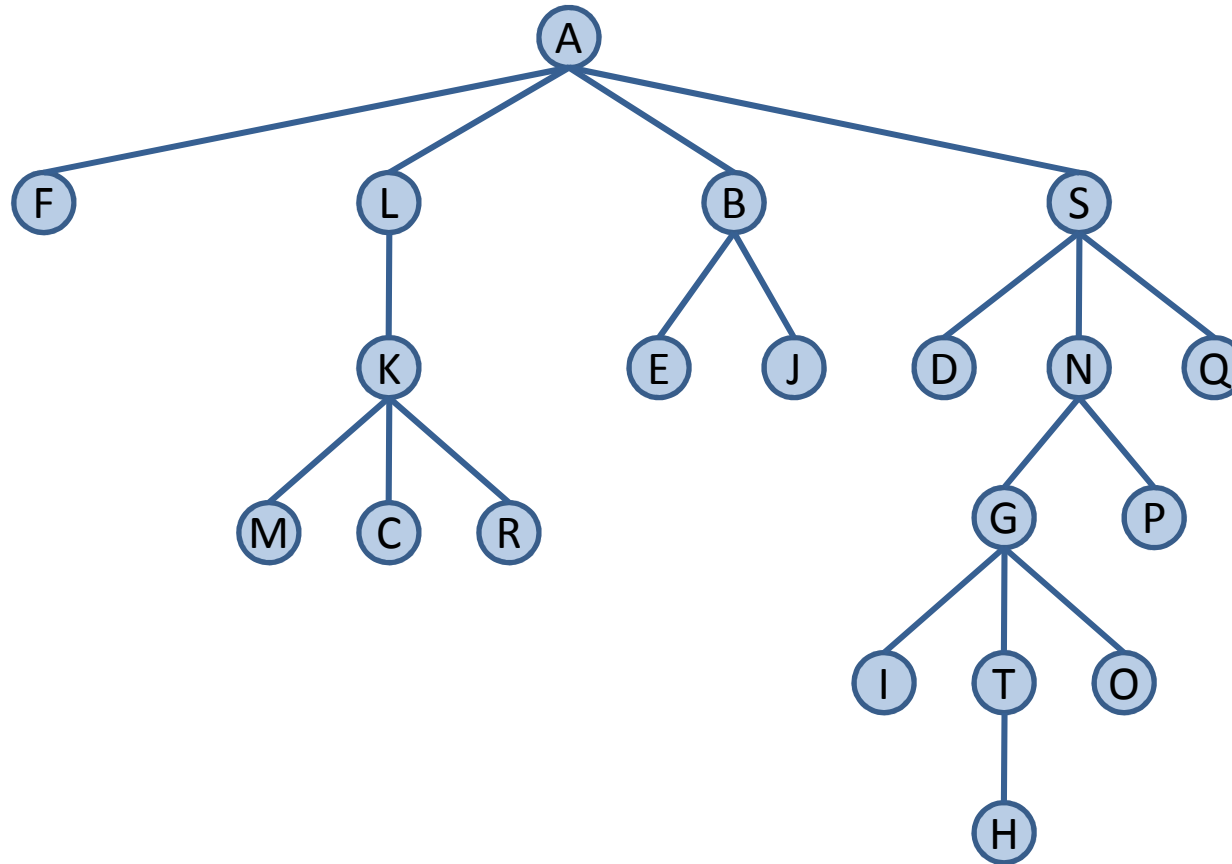


Estructuras de Datos no Lineales

Tema 1 Árboles

Concepto:

Un **árbol** es una colección de elementos de un tipo determinado, cada uno de los cuales se almacena en un **nodo**. Existe una **relación de paternidad** entre los nodos que determina una **estructura jerárquica** sobre los mismos.

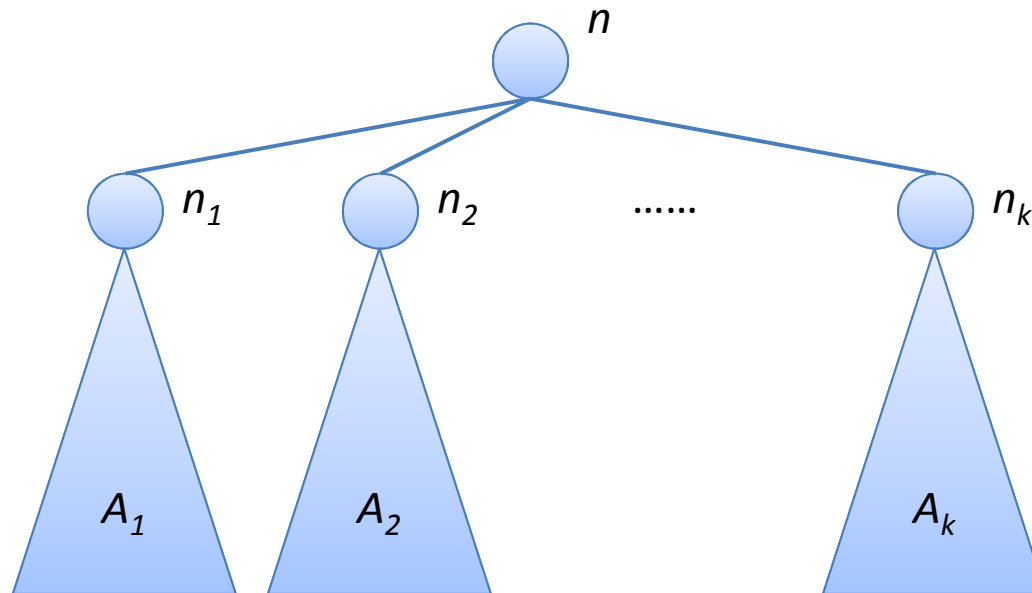


Una definición recursiva más formal es la siguiente:

- Un solo nodo es, por sí mismo, un árbol. Este único nodo se llama nodo **raíz** del árbol.



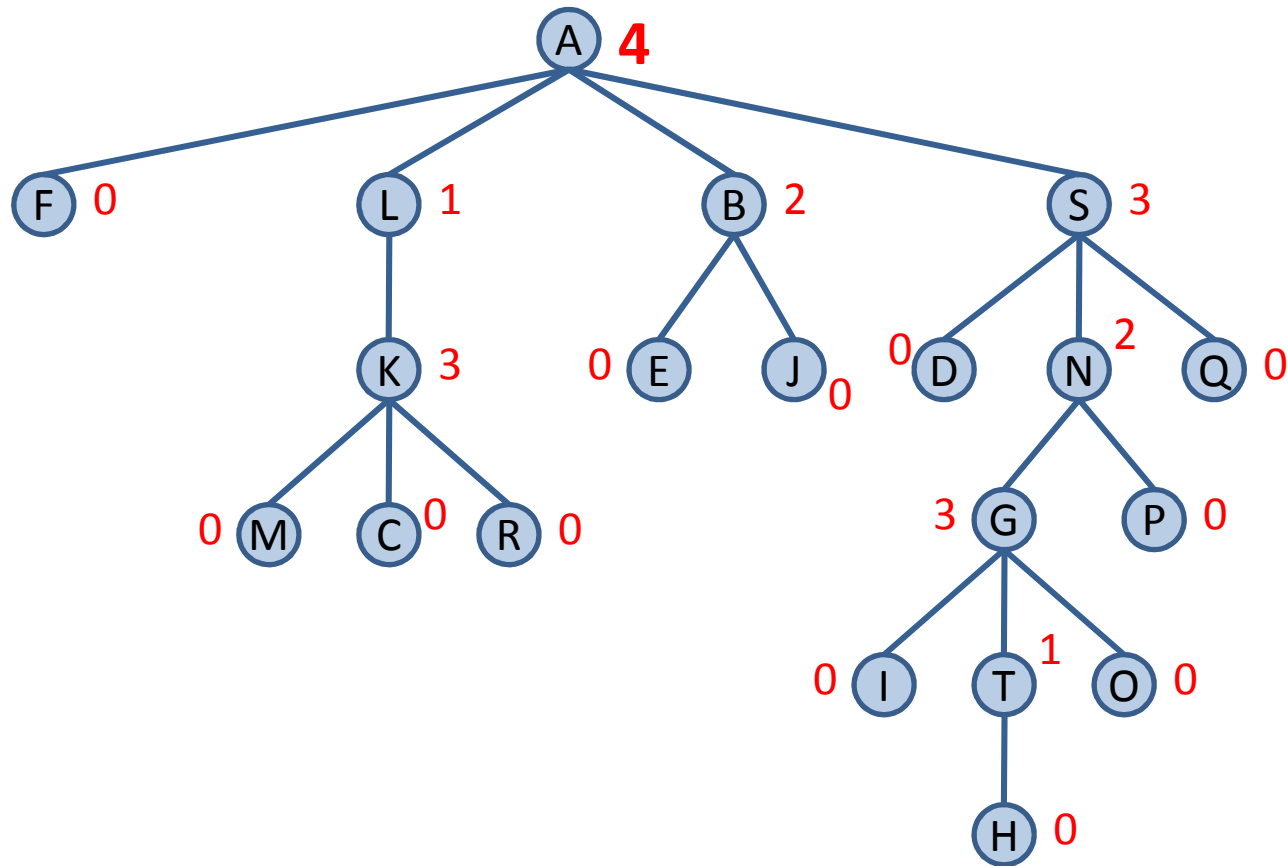
- Si n es un nodo y A_1, A_2, \dots, A_k son árboles con raíces n_1, n_2, \dots, n_k , respectivamente, y se define una relación padre-hijo entre n y n_1, n_2, \dots, n_k , entonces la estructura resultante es un árbol. En este árbol, n es la **raíz**, A_1, A_2, \dots, A_k son **subárboles** de la raíz, n es el **padre** de los nodos n_1, n_2, \dots, n_k y éstos, por tanto, son los **hijos** de n y **hermanos** entre sí.



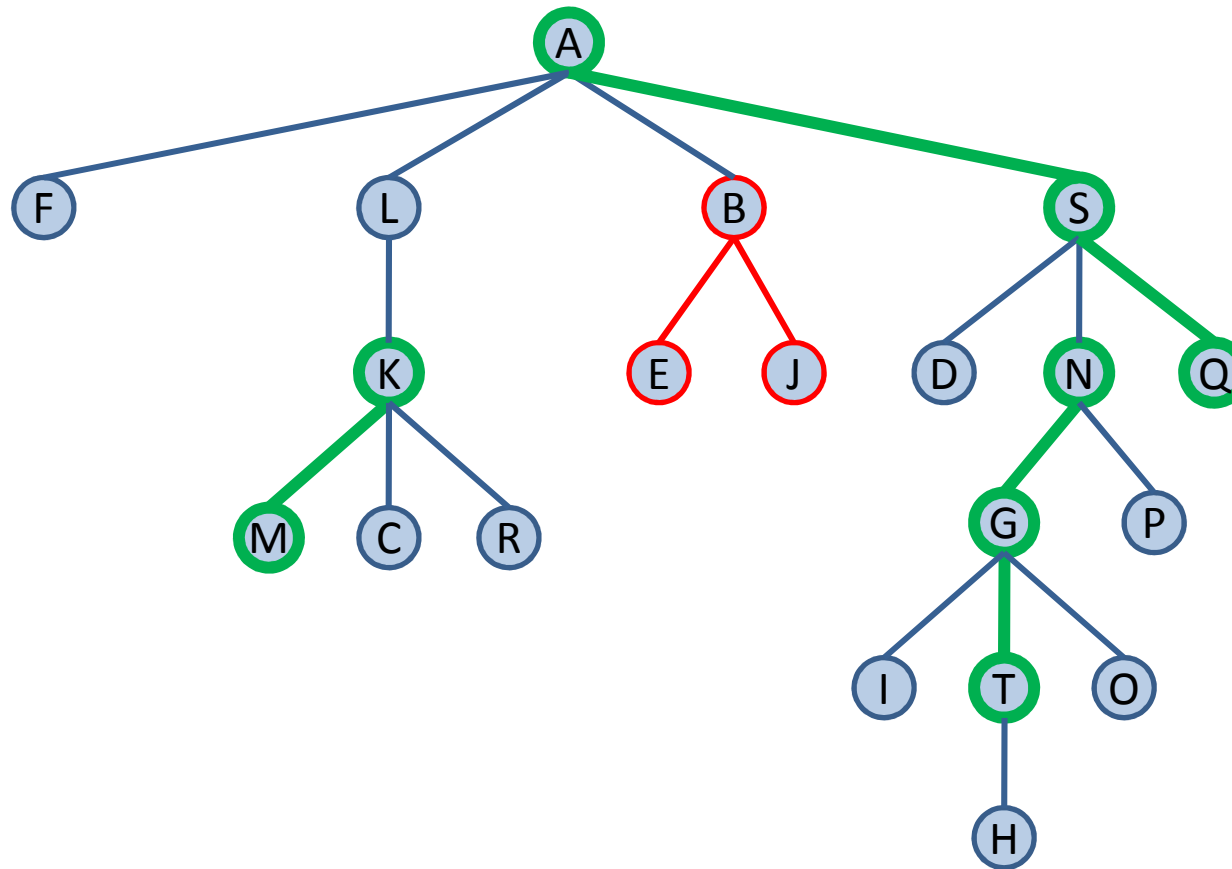
Además, llamaremos **árbol nulo** o **árbol vacío** a aquél que no tiene ningún nodo.

