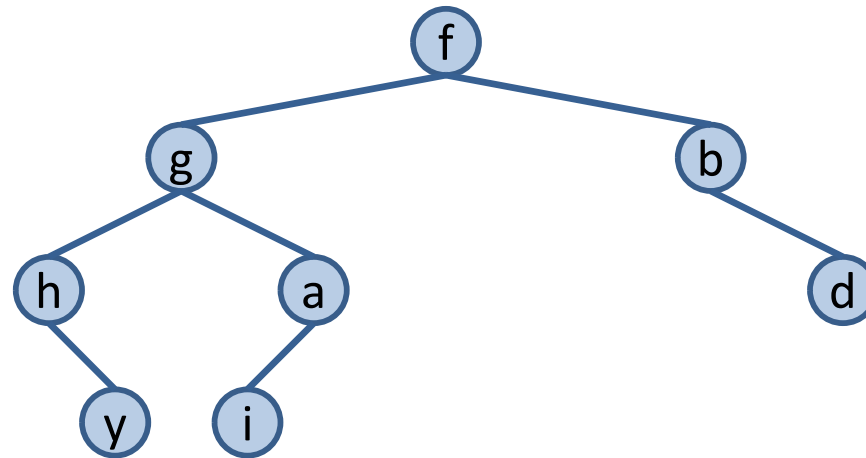
## Inserción y eliminación



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
elto	f	g	a	b	d	h	y								maxNodos-1
padre	*	0	1	0	3	1	5								
hizq	1	5	*	*	*	*	*								
hder	3	2	*	4	*	6	*								

# Implementación vectorial de árboles binarios

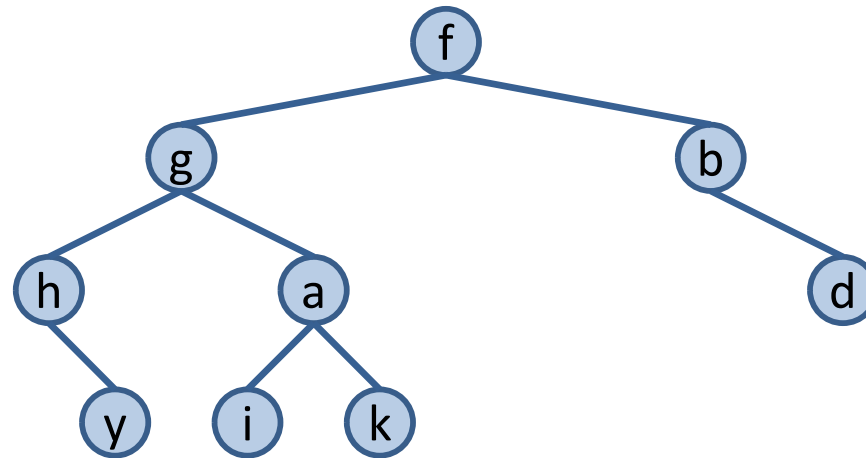
## Inserción y eliminación



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	maxNodos-1
elto	f	g	a	b	d	h	y	i							
padre	*	0	1	0	3	1	5	2							
hizq	1	5	7	*	*	*	*	*							
hder	3	2	*	4	*	6	*	*							

# Implementación vectorial de árboles binarios

## Inserción y eliminación

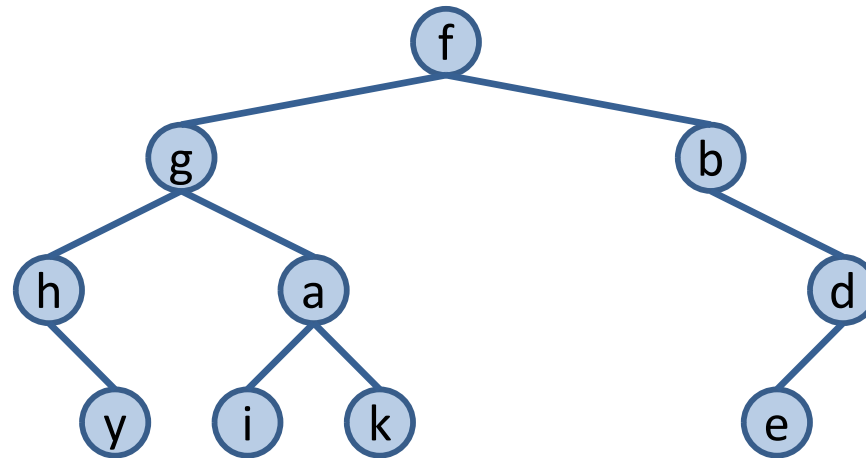


	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
elto	f	g	a	b	d	h	y	i	k						
padre	*	0	1	0	3	1	5	2	2						
hizq	1	5	7	*	*	*	*	*	*						
hder	3	2	8	4	*	6	*	*	*						

maxNodos-1


# Implementación vectorial de árboles binarios

## Inserción y eliminación

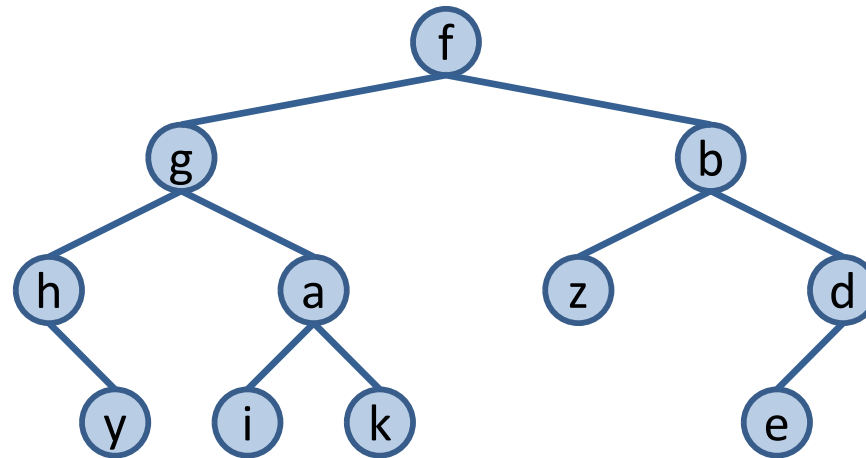


	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
elto	f	g	a	b	d	h	y	i	k	e					maxNodos-1
padre	*	0	1	0	3	1	5	2	2	4					
hizq	1	5	7	*	9	*	*	*	*	*					
hder	3	2	8	4	*	6	*	*	*	*					



# Implementación vectorial de árboles binarios

## Inserción y eliminación

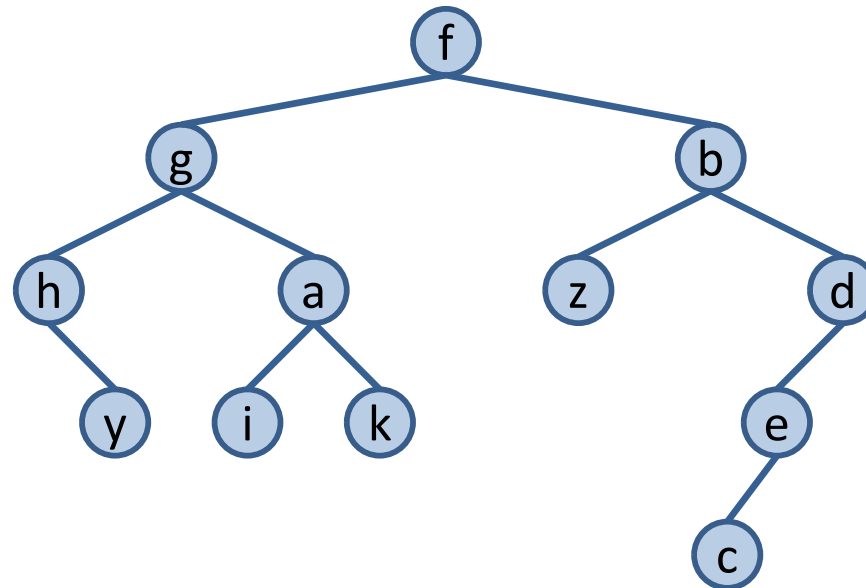


	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
elto	f	g	a	b	d	h	y	i	k	e	z				
padre	*	0	1	0	3	1	5	2	2	4	3				
hizq	1	5	7	10	9	*	*	*	*	*	*				
hder	3	2	8	4	*	6	*	*	*	*	*				

maxNodos-1


# Implementación vectorial de árboles binarios

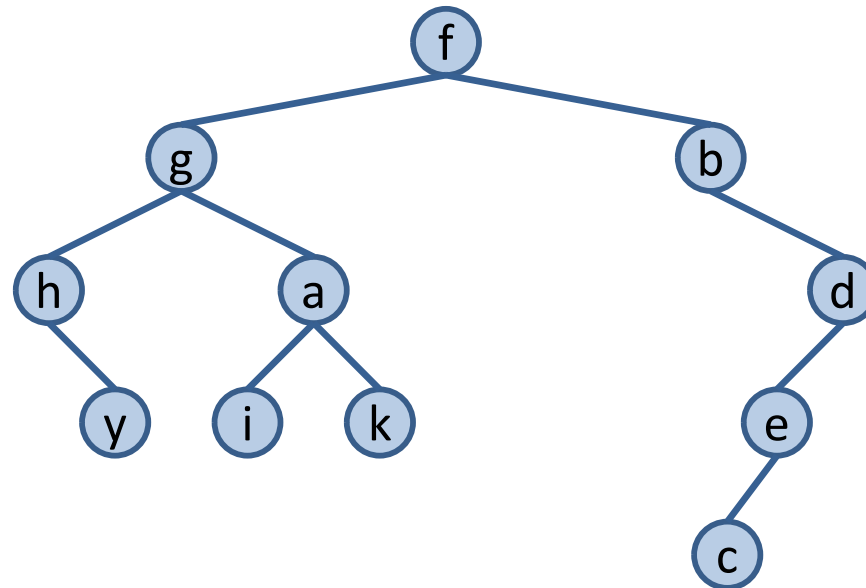
## Inserción y eliminación



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	maxNodos-1
elto	f	g	a	b	d	h	y	i	k	e	z	c			
padre	*	0	1	0	3	1	5	2	2	4	3	9			
hizq	1	5	7	10	9	*	*	*	*	11	*	*			
hder	3	2	8	4	*	6	*	*	*	*	*	*			

# Implementación vectorial de árboles binarios

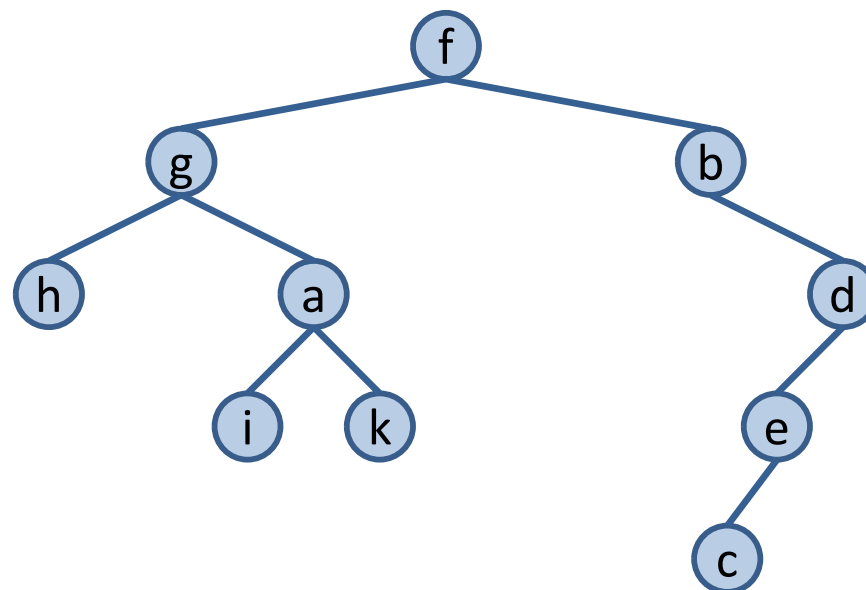
## Inserción y eliminación



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	maxNodos-1
elto	f	g	a	b	d	h	y	i	k	e		c			
padre	*	0	1	0	3	1	5	2	2	4		9			
hizq	1	5	7	*	9	*	*	*	*	11		*			
hder	3	2	8	4	*	6	*	*	*	*		*			

# Implementación vectorial de árboles binarios

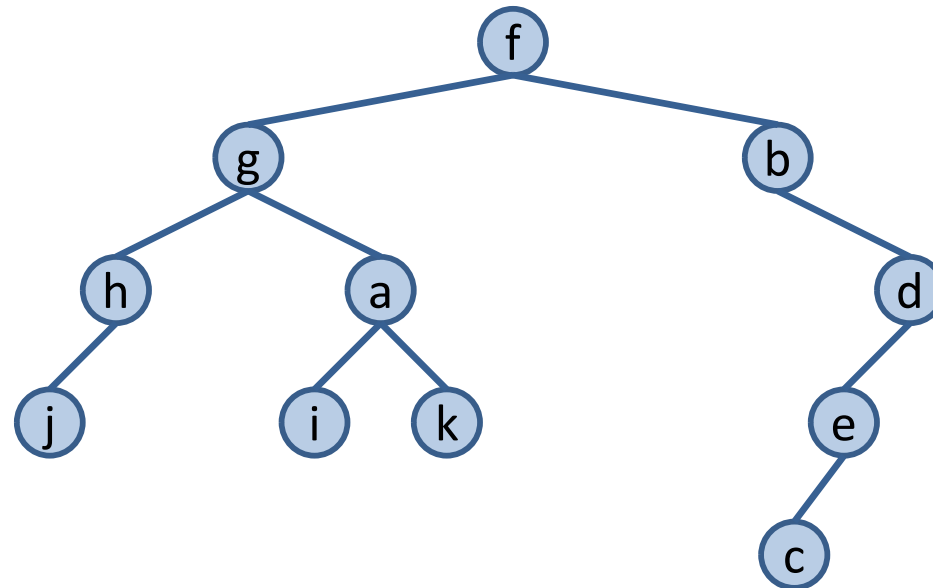
## Inserción y eliminación



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	maxNodos-1
elto	f	g	a	b	d	h		i	k	e		c			
padre	*	0	1	0	3	1		2	2	4		9			
hizq	1	5	7	*	9	*		*	*	11		*			
hder	3	2	8	4	*	*		*	*	*		*			

# Implementación vectorial de árboles binarios

## Inserción y eliminación

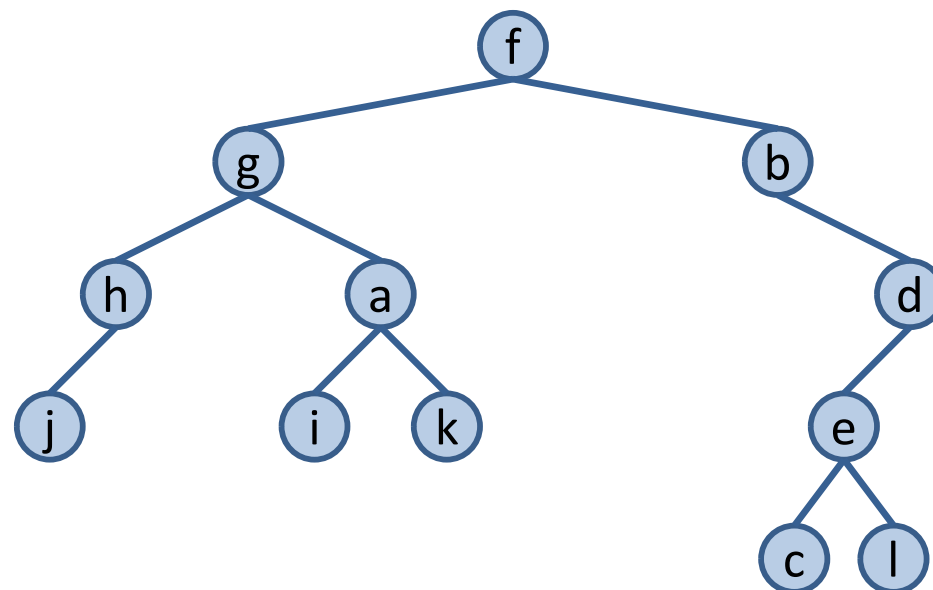


	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
elto	f	g	a	b	d	h	j	i	k	e		c			
padre	*	0	1	0	3	1	5	2	2	4		9			
hizq	1	5	7	*	9	6	*	*	*	11		*			
hder	3	2	8	4	*	*	*	*	*	*		*			

maxNodos-1


# Implementación vectorial de árboles binarios

## Inserción y eliminación



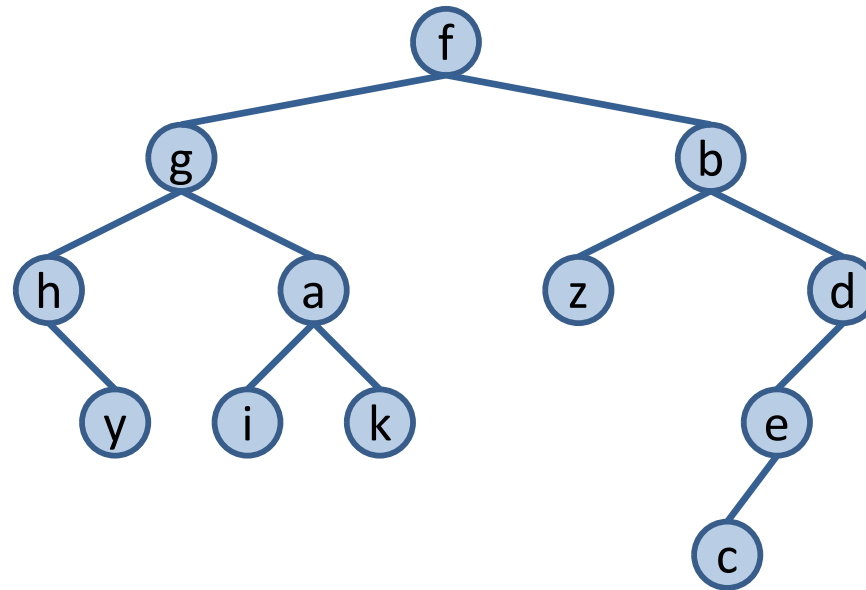
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
elto	f	g	a	b	d	h	j	i	k	e	l	c			
padre	*	0	1	0	3	1	5	2	2	4	9	9			
hizq	1	5	7	*	9	6	*	*	*	11	*	*			
hder	3	2	8	4	*	*	*	*	*	10	*	*			

maxNodos-1


# Implementación vectorial de árboles binarios

## Inserción y eliminación

(distinción entre celdas libres y ocupadas)

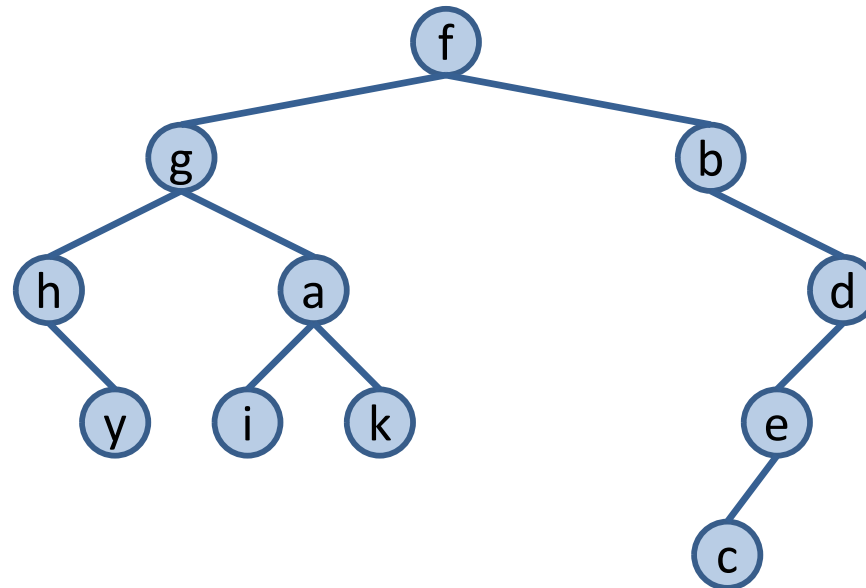


	0	1	2	3	4	5	6	7	8	9	10	11	12	13	maxNodos-1
elto	f	g	a	b	d	h	y	i	k	e	z	c			
padre	*	0	1	0	3	1	5	2	2	4	3	9			
hizq	1	5	7	10	9	*	*	*	*	11	*	*			
hder	3	2	8	4	*	6	*	*	*	*	*	*			

# Implementación vectorial de árboles binarios

## Inserción y eliminación

(distinción entre celdas libres y ocupadas)



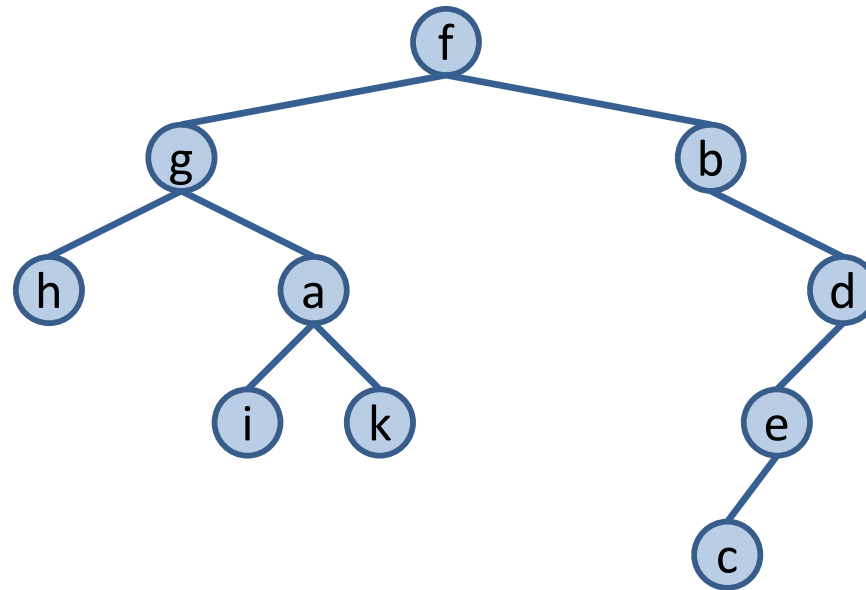
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
elto	f	g	a	b	d	h	y	i	k	e	z	c			
padre	0	0	1	0	3	1	5	2	2	4	*	9	*	*	
hizq	1	5	7	*	9	*	*	*	*	11	*	*			
hder	3	2	8	4	*	6	*	*	*	*	*	*			

	maxNodos-1
	*



# Implementación vectorial de árboles binarios

Inserción y eliminación  
(distinción entre celdas libres y ocupadas)