Definiciones

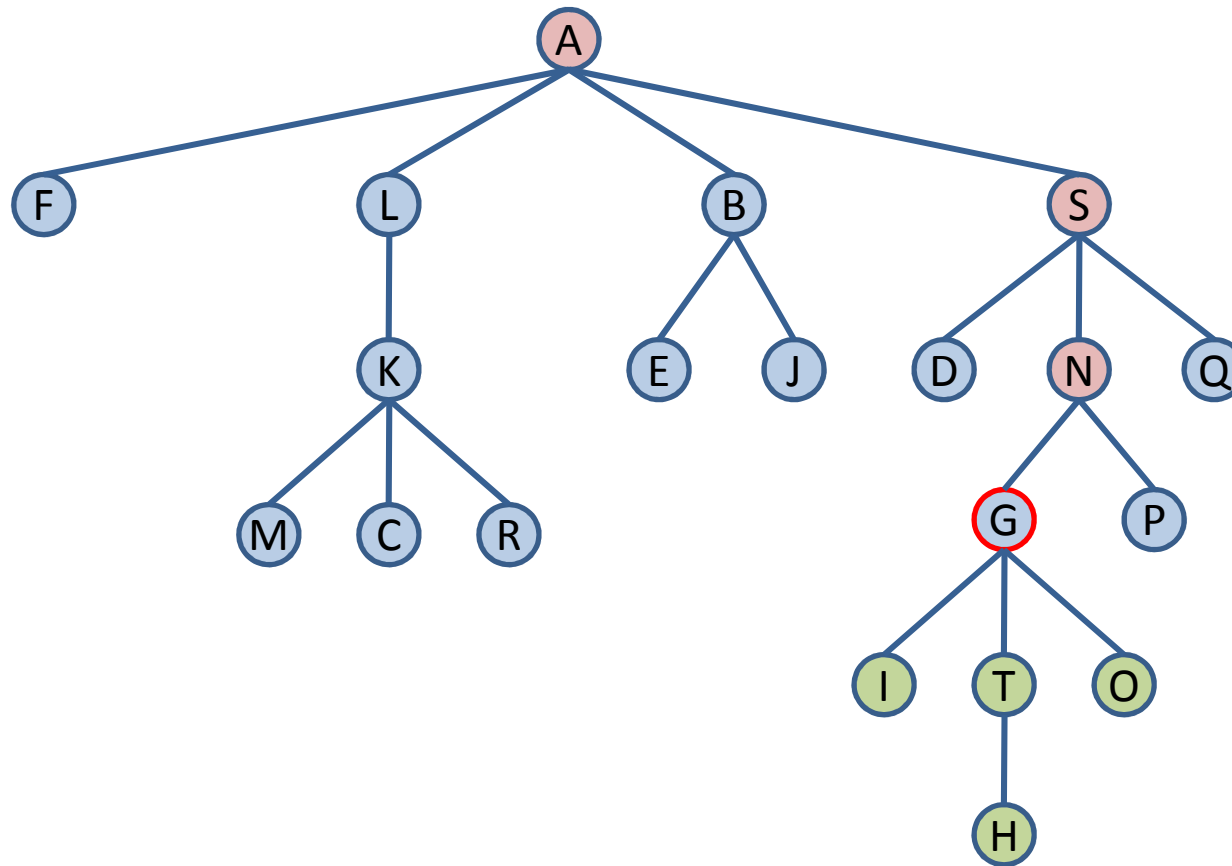
Grado: Número de hijos de un nodo. El grado de un árbol es el máximo de los grados de sus nodos.



Camino: Una sucesión de nodos de un árbol n_1, n_2, \dots, n_k , tal que n_i es el padre de n_{i+1} para $1 \leq i \leq k$. La *longitud* de un camino es el número de nodos menos 1. Por tanto, existe un camino de longitud 0 de cualquier nodo a sí mismo.



Ancestros y descendientes: Si existe un camino de un nodo a a otro b , entonces a es un *antecesor* o *ancestro* de b y b es un *descendiente* de a . Un ancestro o descendiente de un nodo distinto de sí mismo se llama *ancestro propio* o *descendiente propio*, respectivamente.

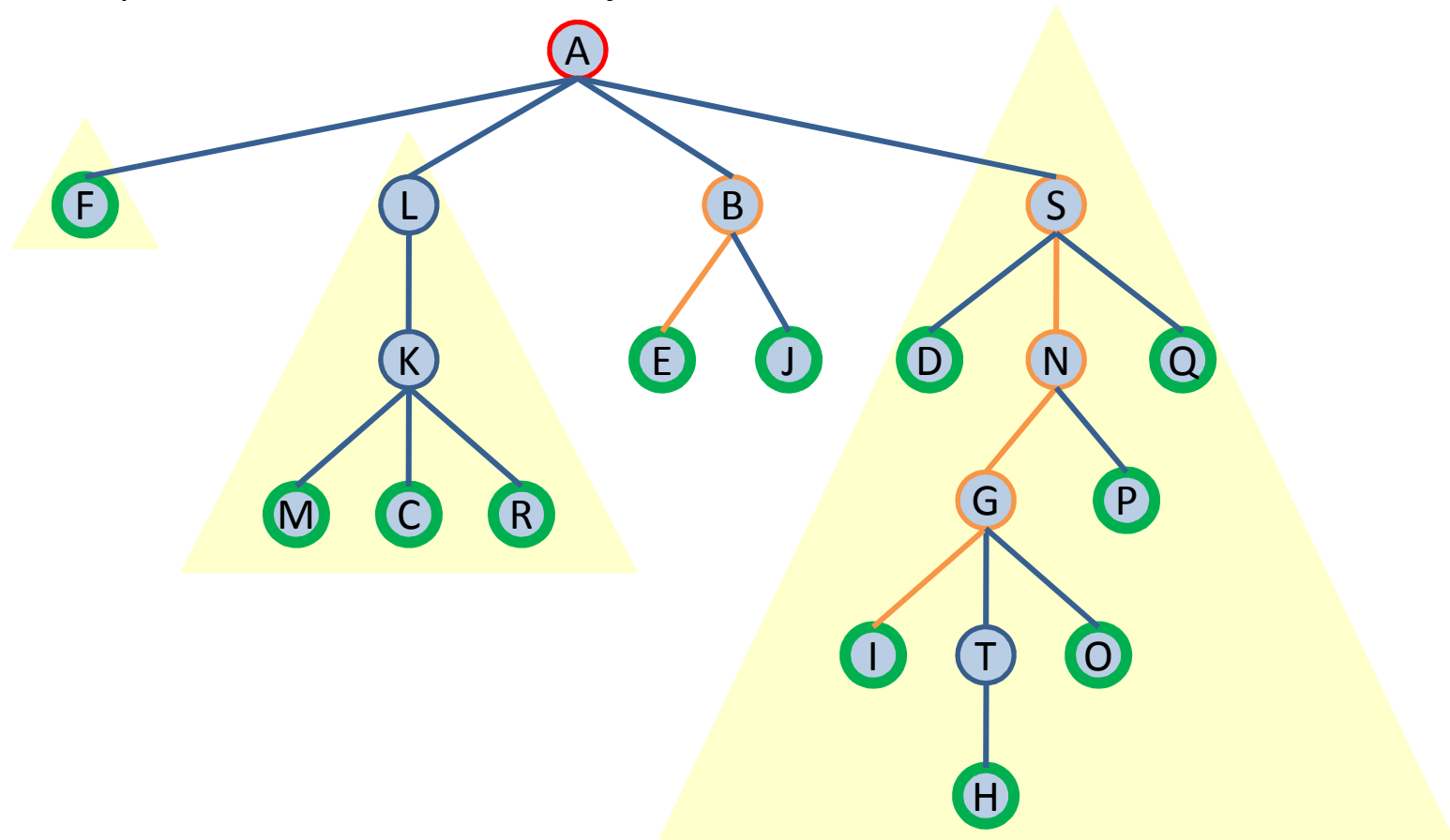


Raíz: Único nodo de un árbol que no tiene antecesores propios.

Hoja: Nodo que no tiene descendientes propios.

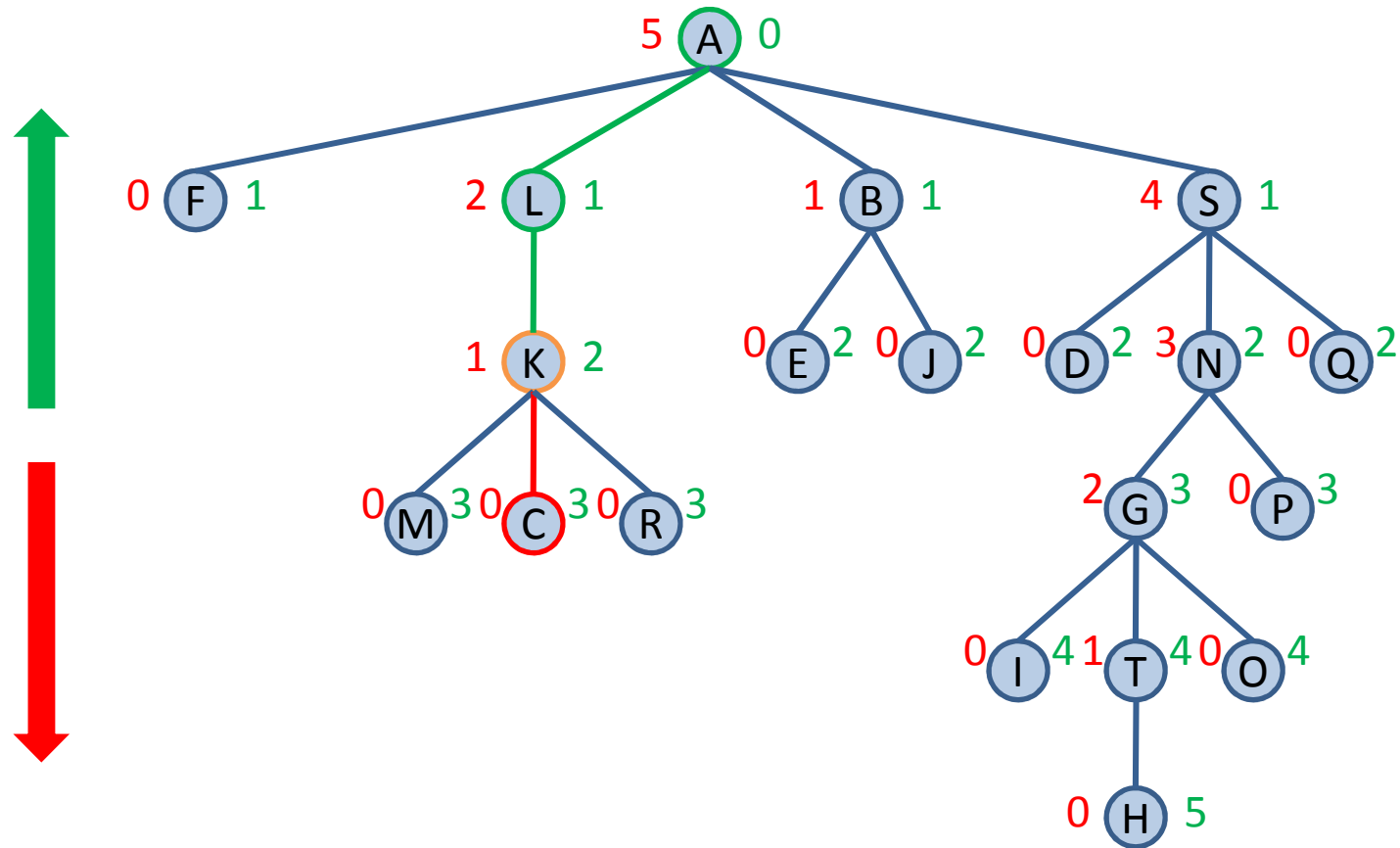
Subárbol: Conjunto de nodos formado por un nodo y todos sus descendientes.