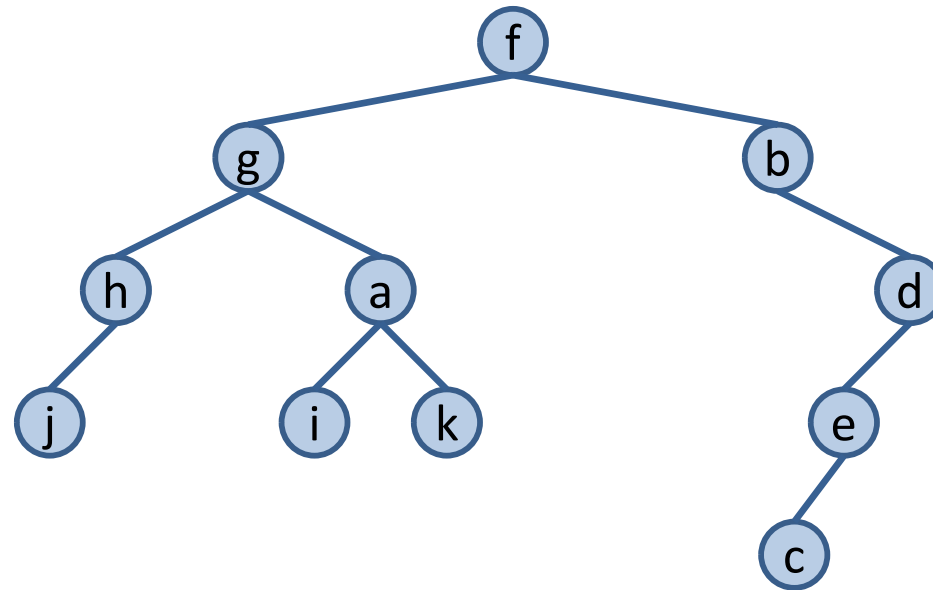
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
elto	f	g	a	b	d	h	y	i	k	e	z	c			
padre	0	0	1	0	3	1	*	2	2	4	*	9	*	*	
hizq	1	5	7	*	9	*	*	*	*	11	*	*			
hder	3	2	8	4	*	*	*	*	*	*	*	*			


# Implementación vectorial de árboles binarios

## Inserción y eliminación

(distinción entre celdas libres y ocupadas)



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
elto	f	g	a	b	d	h	j	i	k	e	z	c			
padre	0	0	1	0	3	1	5	2	2	4	*	9	*	*	
hizq	1	5	7	*	9	6	*	*	*	11	*	*			
hder	3	2	8	4	*	*	*	*	*	*	*	*			

maxNodos-1

	*

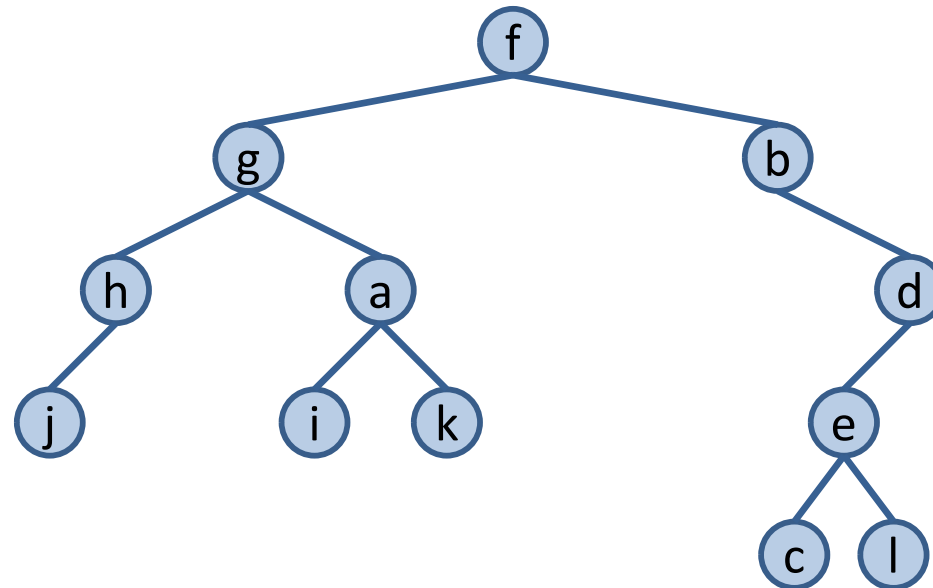
# Implementación vectorial de árboles binarios

## Inserción y eliminación

(distinción entre celdas libres y ocupadas)

Inserción  $O(n)$

Eliminación  $O(1)$



	0	1	2	3	4	5	6	7	8	9	10	11	12	13
elto	f	g	a	b	d	h	j	i	k	e	l	c		
padre	0	0	1	0	3	1	5	2	2	4	9	9	*	*
hizq	1	5	7	*	9	6	*	*	*	11	*	*		
hder	3	2	8	4	*	*	*	*	*	10	*	*		

maxNodos-1

	*

# Implementación vectorial de árboles binarios

Inserción y eliminación eficientes

f

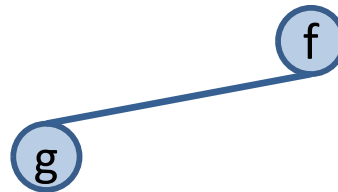
numNodos 1

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
elto	f														
padre	*														
hizq	*														
hder	*														


maxNodos-1

# Implementación vectorial de árboles binarios

Inserción y eliminación eficientes



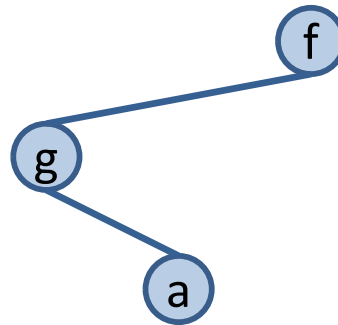
numNodos 2

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
elto	f	g													
padre	*	0													
hizq	1	*													
hder	*	*													


maxNodos-1

# Implementación vectorial de árboles binarios

Inserción y eliminación eficientes

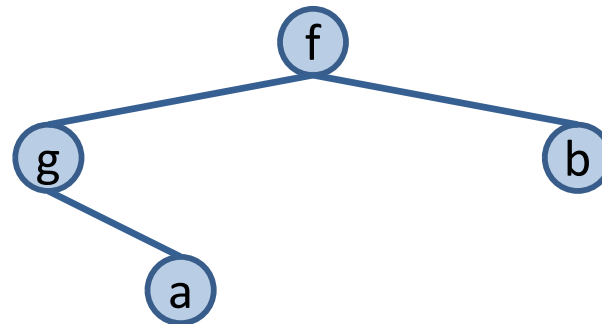


numNodos 3

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	maxNodos-1
elto	f	g	a												
padre	*	0	1												
hizq	1	*	*												
hder	*	2	*												

# Implementación vectorial de árboles binarios

Inserción y eliminación eficientes



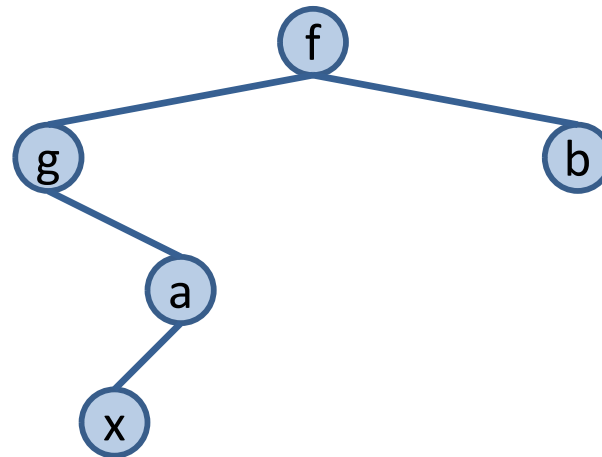
numNodos 4

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
elto	f	g	a	b											
padre	*	0	1	0											
hizq	1	*	*	*											
hder	3	2	*	*											

maxNodos-1


# Implementación vectorial de árboles binarios

Inserción y eliminación eficientes



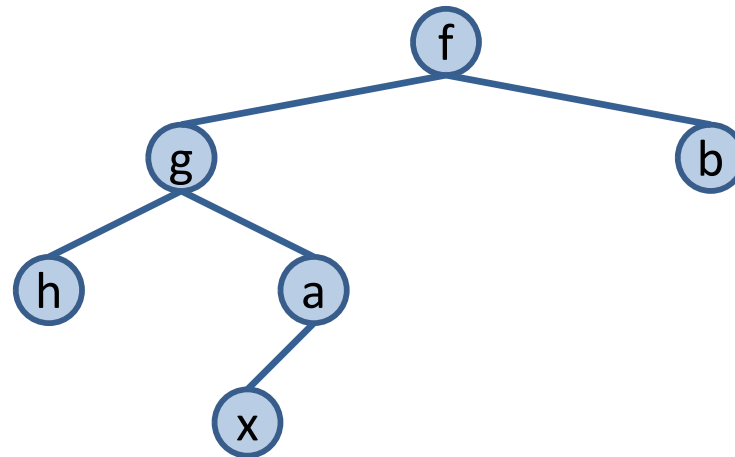
numNodos 5

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	maxNodos-1
elto	f	g	a	b	x										
padre	*	0	1	0	2										
hizq	1	*	4	*	*										
hder	3	2	*	*	*										



# Implementación vectorial de árboles binarios

Inserción y eliminación eficientes

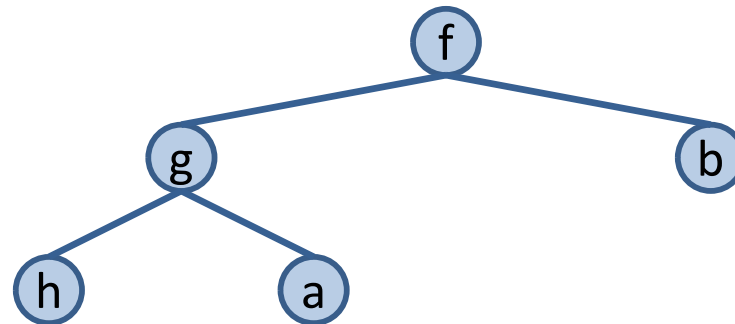


numNodos 6

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	maxNodos-1
elto	f	g	a	b	x	h									
padre	*	0	1	0	2	1									
hizq	1	5	4	*	*	*									
hder	3	2	*	*	*	*									

# Implementación vectorial de árboles binarios

Inserción y eliminación eficientes

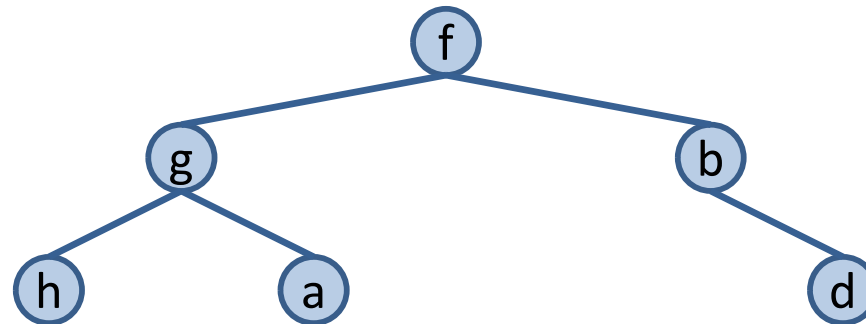


numNodos 5

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
elto	f	g	a	b	h	h									maxNodos-1
padre	*	0	1	0	1	1									
hizq	1	4	*	*	*	*									
hder	3	2	*	*	*	*									

# Implementación vectorial de árboles binarios

Inserción y eliminación eficientes



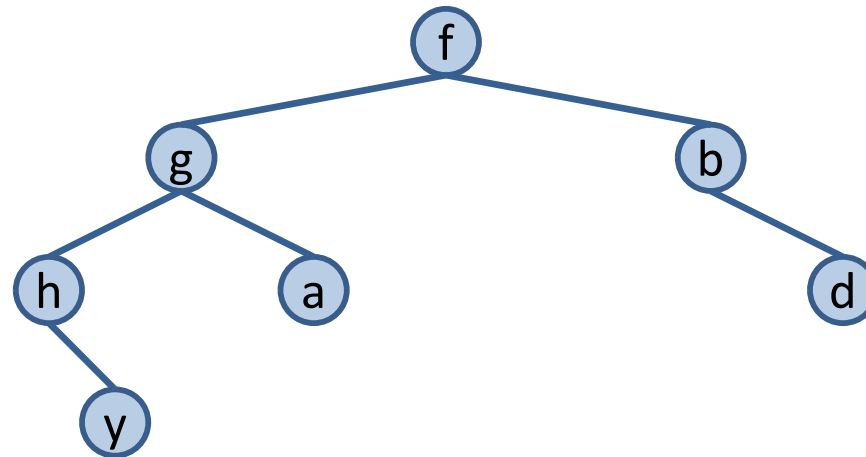
numNodos 6

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
elto	f	g	a	b	h	d									
padre	*	0	1	0	1	3									
hizq	1	4	*	*	*	*									
hder	3	2	*	5	*	*									


maxNodos-1

# Implementación vectorial de árboles binarios

Inserción y eliminación eficientes

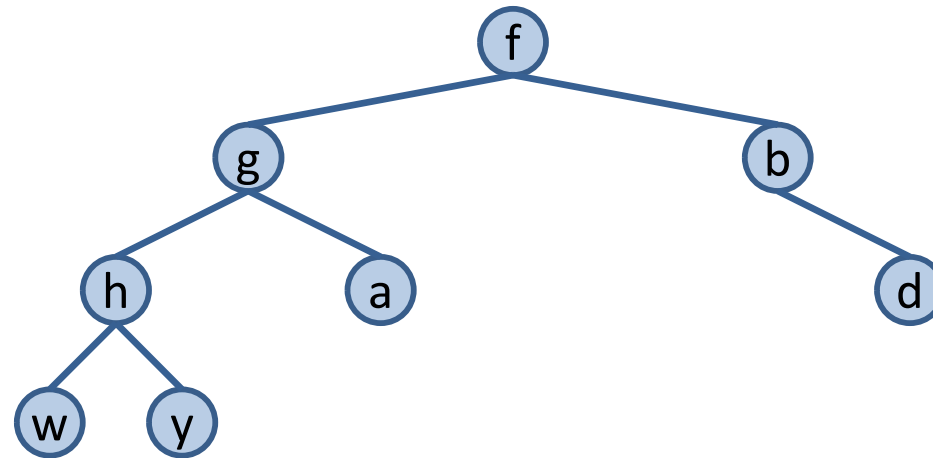


numNodos 7

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	maxNodos-1
elto	f	g	a	b	h	d	y								
padre	*	0	1	0	1	3	4								
hizq	1	4	*	*	*	*	*								
hder	3	2	*	5	6	*	*								

# Implementación vectorial de árboles binarios

Inserción y eliminación eficientes

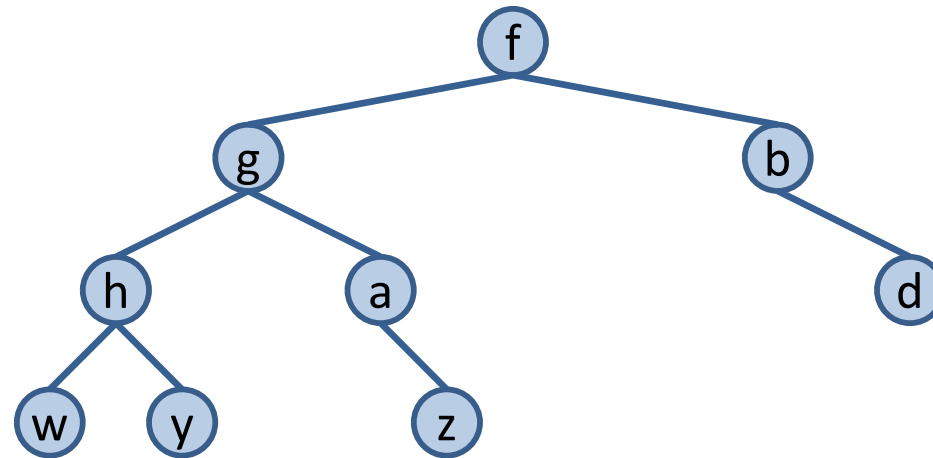


numNodos 8

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	maxNodos-1
elto	f	g	a	b	h	d	y	w							
padre	*	0	1	0	1	3	4	4							
hizq	1	4	*	*	7	*	*	*							
hder	3	2	*	5	6	*	*	*							

# Implementación vectorial de árboles binarios

Inserción y eliminación eficientes



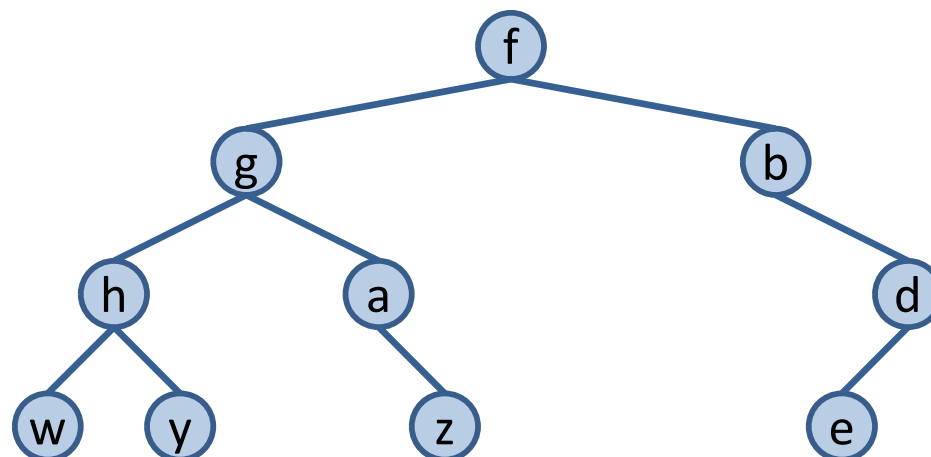
numNodos 9

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
elto	f	g	a	b	h	d	y	w	z						
padre	*	0	1	0	1	3	4	4	2						
hizq	1	4	*	*	7	*	*	*	*						
hder	3	2	8	5	6	*	*	*	*						

maxNodos-1


# Implementación vectorial de árboles binarios

Inserción y eliminación eficientes



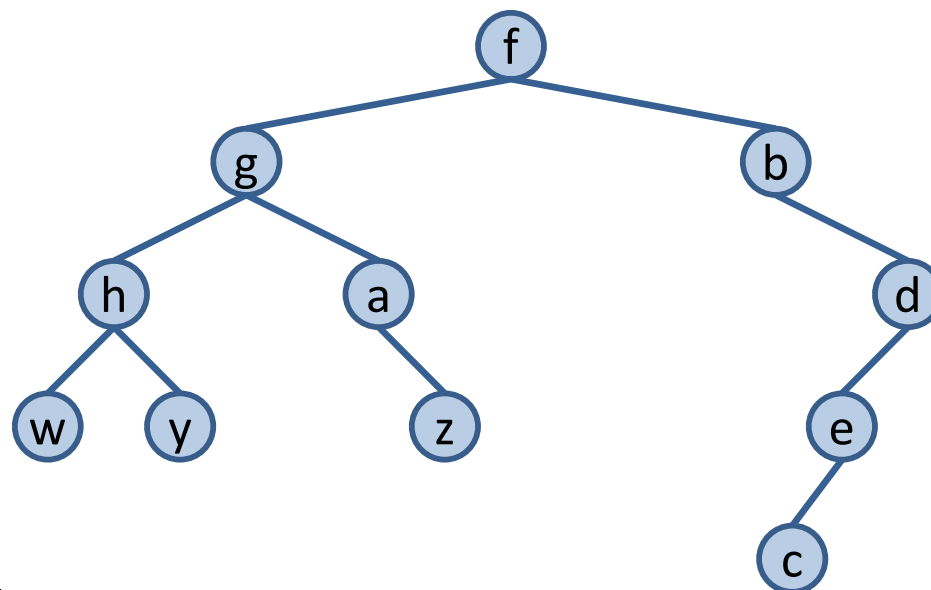
numNodos 10

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
elto	f	g	a	b	h	d	y	w	z	e					
padre	*	0	1	0	1	3	4	4	2	5					
hizq	1	4	*	*	7	9	*	*	*	*					
hder	3	2	8	5	6	*	*	*	*	*					

maxNodos-1


# Implementación vectorial de árboles binarios

Inserción y eliminación eficientes



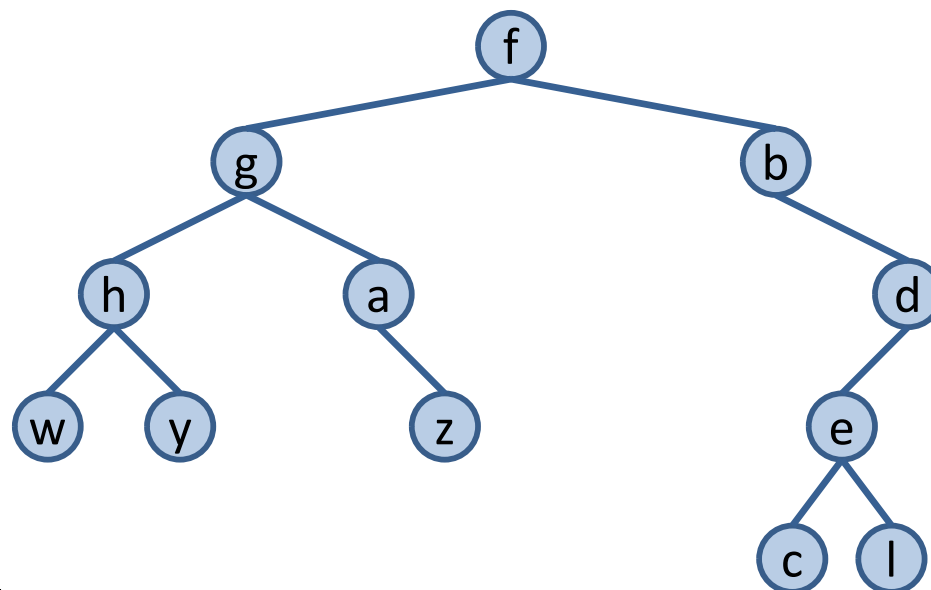
numNodos 11

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	maxNodos-1
elto	f	g	a	b	h	d	y	w	z	e	c				
padre	*	0	1	0	1	3	4	4	2	5	9				
hizq	1	4	*	*	7	9	*	*	*	10	*				
hder	3	2	8	5	6	*	*	*	*	*	*				