Rama: Camino que termina en un nodo hoja.

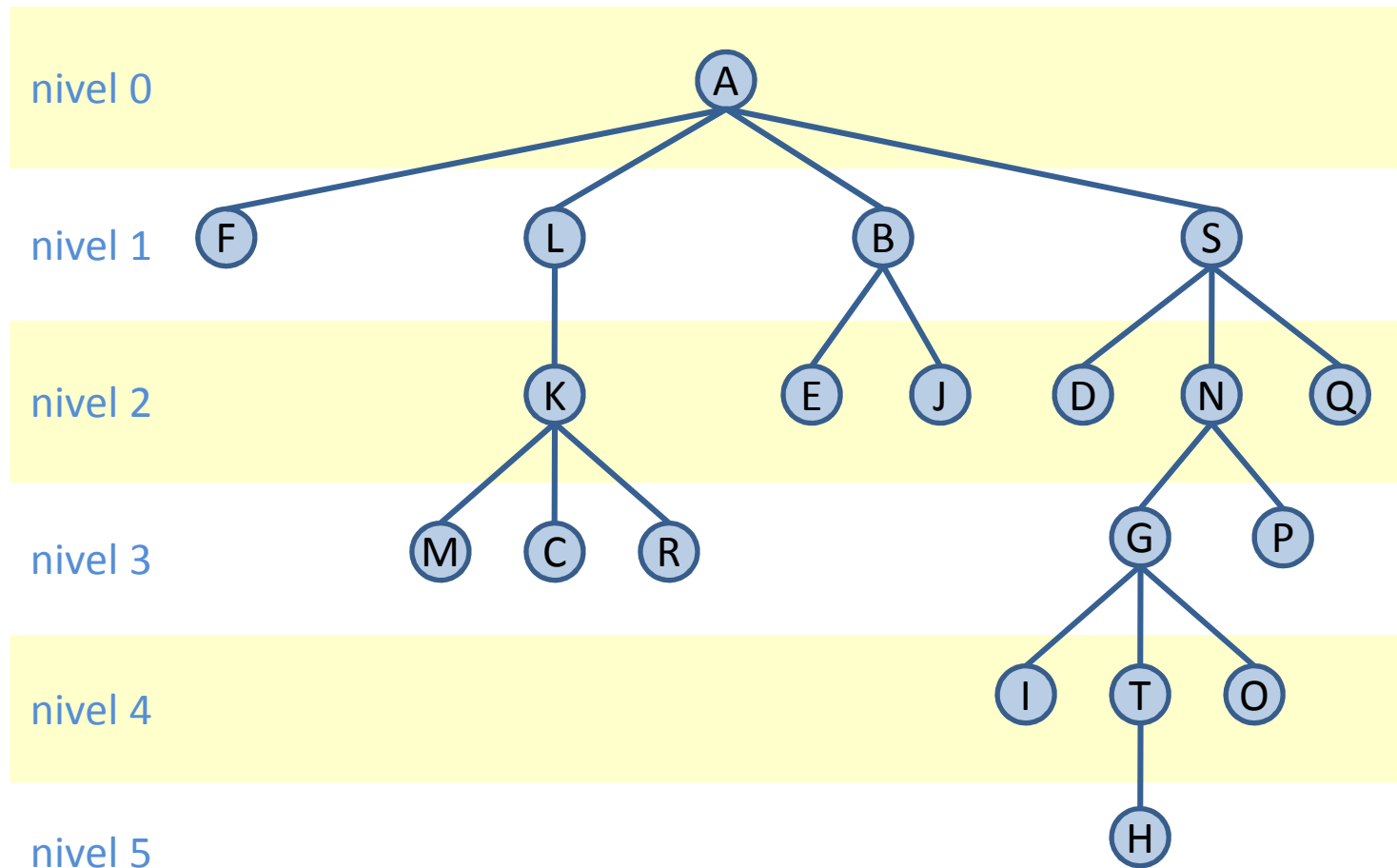


Altura: La altura de un nodo es la longitud de la rama más larga que parte de dicho nodo. La altura de un árbol es la altura del nodo raíz.

Profundidad: La profundidad de un nodo es la longitud del único camino desde la raíz a ese nodo.



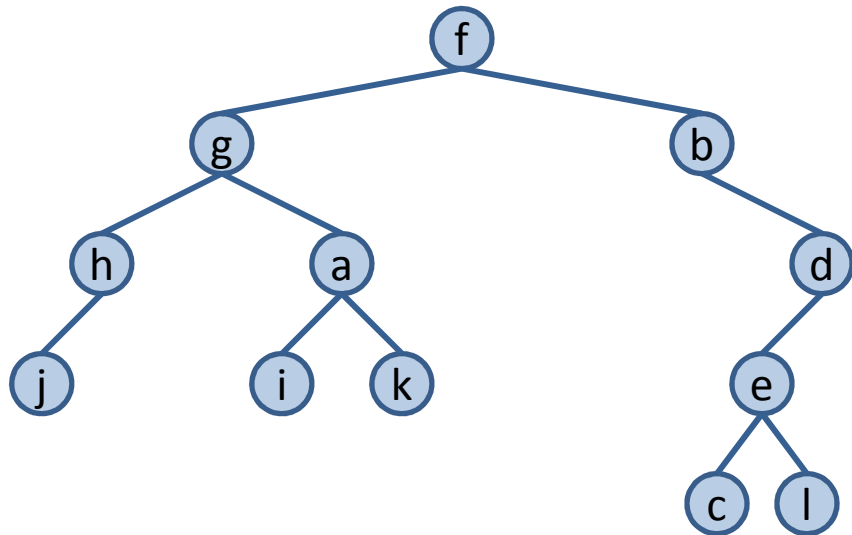
Nivel: El nivel de un nodo coincide con su **profundidad**. Los nodos de un árbol de altura h se clasifican en $h + 1$ niveles numerados de 0 a h , de tal forma que el nivel i lo forman todos los nodos de profundidad i .



TAD Árbol binario

Definición:

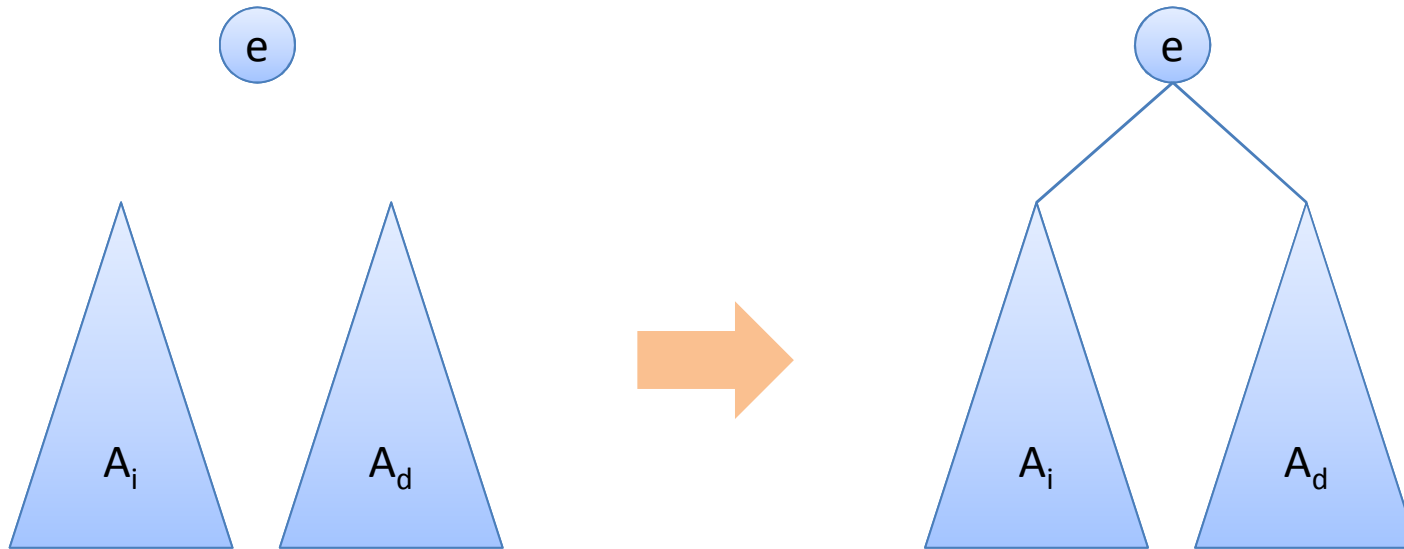
Un árbol binario se define como un árbol cuyos nodos son, a lo sumo, de grado 2, es decir, tienen 0, 1 ó 2 hijos. Éstos se llaman *hijo izquierdo* e *hijo derecho*.



Operaciones:

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

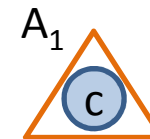
Construcción de un árbol binario (I)



`Abin (); // constructor predeterminado`

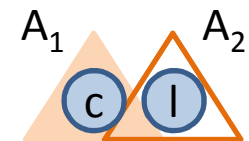
`Abin (const T& e, const Abin& Ai, const Abin& Ad); // constructor`

Construcción de un árbol binario (I)



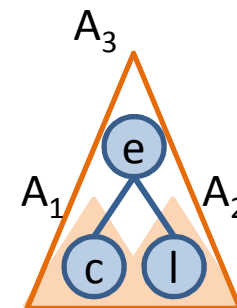
$\text{Abin } A_1('c', \text{Abin}(), \text{Abin}());$

Construcción de un árbol binario (I)



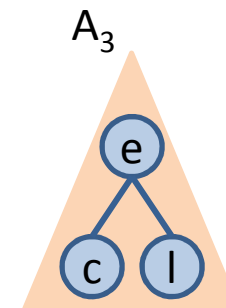
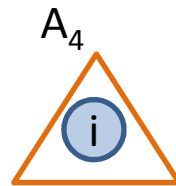
`Abin A2('l', Abin(), Abin());`

Construcción de un árbol binario (I)



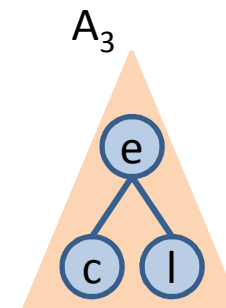
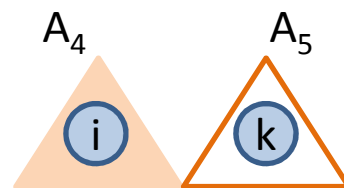
$\text{Abin } A_3('e', A_1, A_2);$

Construcción de un árbol binario (I)



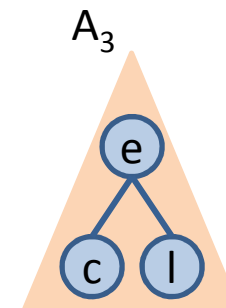
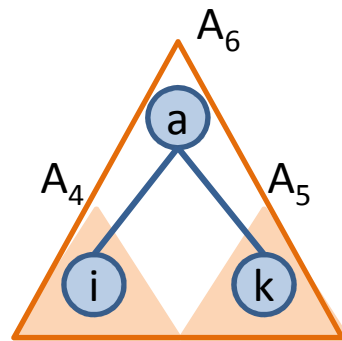
`Abin A_4 ('i', Abin(), Abin());`

Construcción de un árbol binario (I)