# Implementación vectorial de árboles binarios

Inserción y eliminación eficientes

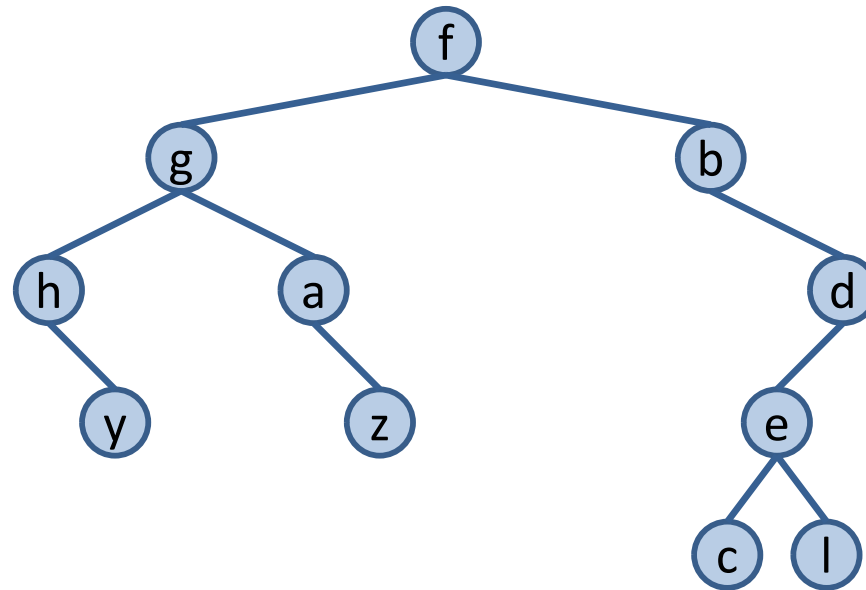


numNodos 12

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	maxNodos-1
elto	f	g	a	b	h	d	y	w	z	e	c	l			
padre	*	0	1	0	1	3	4	4	2	5	9	9			
hizq	1	4	*	*	7	9	*	*	*	10	*	*			
hder	3	2	8	5	6	*	*	*	*	11	*	*			

# Implementación vectorial de árboles binarios

Inserción y eliminación eficientes

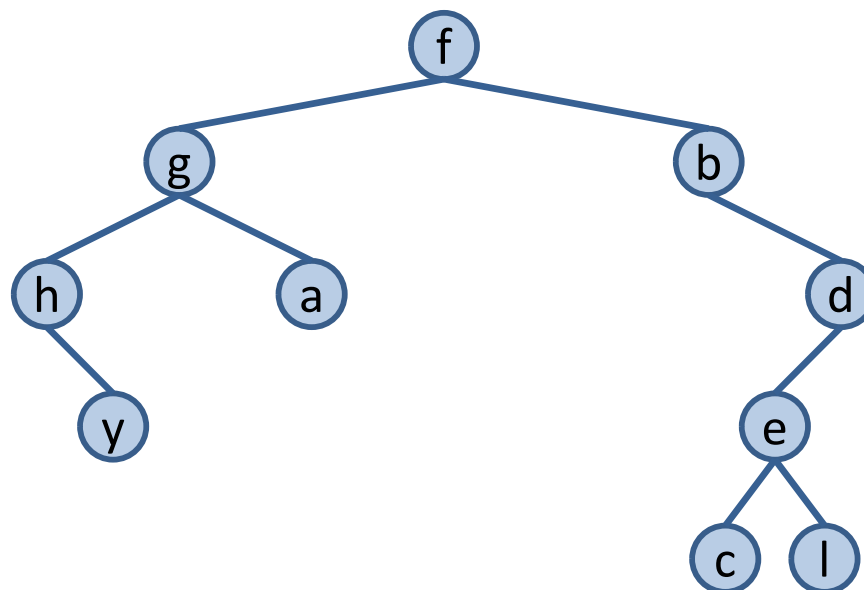


numNodos 11

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	maxNodos-1
elto	f	g	a	b	h	d	y	l	z	e	c	l			
padre	*	0	1	0	1	3	4	9	2	5	9	9			
hizq	1	4	*	*	*	9	*	*	*	10	*	*			
hder	3	2	8	5	6	*	*	*	*	7	*	*			

# Implementación vectorial de árboles binarios

Inserción y eliminación eficientes

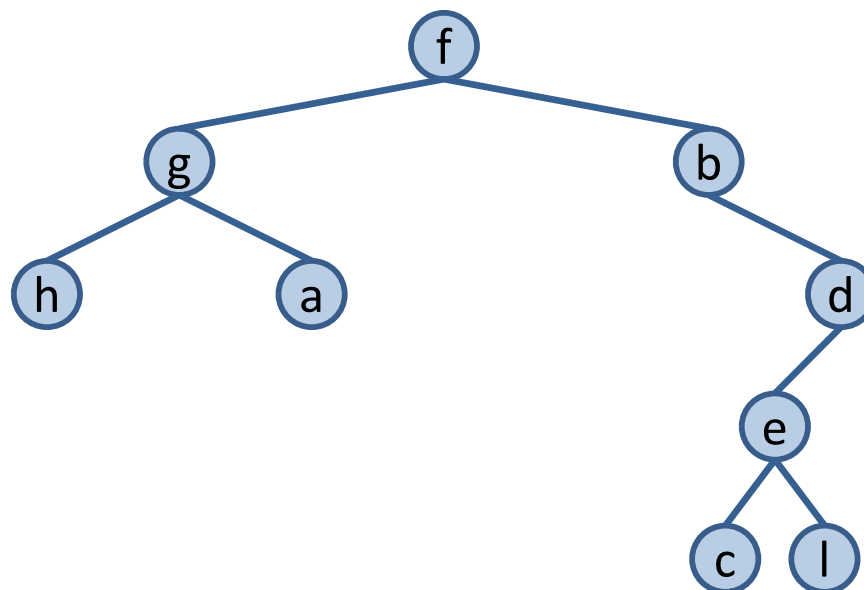


numNodos 10

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	maxNodos-1
elto	f	g	a	b	h	d	y	l	c	e	c	l			
padre	*	0	1	0	1	3	4	9	9	5	9	9			
hizq	1	4	*	*	*	9	*	*	*	8	*	*			
hder	3	2	*	5	6	*	*	*	*	7	*	*			

# Implementación vectorial de árboles binarios

Inserción y eliminación eficientes

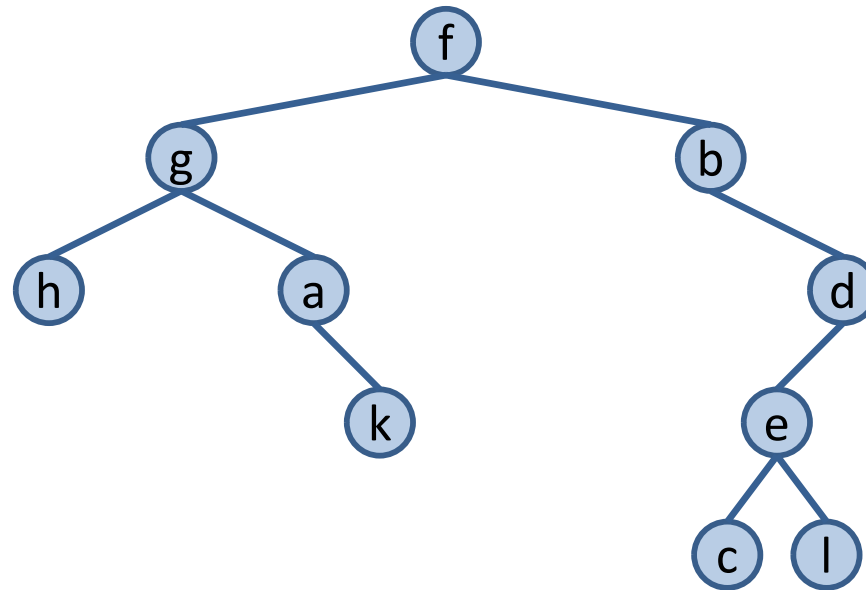


numNodos 9

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	maxNodos-1
elto	f	g	a	b	h	d	e	l	c	e	c	l			
padre	*	0	1	0	1	3	5	6	6	5	9	9			
hizq	1	4	*	*	*	6	8	*	*	8	*	*			
hder	3	2	*	5	*	*	7	*	*	7	*	*			

# Implementación vectorial de árboles binarios

Inserción y eliminación eficientes



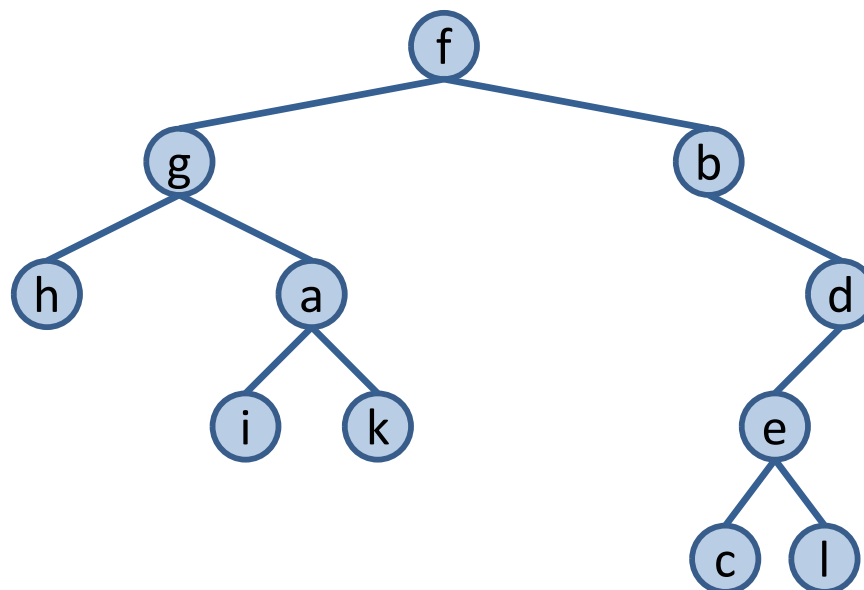
numNodos 10

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
elto	f	g	a	b	h	d	e	l	c	k	c	l			
padre	*	0	1	0	1	3	5	6	6	2	9	9			
hizq	1	4	*	*	*	6	8	*	*	*	*	*			
hder	3	2	9	5	*	*	7	*	*	*	*	*			

maxNodos-1


# Implementación vectorial de árboles binarios

Inserción y eliminación eficientes



numNodos 11

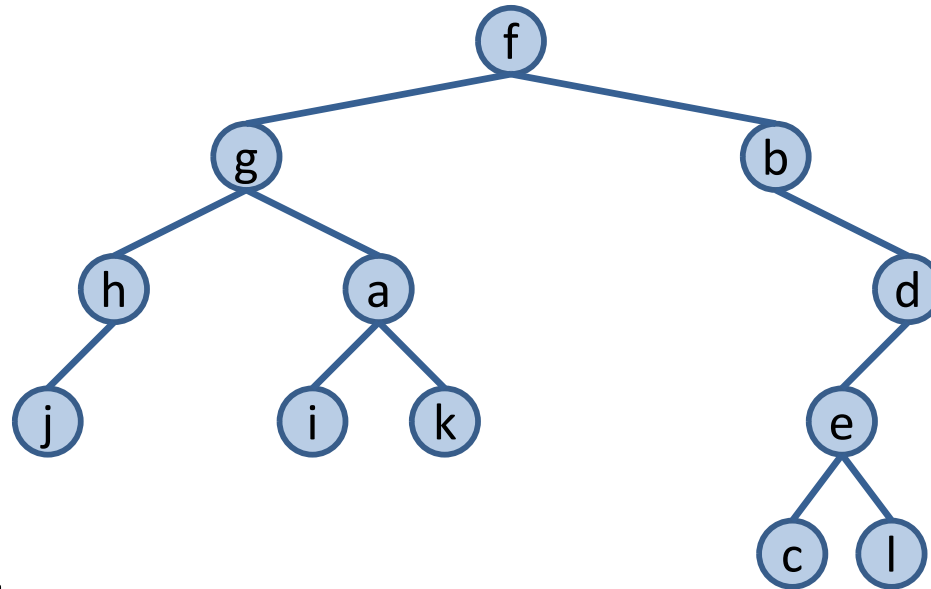
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	maxNodos-1
elto	f	g	a	b	h	d	e	l	c	k	i	l			
padre	*	0	1	0	1	3	5	6	6	2	2	9			
hizq	1	4	10	*	*	6	8	*	*	*	*	*			
hder	3	2	9	5	*	*	7	*	*	*	*	*			

# Implementación vectorial de árboles binarios

Inserción y eliminación eficientes

Inserción  $O(1)$

Eliminación  $O(1)$



numNodos 12

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	maxNodos-1
elto	f	g	a	b	h	d	e	l	c	k	i	j			
padre	*	0	1	0	1	3	5	6	6	2	2	4			
hizq	1	4	10	*	11	6	8	*	*	*	*	*			
hder	3	2	9	5	*	*	7	*	*	*	*	*			

```

#ifndef ABIN_VEC0_H
#define ABIN_VEC0_H
#include <cassert>

template <typename T> class Abin {
public:
    typedef size_t nodo; // Índice del vector entre 0 y maxNodos-1
    static const nodo NODO_NULO;

    explicit Abin(size_t maxNodos); // Ctor., requiere ctor. T()
    void insertarRaiz(const T& e);
    void insertarHijoIzqdo(nodo n, const T& e);
    void insertarHijoDrcho(nodo n, const T& e);
    void eliminarHijoIzqdo(nodo n);
    void eliminarHijoDrcho(nodo n);
    void eliminarRaiz();
    bool arbolVacio() const;
    const T& elemento(nodo n) const; // Lec. en Abin const
    T& elemento(nodo n); // Lec/Esc. en Abin no-const

```



```

nodo raiz() const;
nodo padre(nodo n) const;
nodo hijoIzqdo(nodo n) const;
nodo hijoDrcho(nodo n) const;
Abin(const Abin<T>& A); // Ctor. copia, req ctor T()
Abin<T>& operator =(const Abin<T>& A); // Asig. de árboles
~Abin(); // Destructor

private:
    struct celda {
        T elto;
        nodo padre, hizq, hder;
    };
    celda *nodos; // Vector de nodos
    size_t maxNodos; // Tamaño del vector
    size_t numNodos; // Número de nodos del árbol
};

/* Definición del nodo nulo */
template <typename T>
const typename Abin<T>::nodo Abin<T>::NODO_NULO(SIZE_MAX);

```

```

template <typename T>
inline Abin<T>::Abin(size_t maxNodos) :
    nodos(new celda[maxNodos]),
    maxNodos(maxNodos),
    numNodos(0)
{
}

template <typename T>
inline void Abin<T>::insertarRaiz(const T& e)
{
    assert(numNodos == 0);    // Árbol vacío

    numNodos = 1;
    nodos[0].elto = e;
    nodos[0].padre = NODO_NULO;
    nodos[0].hizq = NODO_NULO;
    nodos[0].hder = NODO_NULO;
}

```

```

template <typename T>
inline void Abin<T>::insertarHijoIzqdo(nodo n, const T& e)
{
    assert(n >= 0 && n < numNodos); // Nodo válido
    assert(nodos[n].hizq == NODO_NULO); // n no tiene hijo izqdo.
    assert(numNodos < maxNodos); // Árbol no lleno

    // Añadir el nuevo nodo al final de la secuencia.
    nodos[n].hizq = numNodos;
    nodos[numNodos].elto = e;
    nodos[numNodos].padre = n;
    nodos[numNodos].hizq = NODO_NULO;
    nodos[numNodos].hder = NODO_NULO;
    ++numNodos;
}

```

```

template <typename T>
inline void Abin<T>::insertarHijoDrcho(nodo n, const T& e)
{
    assert(n >= 0 && n < numNodos); // Nodo válido
    assert(nodos[n].hder == NODO_NULO); // n no tiene hijo drcho.
    assert(numNodos < maxNodos); // Árbol no lleno

    // Añadir el nuevo nodo al final de la secuencia.
    nodos[n].hder = numNodos;
    nodos[numNodos].elto = e;
    nodos[numNodos].padre = n;
    nodos[numNodos].hizq = NODO_NULO;
    nodos[numNodos].hder = NODO_NULO;
    ++numNodos;
}

```

```

template <typename T>
void Abin<T>::eliminarHijoIzqdo(nodo n)
{
    nodo hizqdo ;

    assert(n >= 0 && n < numNodos);    // Nodo válido
    hizqdo = nodos[n].hizq;
    assert(hizqdo != NODO_NULO);        // Existe hijo izqdo. de n.
    assert(nodos[hizqdo].hizq == NODO_NULO &&    // Hijo izqdo. de
           nodos[hizqdo].hder == NODO_NULO);    // n es hoja.

    if (hizqdo != numNodos-1)
    {
        // Mover el último nodo a la posición del hijo izqdo.
        nodos[hizqdo] = nodos[numNodos-1];
        // Actualizar la posición del hijo (izquierdo o derecho)
        // en el padre del nodo movido.
        if (nodos[nodos[hizqdo].padre].hizq == numNodos-1)
            nodos[nodos[hizqdo].padre].hizq = hizqdo;
        else
            nodos[nodos[hizqdo].padre].hder = hizqdo;
    }
}

```

```

    // Si el nodo movido tiene hijos,
    // actualizar la posición del padre.
    if (nodos[hizqdo].hizq != NODO_NULO)
        nodos[nodos[hizqdo].hizq].padre = hizqdo;
    if (nodos[hizqdo].hder != NODO_NULO)
        nodos[nodos[hizqdo].hder].padre = hizqdo;
}
nodos[n].hizq = NODO_NULO;
--numNodos;
}

```

```

template <typename T>
void Abin<T>::eliminarHijoDrcho(nodo n)
{
    nodo hdrcho;

    assert(n >= 0 && n < numNodos); // Nodo válido
    hdrcho = nodos[n].hder;
    assert(hdrcho != NODO_NULO);      // Existe hijo drcho. de n.
    assert(nodos[hdrcho].hizq == NODO_NULO && // Hijo drcho. de
           nodos[hdrcho].hder == NODO_NULO); // n es hoja.

    if (hdrcho != numNodos-1)
    {
        // Mover el último nodo a la posición del hijo drcho.
        nodos[hdrcho] = nodos[numNodos-1];
        // Actualizar la posición del hijo (izquierdo o derecho)
        // en el padre del nodo movido.
        if (nodos[nodos[hdrcho].padre].hizq == numNodos-1)
            nodos[nodos[hdrcho].padre].hizq = hdrcho;
        else
            nodos[nodos[hdrcho].padre].hder = hdrcho;
    }
}

```

```

    // Si el nodo movido tiene hijos,
    // actualizar la posición del padre.
    if (nodos[hdrcho].hizq != NODO_NULO)
        nodos[nodos[hdrcho].hizq].padre = hdrcho;
    if (nodos[hdrcho].hder != NODO_NULO)
        nodos[nodos[hdrcho].hder].padre = hdrcho;
}
nodos[n].hder = NODO_NULO;
--numNodos;
}

template <typename T>
inline void Abin<T>::eliminarRaiz()
{
    assert(numNodos == 1);
    numNodos = 0;
}

```



```

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

```

```

template <typename T>
inline const T& Abin<T>::elemento(nodo n) const
{
    assert(n >= 0 && n < numNodos);
    return nodos[n].elto;
}

```

```

template <typename T>
inline T& Abin<T>::elemento(nodo n)
{
    assert(n >= 0 && n < numNodos);
    return nodos[n].elto;
}

```

```

template <typename T>
inline typename Abin<T>::nodo Abin<T>::raiz() const
{
    return (numNodos > 0) ? 0 : NODO_NULO;
}

```

```

template <typename T> inline
typename Abin<T>::nodo Abin<T>::padre(nodo n) const
{
    assert(n >= 0 && n < numNodos);
    return nodos[n].padre;
}

```

```

template <typename T> inline
typename Abin<T>::nodo Abin<T>::hijoIzqdo(nodo n) const
{
    assert(n >= 0 && n < numNodos);
    return nodos[n].hizq;
}

```

```

template <typename T> inline
typename Abin<T>::nodo Abin<T>::hijoDrcho(nodo n) const
{
    assert(n >= 0 && n < numNodos);
    return nodos[n].hder;
}

```

```

template <typename T>
Abin<T>::Abin(const Abin<T>& A) :
    nodos(new celda[A.maxNodos]),
    maxNodos(A.maxNodos),
    numNodos(A.numNodos)
{
    // Copiar el vector.
    for (nodo n = 0; n <= numNodos-1; n++)
        nodos[n] = a.nodos[n];
}

```

```

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

```

```

template <typename T>
Abin<T>& Abin<T>::operator =(const Abin<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 celda[maxNodos];
        }
        // Copiar el vector.
        numNodos = A.numNodos;
        for (nodo n = 0; n <= numNodos-1; n++)
            nodos[n] = A.nodos[n];
    }
    return *this;
}

#endif // ABIN_VEC0_H

```

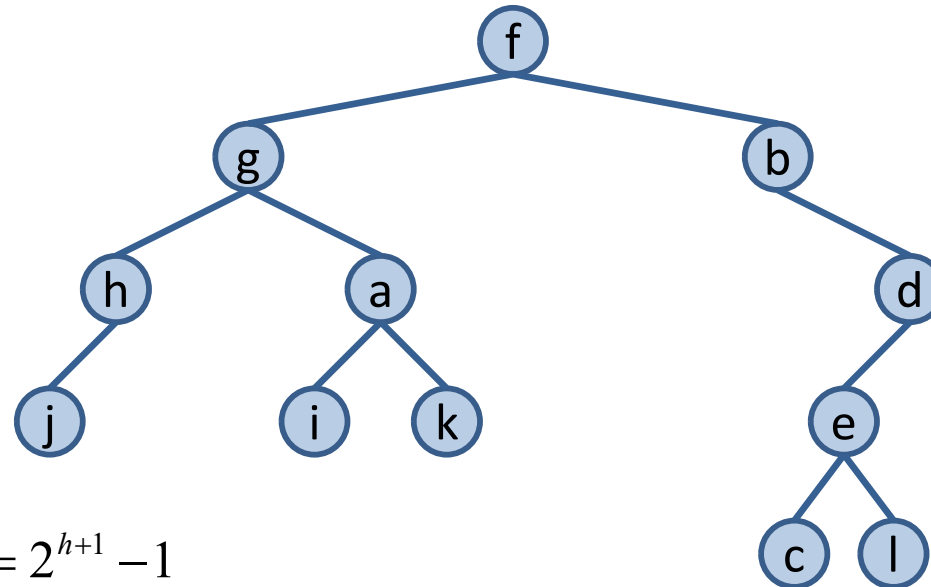
# Implementación de un árbol binario mediante un vector de posiciones relativas

Mediante esta implementación no nos hará falta almacenar punteros para poder acceder a los nodos (Hijos y padre).