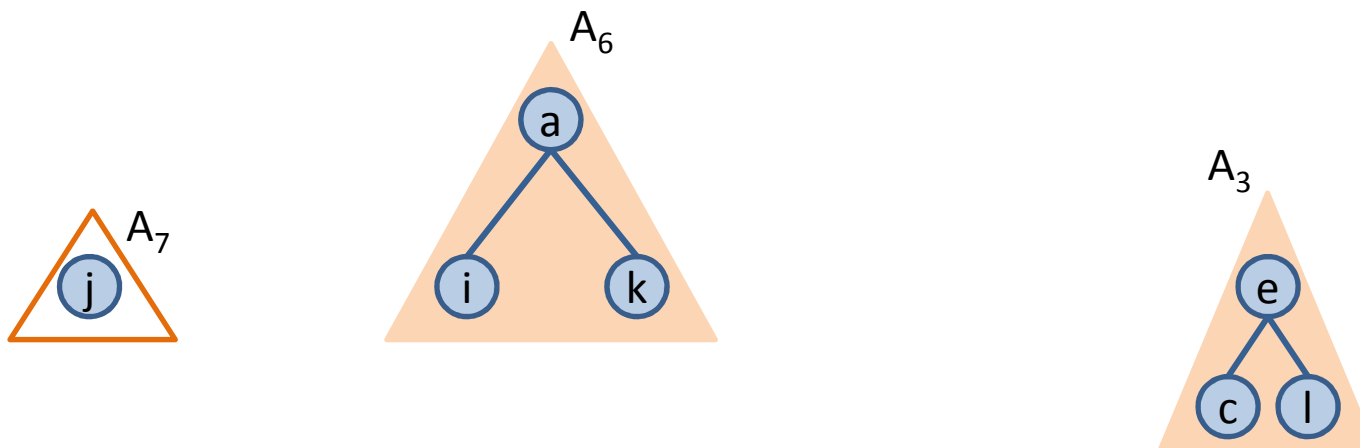
Abin A_5 ('k', Abin(), Abin());

Construcción de un árbol binario (I)



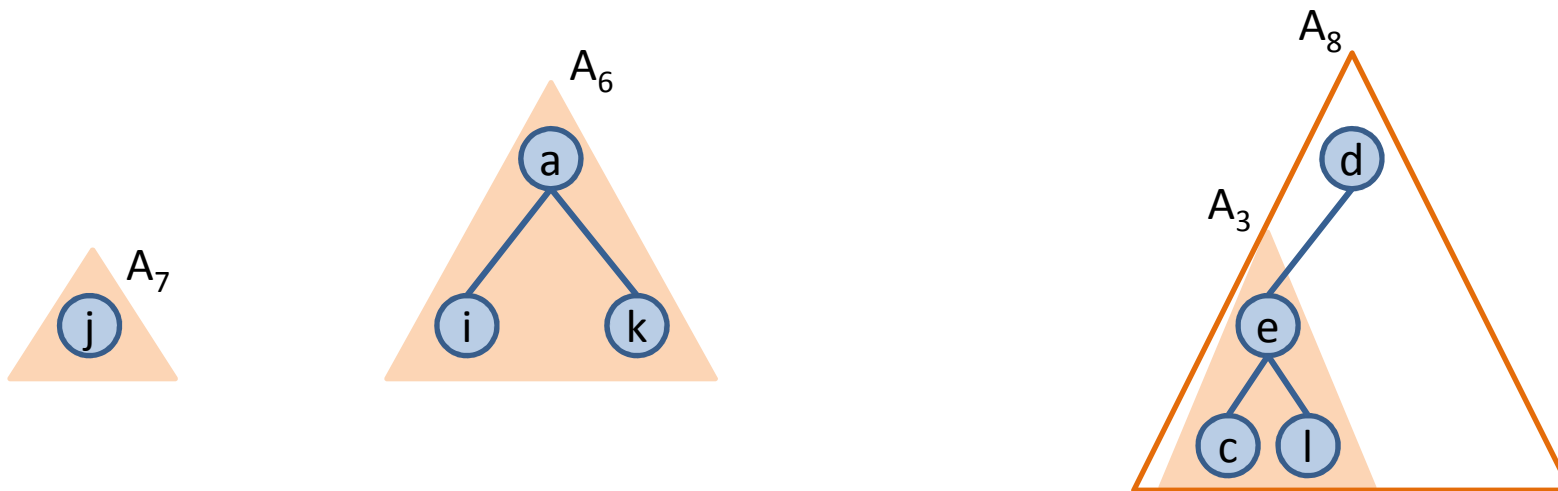
$\text{Abin } A_6('a', A_4, A_5);$

Construcción de un árbol binario (I)



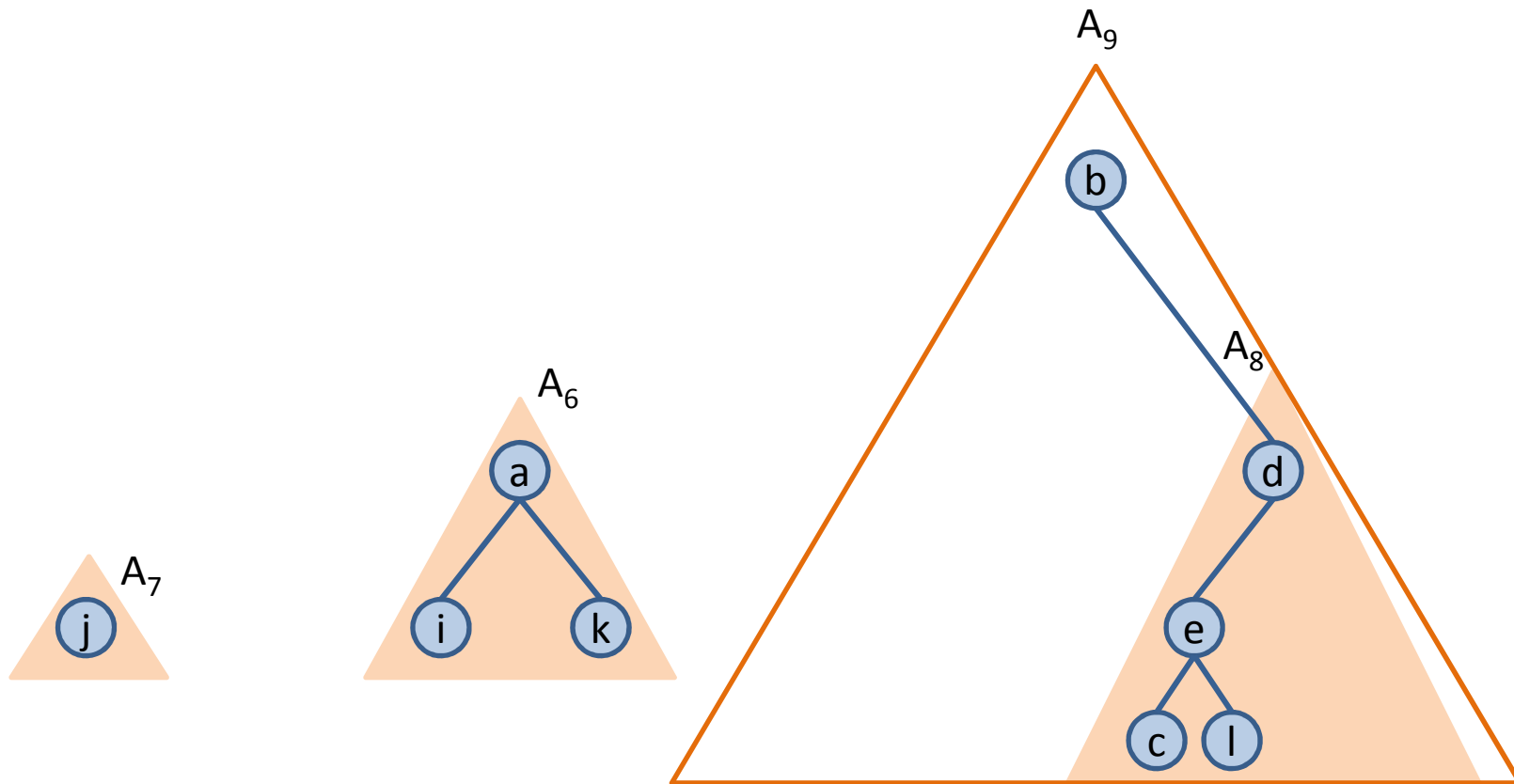
`Abin A7('j', Abin(), Abin());`

Construcción de un árbol binario (I)



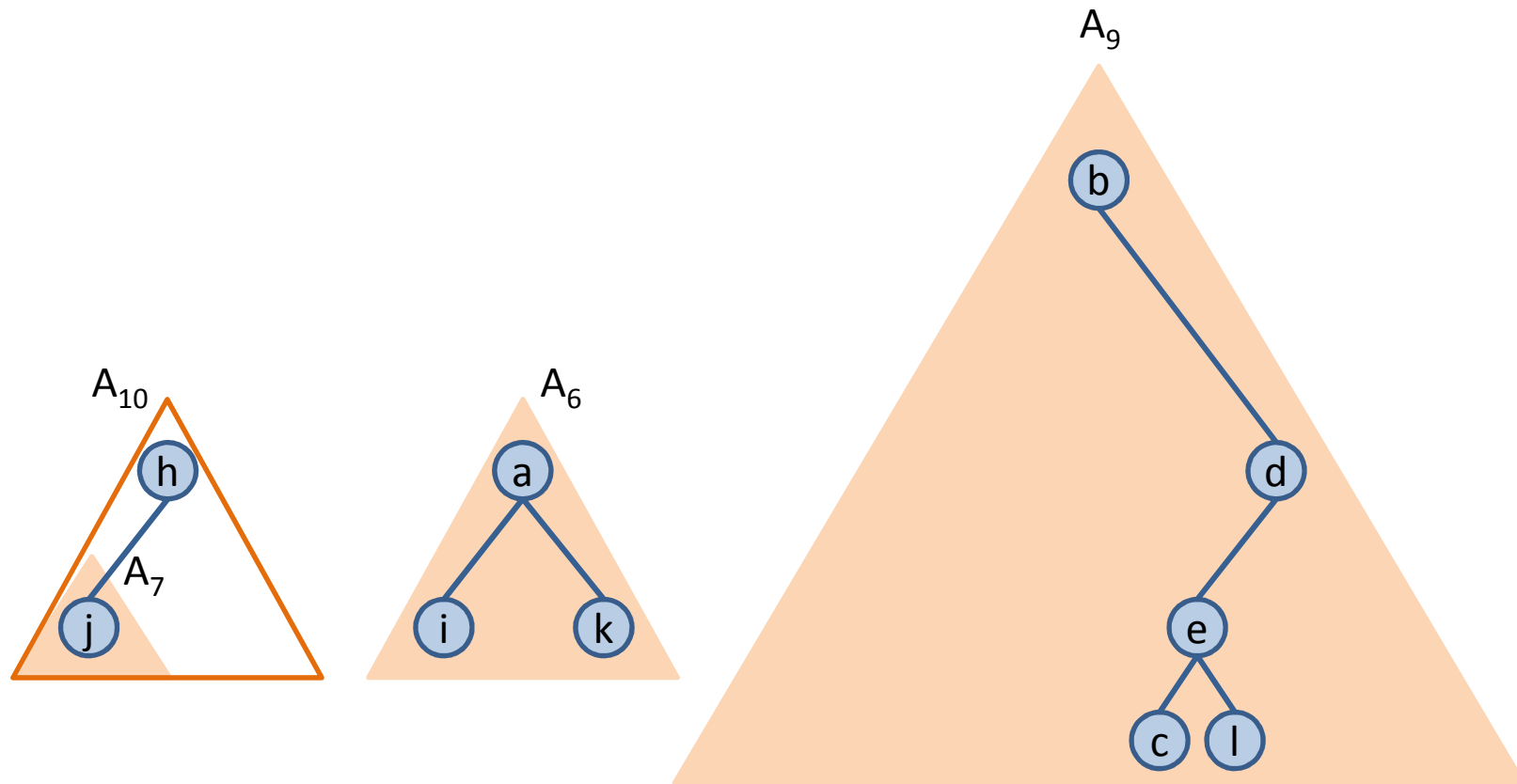
`Abin A8('d', A3, Abin());`

Construcción de un árbol binario (I)

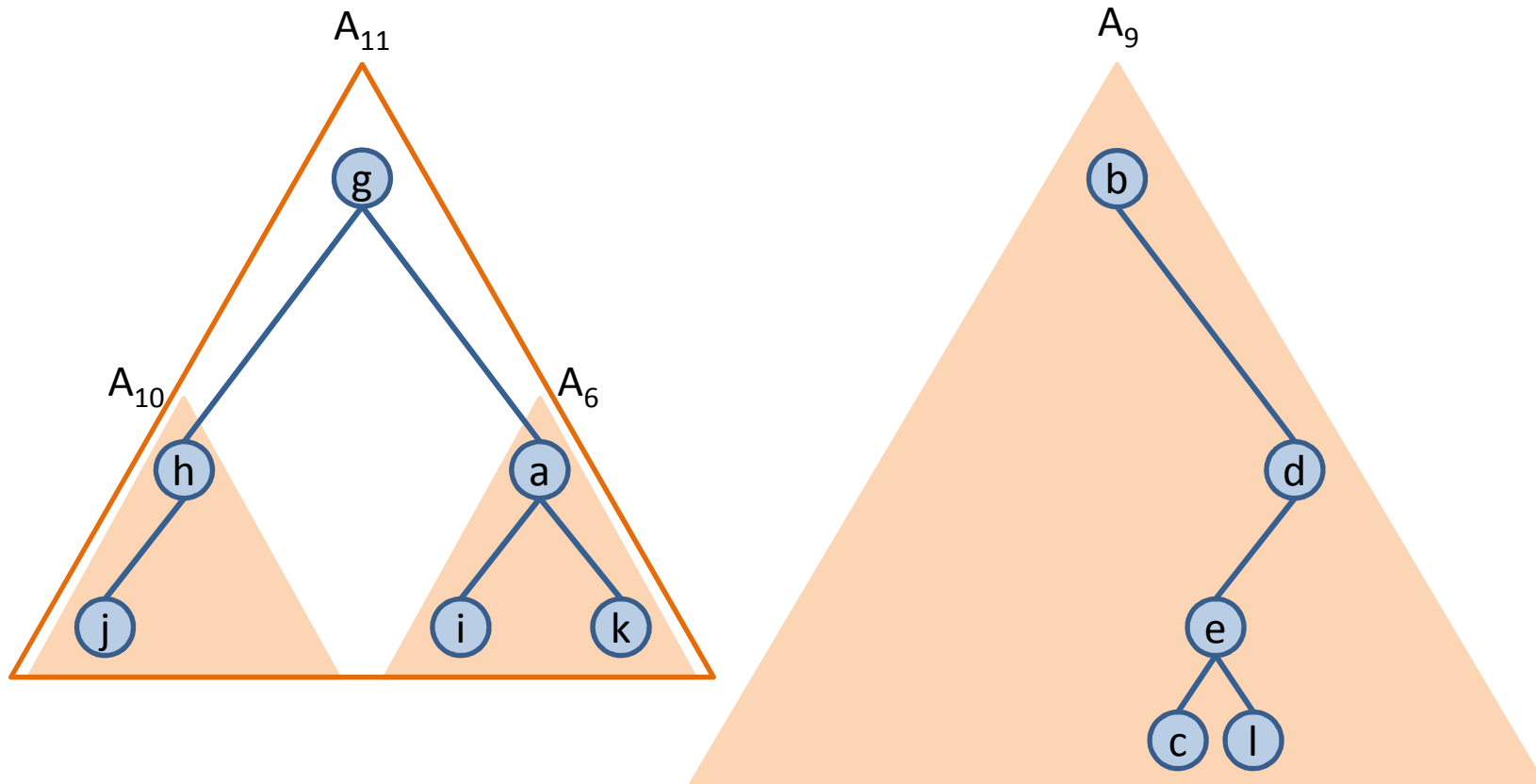


$\text{Abin } A_9('b', \text{Abin}(), A_8);$

Construcción de un árbol binario (I)

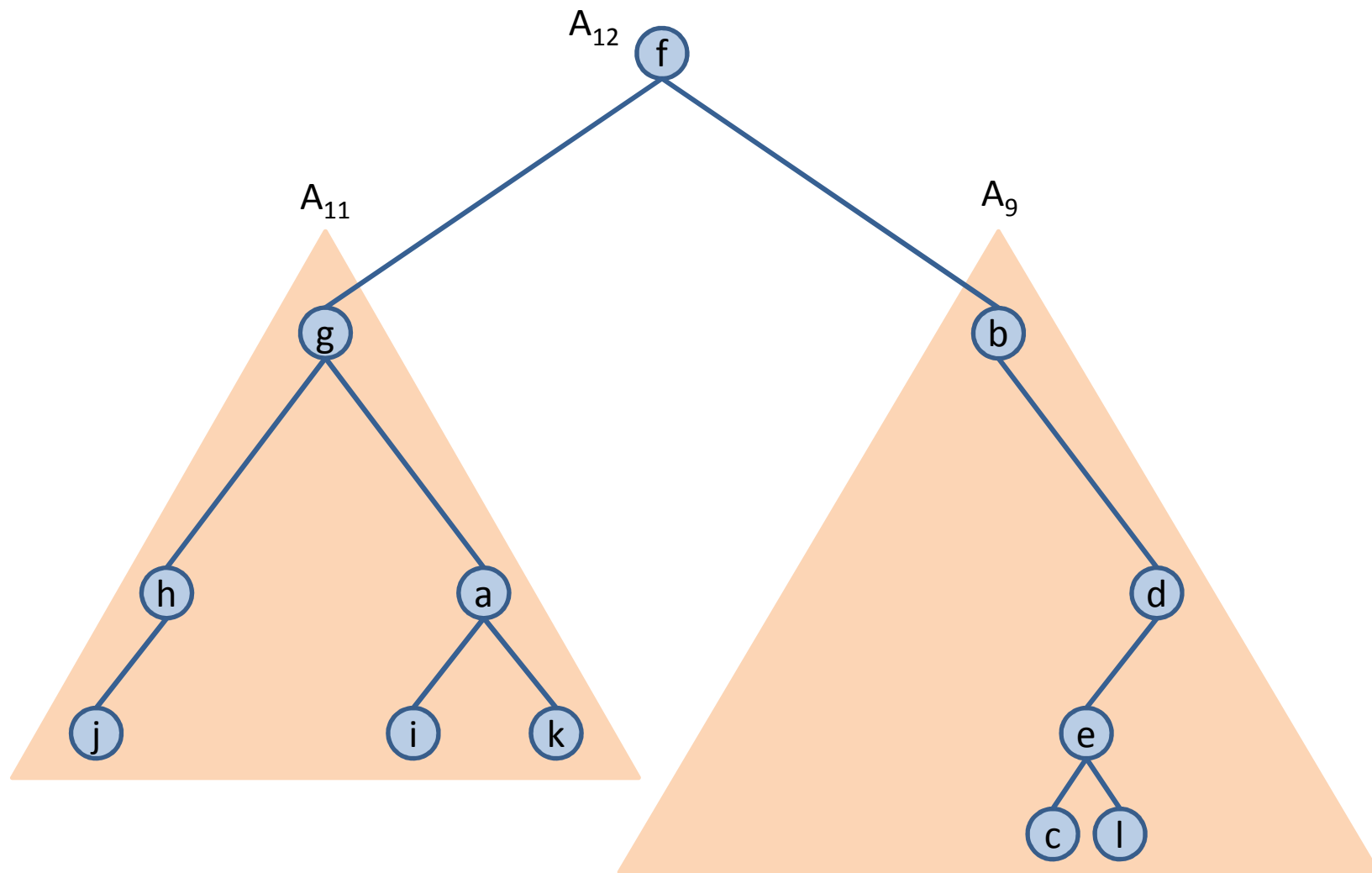


Construcción de un árbol binario (I)



$\text{Abin } A_{11}('g', A_{10}, A_6);$

Construcción de un árbol binario (I)



$\text{Abin } A_{12}('f', A_{11}, A_9);$

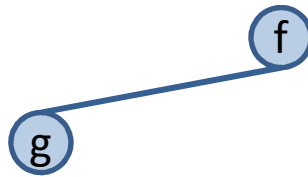
Construcción de un árbol binario (II)

Creación del árbol binario como un contenedor vacío.

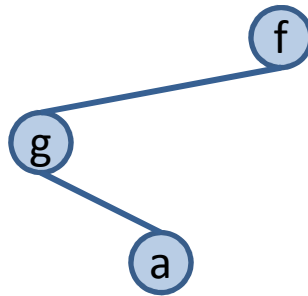
Construcción de un árbol binario (II)



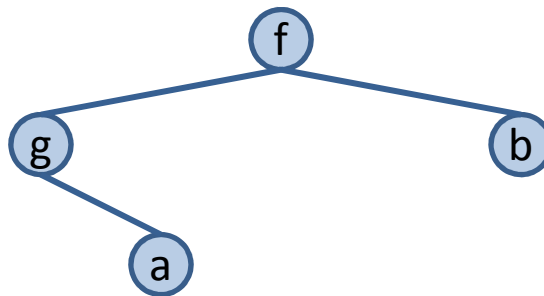
Construcción de un árbol binario (II)



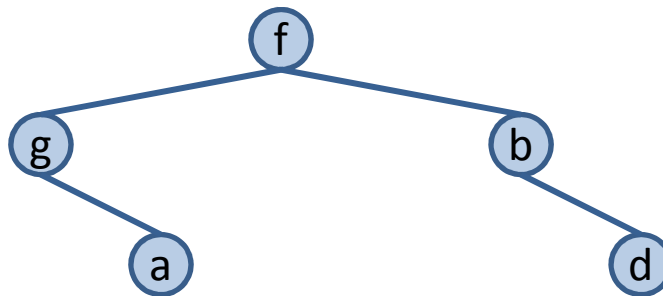
Construcción de un árbol binario (II)



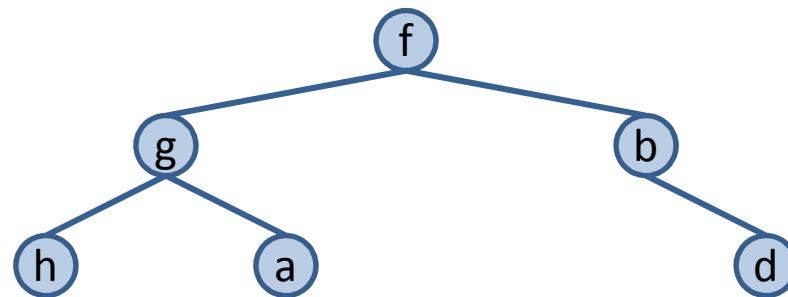
Construcción de un árbol binario (II)



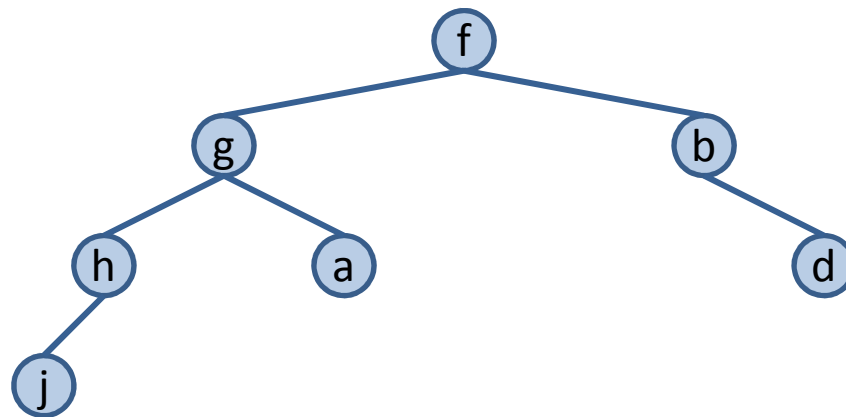
Construcción de un árbol binario (II)



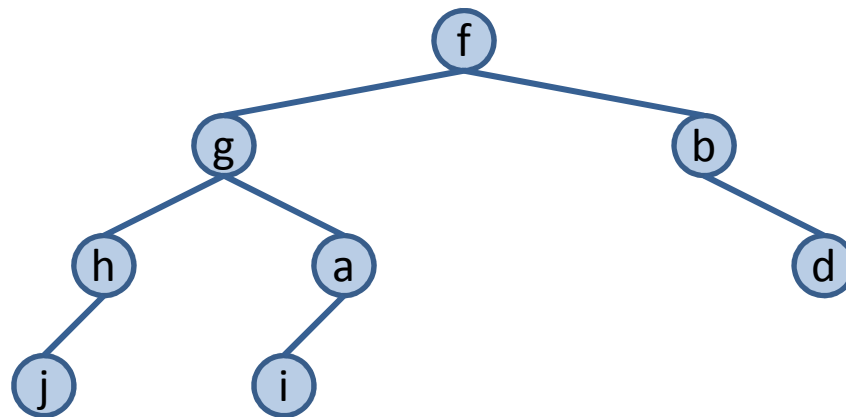
Construcción de un árbol binario (II)