Es decir, podemos acceder a ellos pero sin tenerlos almacenados.

Cálculo de las posiciones de los nodos

$$\begin{aligned} \text{hizq}(i) &= 2i+1 \\ \text{hder}(i) &= 2i+2 \\ \text{padre}(i) &= (i-1)/2 \\ &\quad \text{division entera} \end{aligned}$$



La altura 'h' es el numero de niveles -1, pero como nuestros niveles empiezan en 0, coincide con el nivel más profundo.

Altura

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

Ahora guardamos el vector solamente con el contenido de los nodos

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
nodos	f	g	b	h	a		d	j		i	k			e		

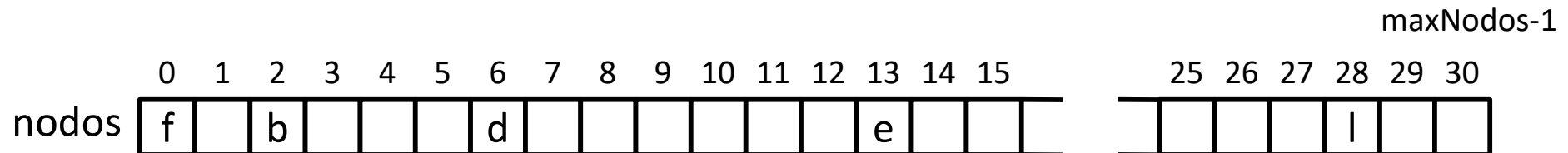
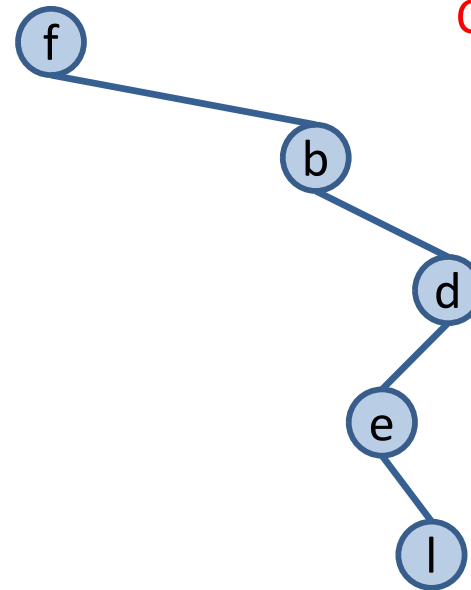
	25	26	27	28	29	30
				c	l	

si vemos, se cumple que los hijos de a (4) son i (9) y k (10) ->  $2 \cdot 4 + 1 = 9$  y  $2 \cdot 4 + 2 = 10$ .  
Además el padre es g (1) ->  $(4-1)/2 = 1,5$  -> parte entera = 1.

# Implementación de un árbol binario mediante un vector de posiciones relativas

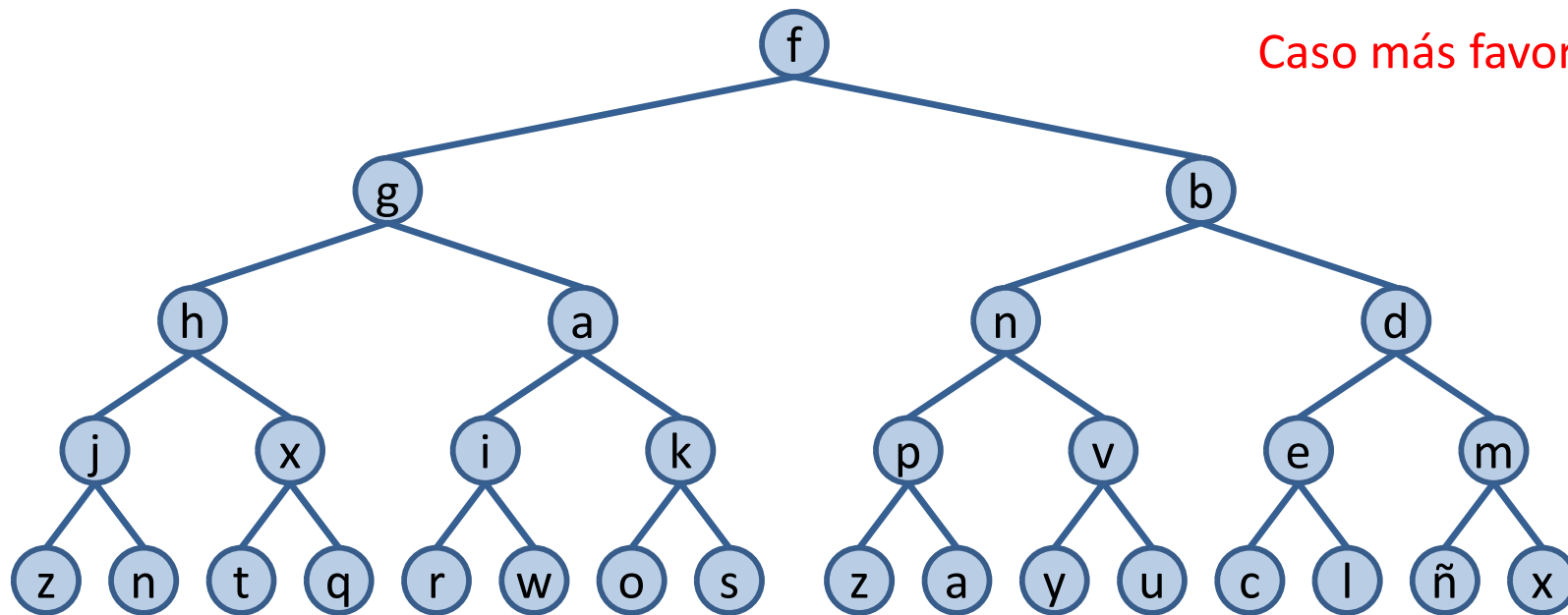
Sea un árbol de altura máxima  $h$ .  
La ausencia de un nodo en el nivel  $n \leq h$  provocará  $2^{h-n+1}-1$  posiciones libres en el vector.

Caso más desfavorable



# Implementación de un árbol binario mediante un vector de posiciones relativas

Caso más favorable



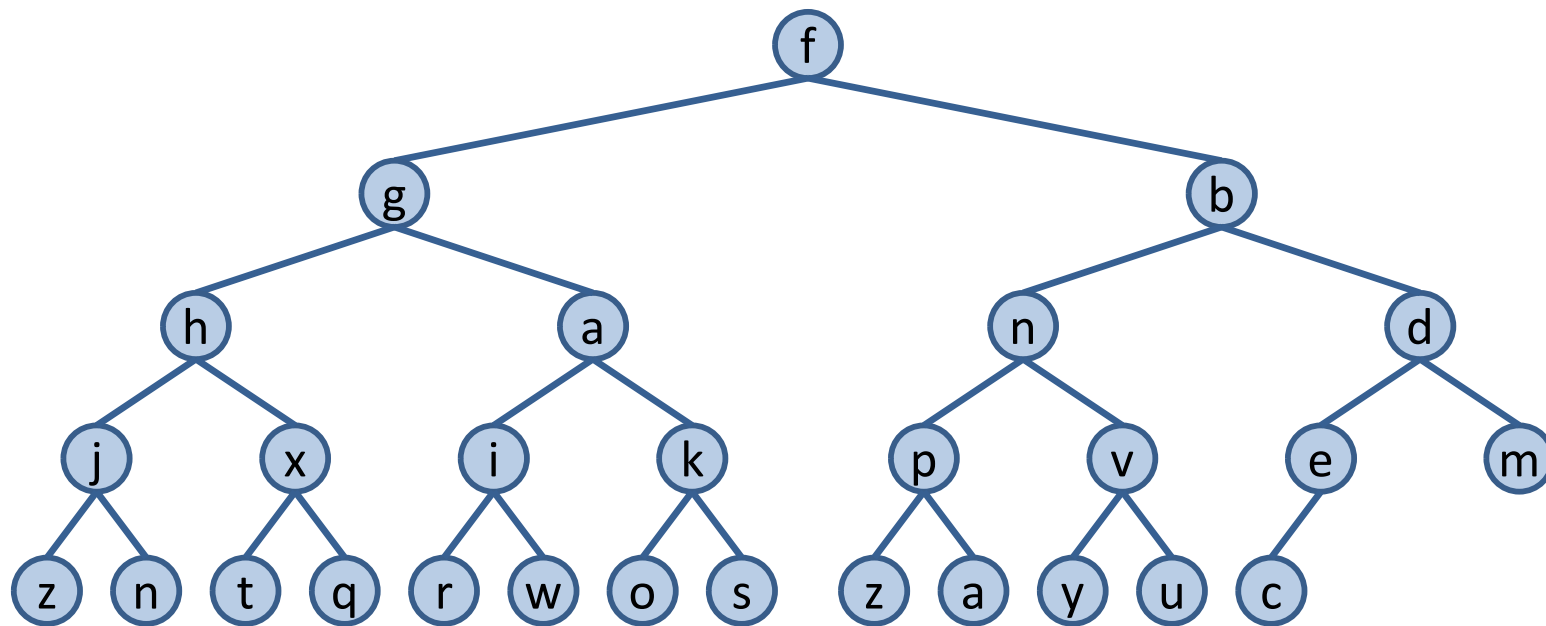
maxNodos-1

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	25	26	27	28	29	30
f	g	b	h	a	n	d	j	x	i	k	p	v	e	m	z	y	u	c	l	ñ	x