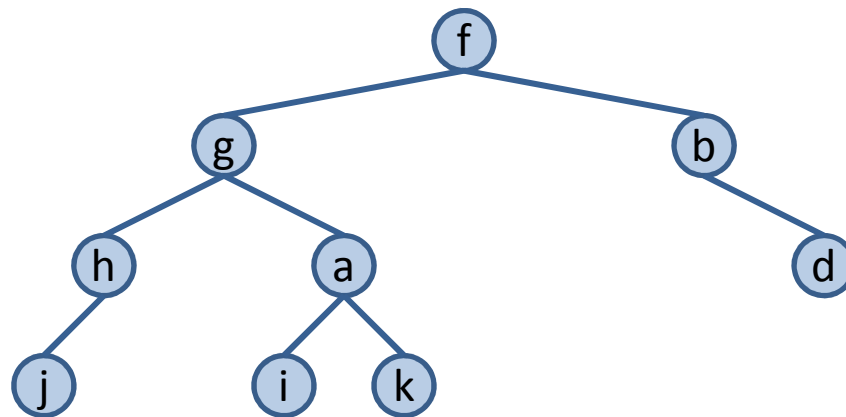
Construcción de un árbol binario (II)



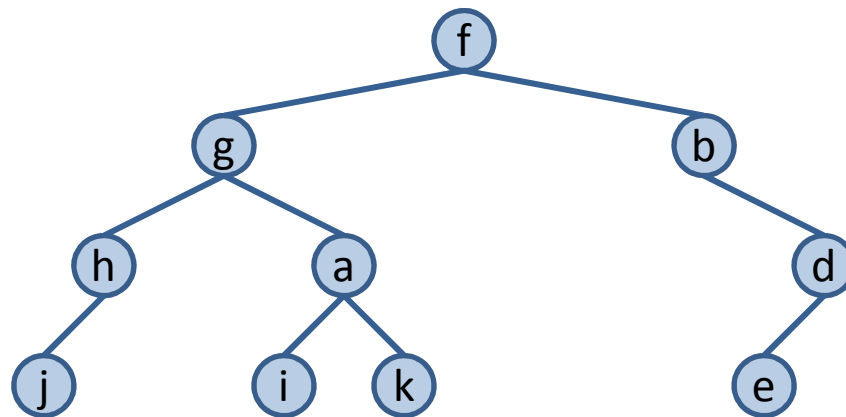
Construcción de un árbol binario (II)



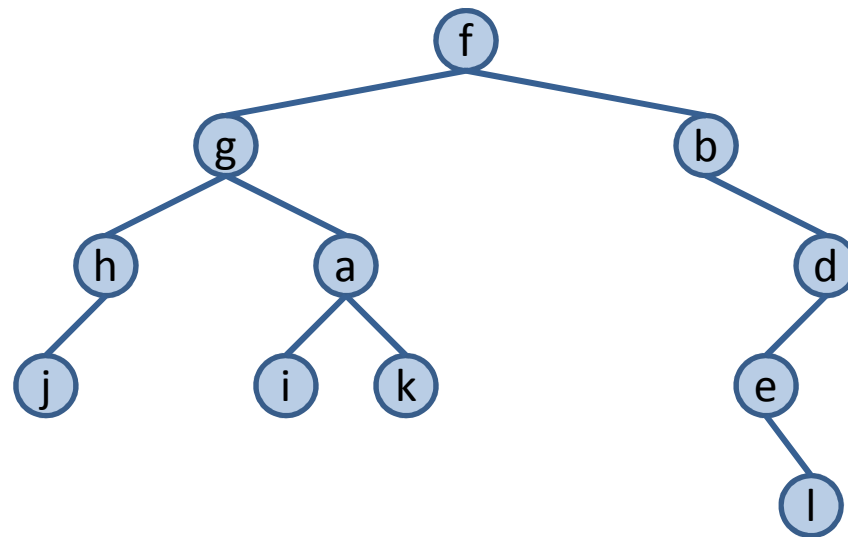
Construcción de un árbol binario (II)



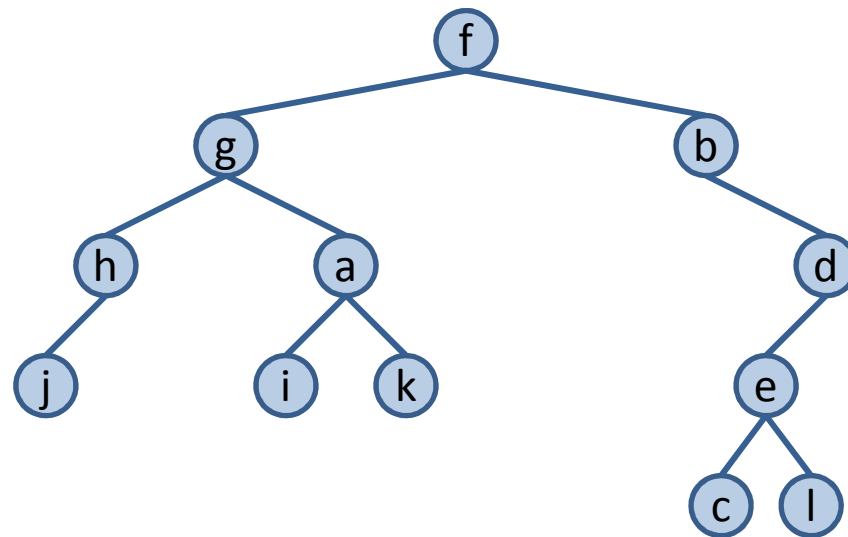
Construcción de un árbol binario (II)



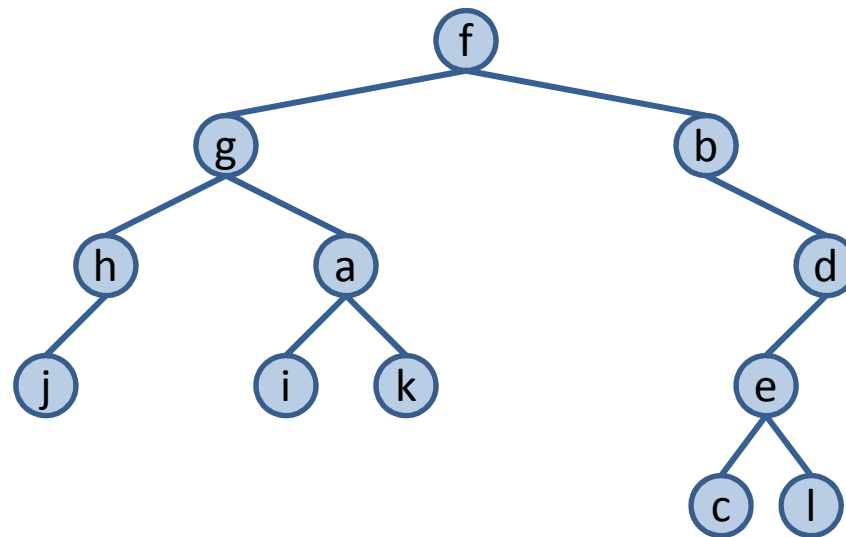
Construcción de un árbol binario (II)



Construcción de un árbol binario (II)



Construcción de un árbol binario (II)



```
Abin(); // constructor  
void insertarRaizB(const T& e);  
void insertarHijoIzqdoB(nodo n, const T& e);  
void insertarHijoDrchoB(nodo n, const T& e);
```

Especificación de operaciones:

Abin ()

Post: Crea y devuelve un árbol vacío.

void insertarRaizB (const T& e)

Pre: El árbol está vacío.

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

void insertarHijoIzqdoB (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 insertarHijoDrchoB (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 eliminarHijoIzqdoB (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 eliminarHijoDrchoB (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 eliminarRaizB ()

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

Post: Destruye el nodo raíz. El árbol queda vacío

bool arbolVacioB () 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 raízB () const

Post: Devuelve el nodo raíz del árbol. Si el árbol está vacío, devuelve *NODO_NULO*.

nodo padreB (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 hijoIzqdoB (nodo n) const

Pre: *n* es un nodo del árbol.

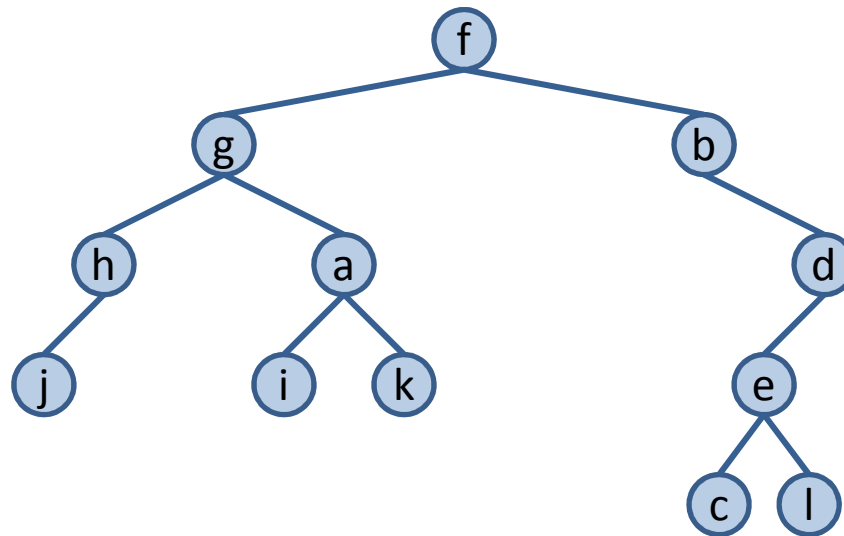
Post: Devuelve el nodo hijo izquierdo del nodo *n*. Si no existe, devuelve *NODO_NULO*.

nodo hijoDrchoB (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*.

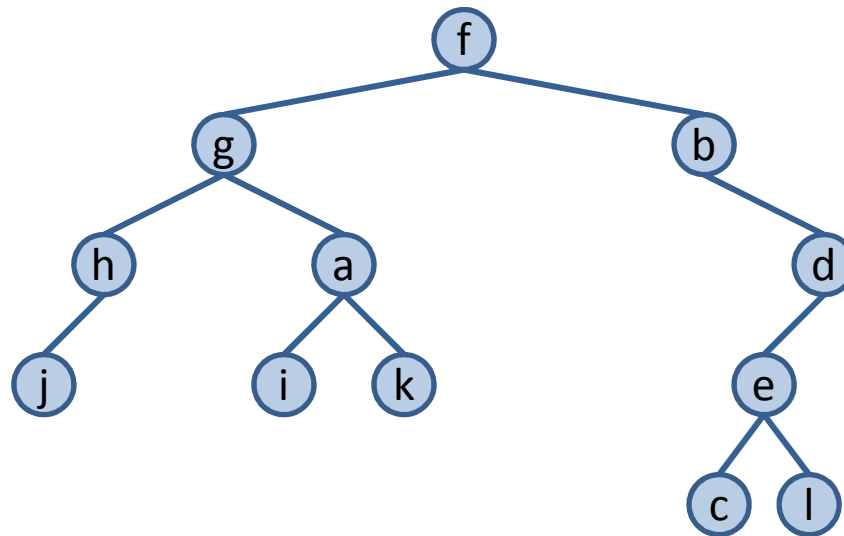
Implementación vectorial de árboles binarios



	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			

maxNodos-1

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			
hizq	1	5	7	*	9	6	*	*	*	11	*	*			
hder	3	2	8	4	*	*	*	*	*	10	*	*			

Implementación vectorial de árboles binarios

Inserción y eliminación

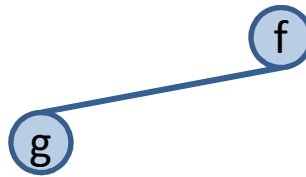
f

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

maxNodos-1

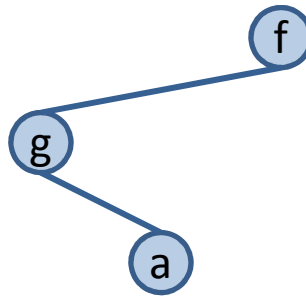
Implementación vectorial de árboles binarios