nodos



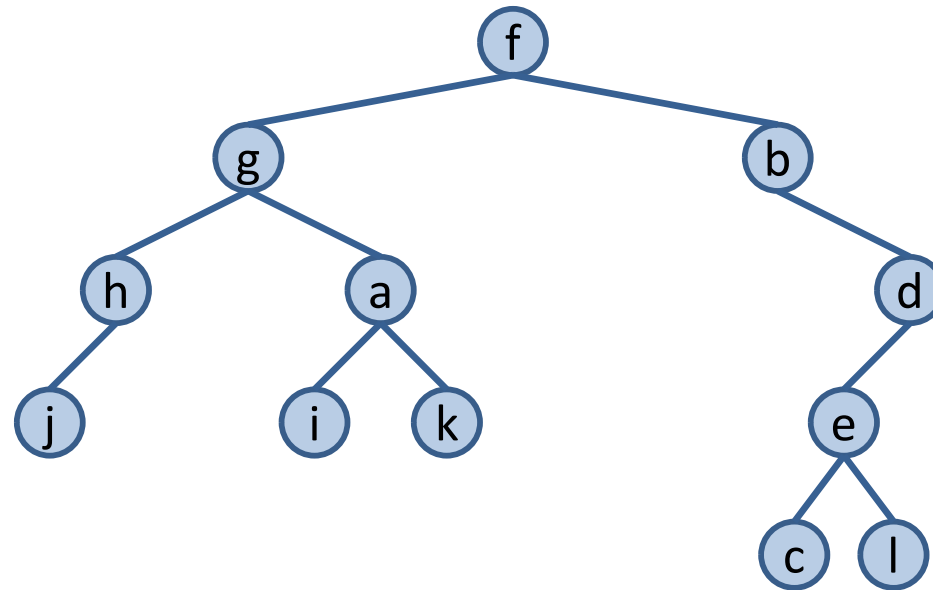
# Implementación de un árbol binario mediante un vector de posiciones relativas



																maxNodos-1							
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	25	26	27	28	29	30	
nodos	f	g	b	h	a	n	d	j	x	i	k	p	v	e	m	z		y	u	c			

La eficiencia espacial será mayor cuanto más lleno esté el árbol, es decir, cuantos menos nodos falten y, por tanto, más bajos sean los niveles en que falten.

# Implementación de un árbol binario mediante un vector de posiciones relativas



																maxNodos-1								
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15		25	26	27	28	29	30	
nodos	f	g	b	h	a	@	d	j	@	i	k	@	@	e	@	@			@	@	c	l	@	@

```

#ifndef ABIN_VEC1_H
#define ABIN_VEC1_H
#include <cassert>

template <typename T> class Abin {
public:
    typedef size_t nodo; // índice del vector,
                        // entre 0 y maxNodos-1
    static const nodo NODO_NULO;

    explicit Abin(size_t maxNodos, const T& e_nulo = T());
    void insertarRaiz(const T& e);
    void insertarHijoIzqdo(nodo n, const T& e);
    void insertarHijoDrcho(nodo n, const T& e);
    void eliminarHijoIzqdo(nodo n);
    void eliminarHijoDrcho(nodo n);
    void eliminarRaiz();
    bool arbolVacio() const;

```

Llamas al ctor del elemento nulo

```

    const T& elemento(nodo n) const; // Lec. en Abin const
    T& elemento(nodo n);           // Lec/Esc. en Abin no-const
    nodo raiz() const;
    nodo padre(nodo n) const;
    nodo hijoIzqdo(nodo n) const;
    nodo hijoDrcho(nodo n) const;
    Abin(const Abin<T>& a);        // Ctor. de copia
    Abin<T>& operator =(const Abin<T>& a); // Asig. de árboles
    ~Abin();                      // Destructor
private:
    T* nodos;                     // Vector de nodos
    size_t maxNodos;             // Tamaño del vector
    T ELTO_NULO;                 // Marca celdas vacías
};

/* Definición del nodo nulo */
template <typename T>
const typename Abin<T>::nodo Abin<T>::NODO_NULO(SIZE_MAX);

```

```

template <typename T>
Abin<T>::Abin(size_t maxNodos, const T& e_nulo) :
    nodos(new T[maxNodos]), Inicializas el árbol con todos los elementos nulos
    maxNodos(maxNodos),
    ELTO_NULO(e_nulo)
{
    // Marcar todas las celdas libres.
    for (nodo n = 0; n <= maxNodos-1; n++)
        nodos[n] = ELTO_NULO;
}

```

```

template <typename T>
inline void Abin<T>::insertarRaiz(const T& e)
{
    assert(nodos[0] == ELTO_NULO); // Árbol vacío.

    nodos[0] = e;
}

```

```

template <typename T> inline
void Abin<T>::insertarHijoIzqdo(nodo n, const T& e)
{
    Nodo válido -> nodo dentro del rango 0 < nodo < maxNodos +1 y no eres elemento nulo
    assert(n >= 0 && n <= maxNodos-1); // Nodo válido.
    assert(nodos[n] != ELTO_NULO); // Nodo del árbol.
    assert(2*n+1 < maxNodos); // Hijo izqdo. cabe en el árbol.
    assert(nodos[2*n+1] == ELTO_NULO); // n no tiene hijo izqdo.

    nodos[2*n+1] = e;
}

```

```

template <typename T> inline
void Abin<T>::insertarHijoDrcho(nodo n, const T& e)
{
    assert(n >= 0 && n < maxNodos-1); // Nodo válido
    assert(nodos[n] != ELTO_NULO); // Nodo del árbol
    assert(2*n+2 < maxNodos); // Hijo drcho. cabe en el árbol.
    assert(nodos[2*n+2] == ELTO_NULO); // n no tiene hijo drcho.

    nodos[2*n+2] = e;
}

```

```
template <typename T> inline
void Abin<T>::eliminarHijoIzqdo(nodo n)
{
```

Nodo izquierdo  $n \rightarrow 2n + 1$   
Hjo Izquierdo de  $n \rightarrow 4n + 3$   
Hijo derecho de  $n \rightarrow 4n + 4$

```
    assert(n >= 0 && n <= maxNodos-1);    // Nodo válido.
    assert(nodos[n] != ELTO_NULO);          // Nodo del árbol.
    assert(2*n+1 < maxNodos);               // Hijo izqdo. cabe en el árbol.
    assert(nodos[2*n+1] != ELTO_NULO);      // n tiene hijo izqdo.
    if (4*n+4 < maxNodos)                   // Caben los hijos del hijo izqdo. de n
        assert(nodos[4*n+3] == ELTO_NULO && // Hijo izqdo. de
                nodos[4*n+4] == ELTO_NULO); // n es hoja
    else if (4*n+3 < maxNodos)              // Sólo cabe h. izq. de h. izq. de n
        assert(nodos[4*n+3] == ELTO_NULO); // Hijo izq. de n es hoja

    nodos[2*n+1] = ELTO_NULO;
```

```
}
```

Nodo n valido y no sea elemento nulo.

El hijo izquierdo es nodo válido y no nulo.

Que sea hoja -> hijos estén fuera de rango, que sean nulos o ambos

Nota: Los hijos son el doble +1 ó el doble +2 del nodo n .

Si no cabe uno, no cabe el otro, por eso se comprueba uno y luego si cabe se comprueba el otro.

```
template <typename T> inline
void Abin<T>::eliminarHijoDrcho(nodo n)
{
```

Nodo derecho n ->  $2n + 2$   
Hjo Izquierdo de n ->  $4n + 5$   
Hijo derecho de n ->  $4n + 6$

```
    assert(n >= 0 && n <= maxNodos-1);    // Nodo válido.
    assert(nodos[n] != ELTO_NULO);          // Nodo del árbol.
    assert(2*n+2 < maxNodos);               // Hijo drcho. cabe en el árbol.
    assert(nodos[2*n+2] != ELTO_NULO);      // n tiene hijo drcho.
    if (4*n+6 < maxNodos)                   // Caben los hijos del hijo drcho. de n
        assert(nodos[4*n+5] == ELTO_NULO && // Hijo drcho. de
                nodos[4*n+6] == ELTO_NULO); // n es hoja
    else if (4*n+5 < maxNodos) //Sólo cabe h. izq. de h. drch de n
        assert(nodos[4*n+5] == ELTO_NULO); //Hijo drch de n es hoja
```

```
    nodos[2*n+2] = ELTO_NULO;
```

```
}
template <typename T>
inline void Abin<T>::eliminarRaiz()
{
```

```
    assert(nodos[0] != ELTO_NULO); // Árbol no vacío
    assert(nodos[1] == ELTO_NULO &&
           nodos[2] == ELTO_NULO); // La raíz es hoja
    nodos[0] = ELTO_NULO;
```

```
}
```



```

template <typename T>
inline bool Abin<T>::arbolVacio() const
{
    return (nodos[0] == ELTO_NULO);
}

```

```

template <typename T>
inline const T& Abin<T>::elemento(nodo n) const
{
    assert(n >= 0 && n <= maxNodos-1); // Nodo válido.
    assert(nodos[n] != ELTO_NULO);      // Nodo del árbol.
    return nodos[n];
}

```

```

template <typename T>
inline T& Abin<T>::elemento(nodo n)
{
    assert(n >= 0 && n <= maxNodos-1); // Nodo válido.
    assert(nodos[n] != ELTO_NULO);      // Nodo del árbol.
    return nodos[n];
}

```

```

template <typename T>
inline typename Abin<T>::nodo Abin<T>::raiz() const
{
    return (nodos[0] == ELTO_NULO) ? NODO_NULO : 0;
}

template <typename T> inline
typename Abin<T>::nodo Abin<T>::padre(nodo n) const
{
    assert(n >= 0 && n <= maxNodos-1); // Nodo válido.
    assert(nodos[n] != ELTO_NULO);      // Nodo del árbol.

    return (n == 0) ? NODO_NULO : (n-1)/2;
    Si queremos devolver el padre del raiz, se devuelve NODO NULO
}

```

```

template <typename T> inline
typename Abin<T>::nodo Abin<T>::hijoIzqdo(nodo n) const
{
    assert(n >= 0 && n <= maxNodos-1);    // Nodo válido.
    assert(nodos[n] != ELTO_NULO);         // Nodo del árbol.

    return (2*n+1 >= maxNodos || nodos[2*n+1] == ELTO_NULO) ?
        NODO_NULO : 2*n+1;
}

```

```

template <typename T> inline
typename Abin<T>::nodo Abin<T>::hijoDrcho(nodo n) const
{
    assert(n >= 0 && n <= maxNodos-1);    // Nodo válido.
    assert(nodos[n] != ELTO_NULO);         // Nodo del árbol.

    return (2*n+2 >= maxNodos || nodos[2*n+2] == ELTO_NULO) ?
        NODO_NULO : 2*n+2;
}

```

```

template <typename T>
Abin<T>::Abin(const Abin<T>& A) :
    nodos(new T[A.maxNodos]),
    maxNodos(A.maxNodos),
    ELTO_NULO(A.ELTO_NULO)
{
    // Copiar el vector
    for (nodo n = 0; n <= maxNodos-1; n++)
        nodos[n] = A.nodos[n];
}

```

```

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

```

```

template <typename T>
Abin<T>& Abin<T>::operator =(const Abin<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];
        }
        ELTO_NULO = A.ELTO_NULO;
        // Copiar el vector.
        for (nodo n = 0; n <= maxNodos-1; n++)
            nodos[n] = A.nodos[n];
    }
    return *this;
}

#endif // ABIN_VEC1_H

```