Inserción y eliminación

[illegible]

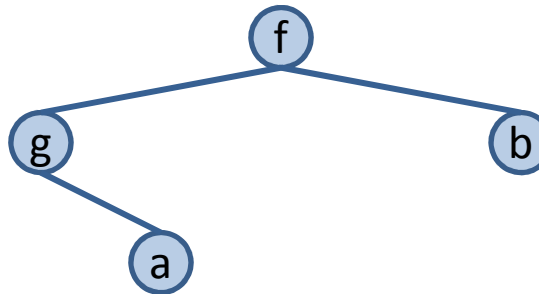
Implementación vectorial de árboles binarios

Inserción y eliminación

[illegible]

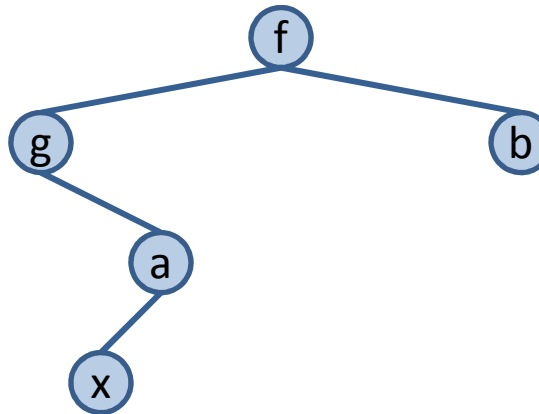
Implementación vectorial de árboles binarios

Inserción y eliminación

[illegible]

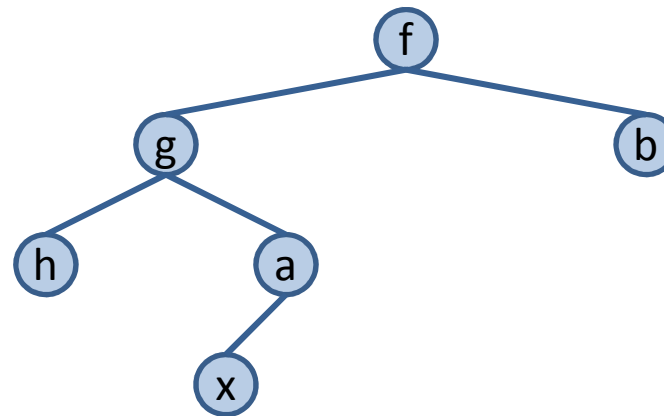
Implementación vectorial de árboles binarios

Inserción y eliminación

[illegible]

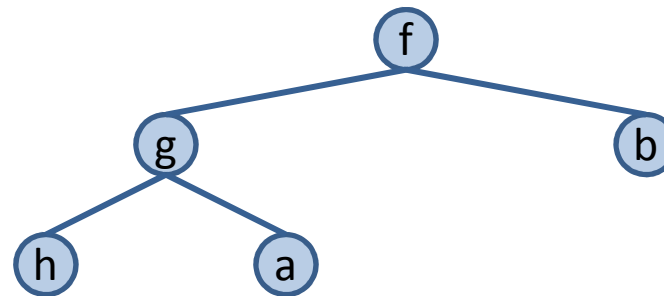
Implementación vectorial de árboles binarios

Inserción y eliminación

[illegible]

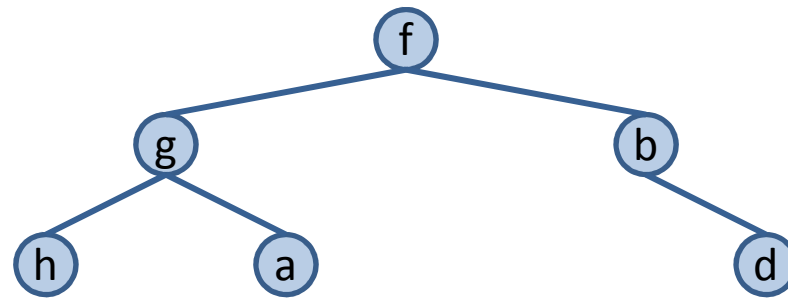
Implementación vectorial de árboles binarios

Inserción y eliminación

[illegible]

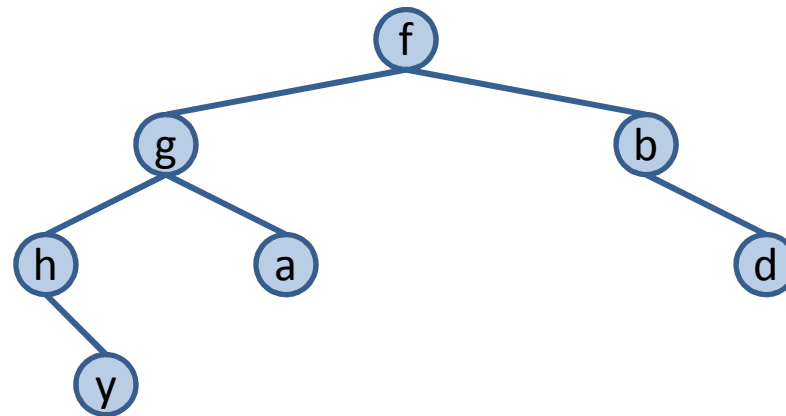
Implementación vectorial de árboles binarios

Inserción y eliminación

[illegible]

Implementación vectorial de árboles binarios

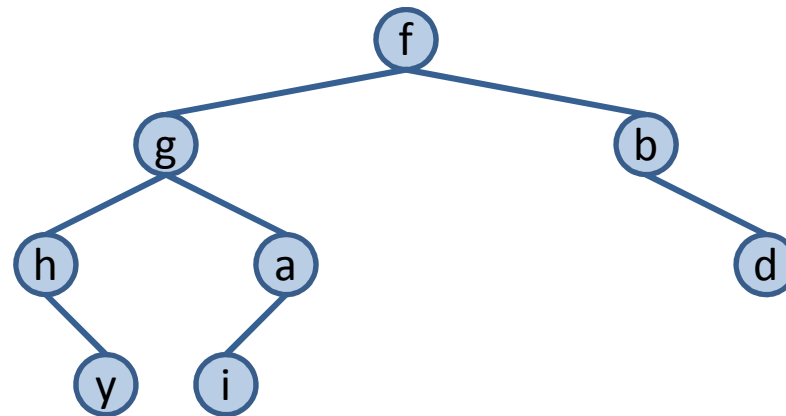
Inserción y eliminación



	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

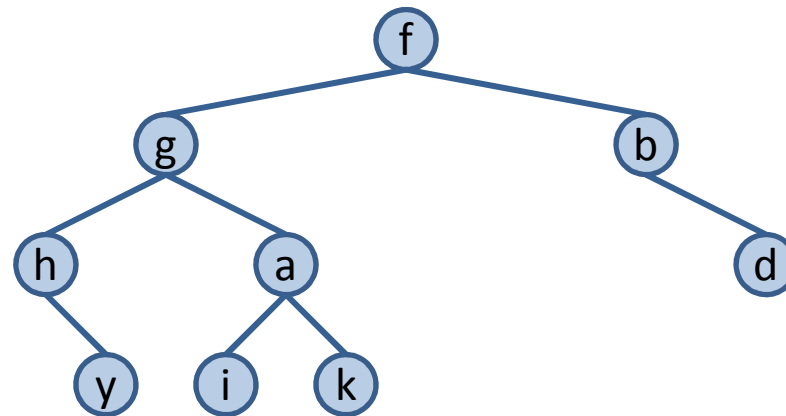
Inserción y eliminación



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

Implementación vectorial de árboles binarios

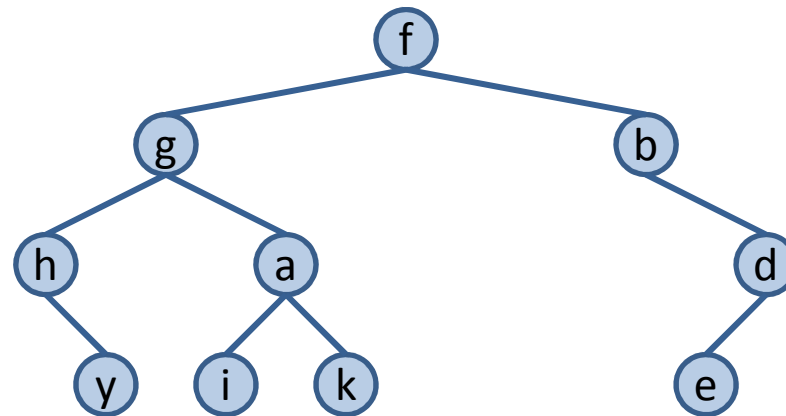
Inserción y eliminación



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	maxNodos-1
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	*	*	*						

Implementación vectorial de árboles binarios

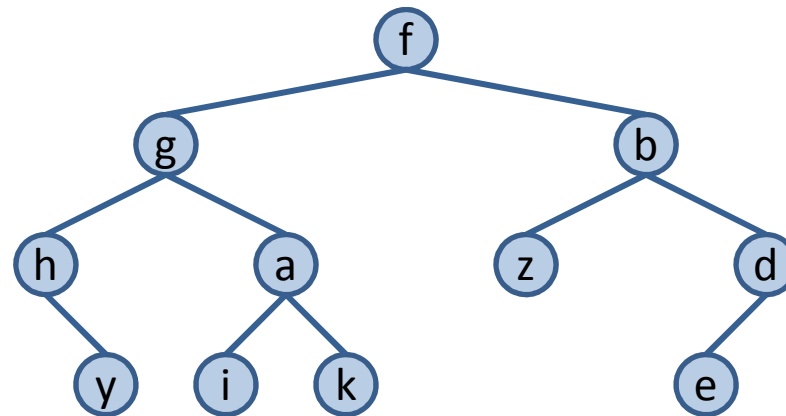
Inserción y eliminación



	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					
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

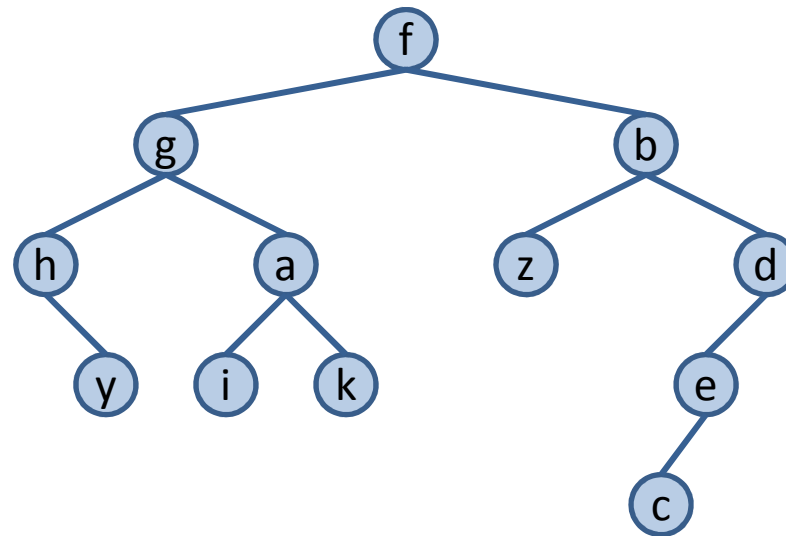
Inserción y eliminación



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	maxNodos-1
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	*	*	*	*	*				

Implementación vectorial de árboles binarios

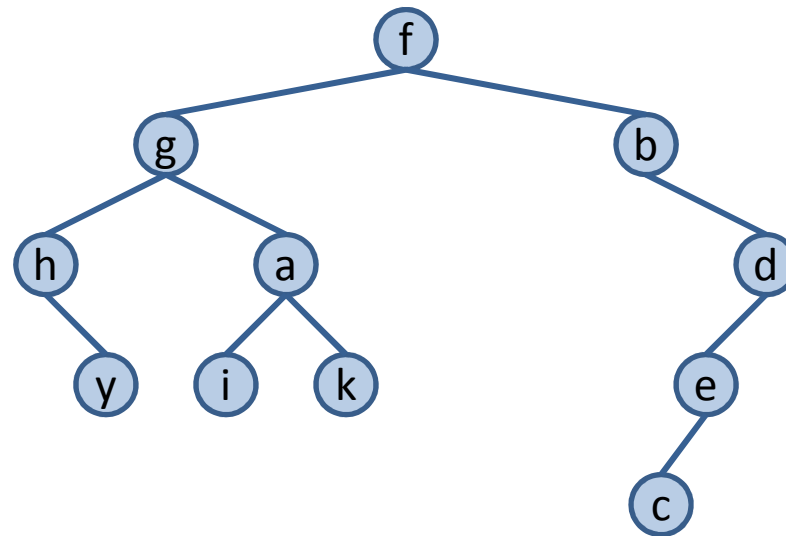
Inserción y eliminación



	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

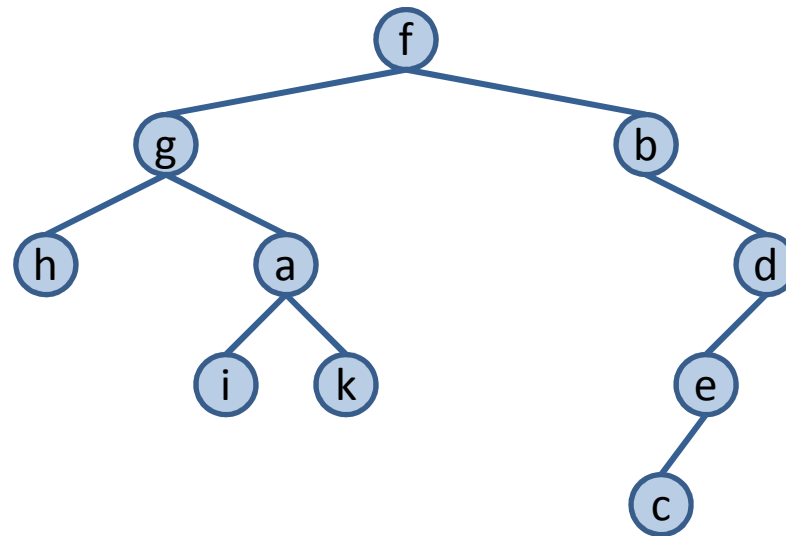
Inserción y eliminación



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

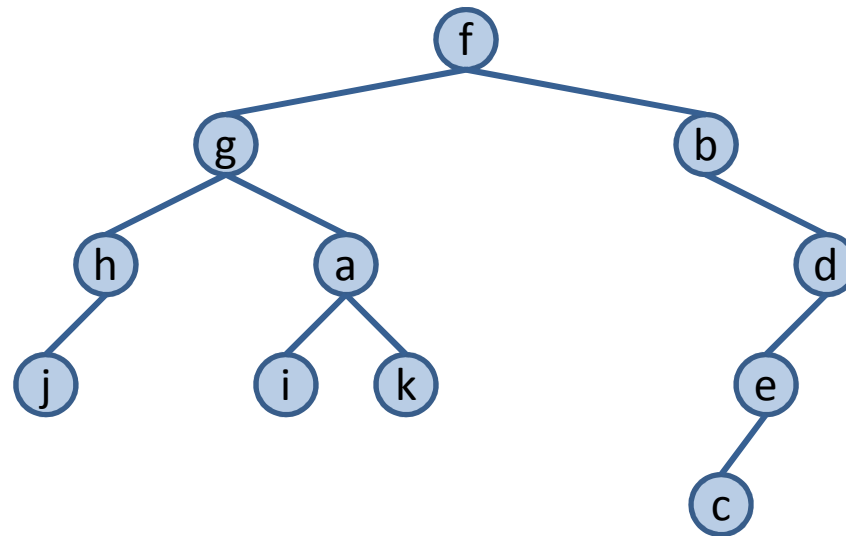
Inserción y eliminación



	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

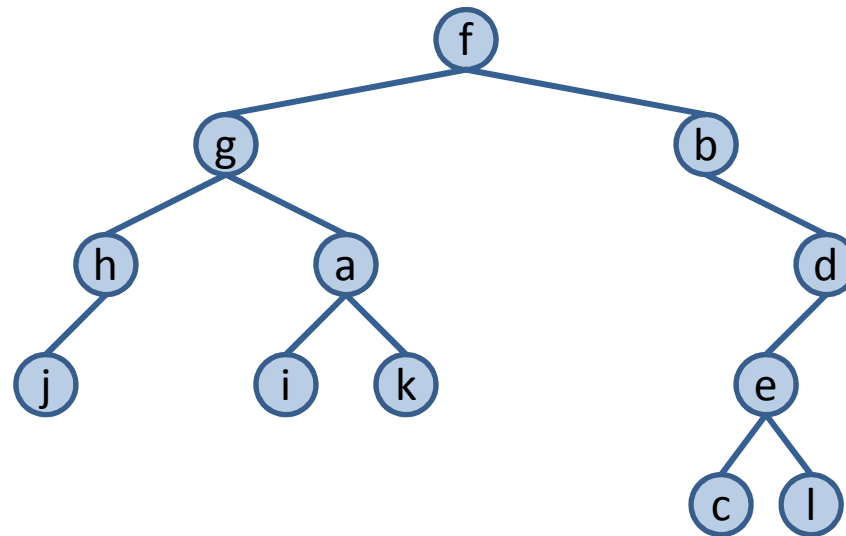
Inserción y eliminación



	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			maxNodos-1
padre	*	0	1	0	3	1	5	2	2	4		9			
hizq	1	5	7	*	9	6	*	*	*	11		*			
hder	3	2	8	4	*	*	*	*	*	*		*			

Implementación vectorial de árboles binarios

Inserción y eliminación

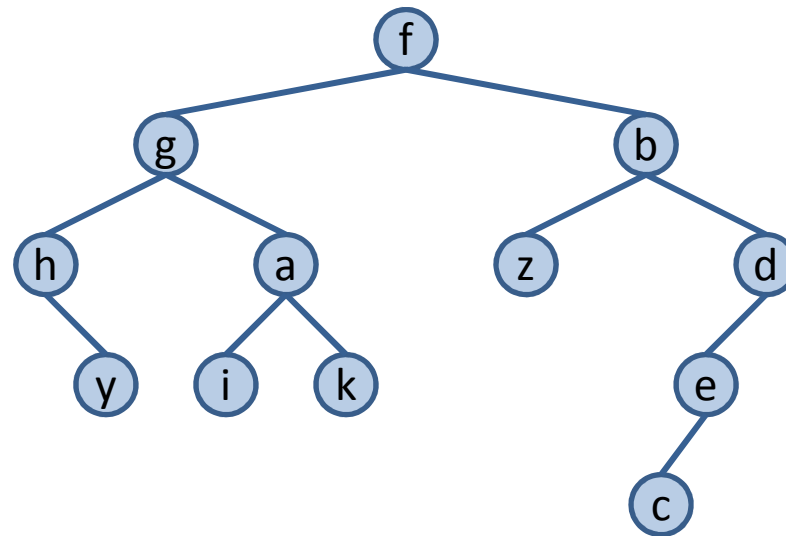


	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
elto	f	g	a	b	d	h	j	i	k	e	l	c			
padre	*	0	1	0	3	1	5	2	2	4	9	9			
hizq	1	5	7	*	9	6	*	*	*	11	*	*			
hder	3	2	8	4	*	*	*	*	*	10	*	*			

maxNodos-1

Implementación vectorial de árboles binarios

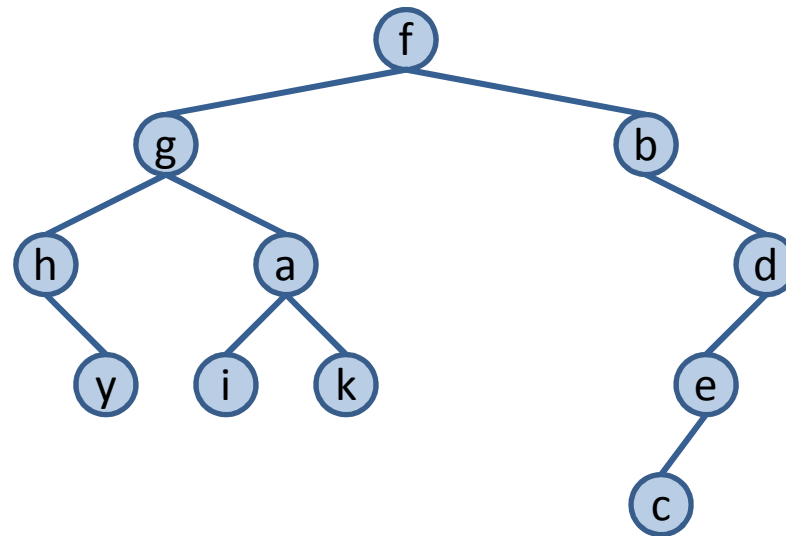
Inserción y eliminación



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	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



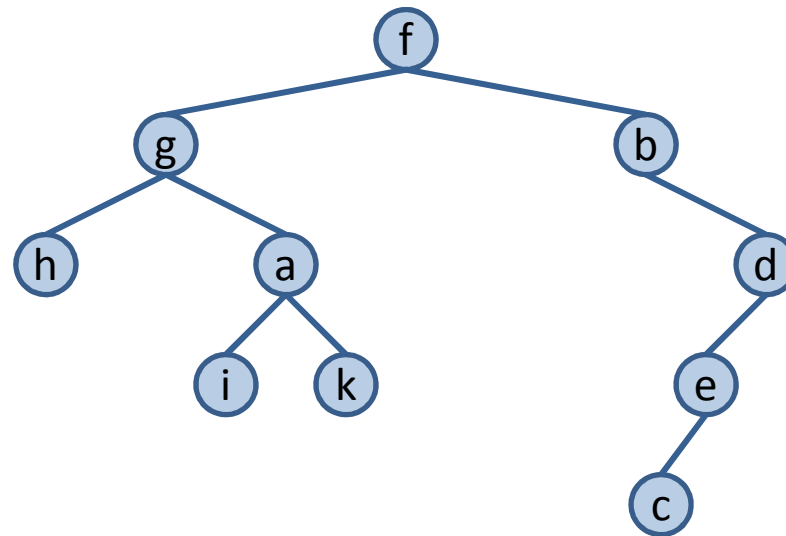
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
elto	f	g	a	b	d	h	y	i	k	e	z	c		
padre	0	0	1	0	3	1	5	2	2	4	*	9	*	*
hizq	1	5	7	*	9	*	*	*	*	11	*	*		
hder	3	2	8	4	*	6	*	*	*	*	*	*		

maxNodos-1

	*

Implementación vectorial de árboles binarios

Inserción y eliminación

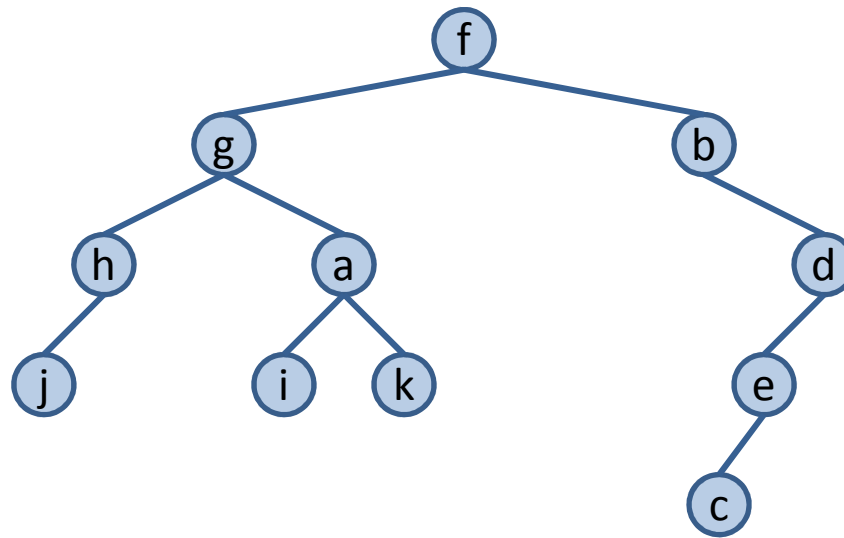
[illegible]

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	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	*	*	*	*	*	*	*	*			

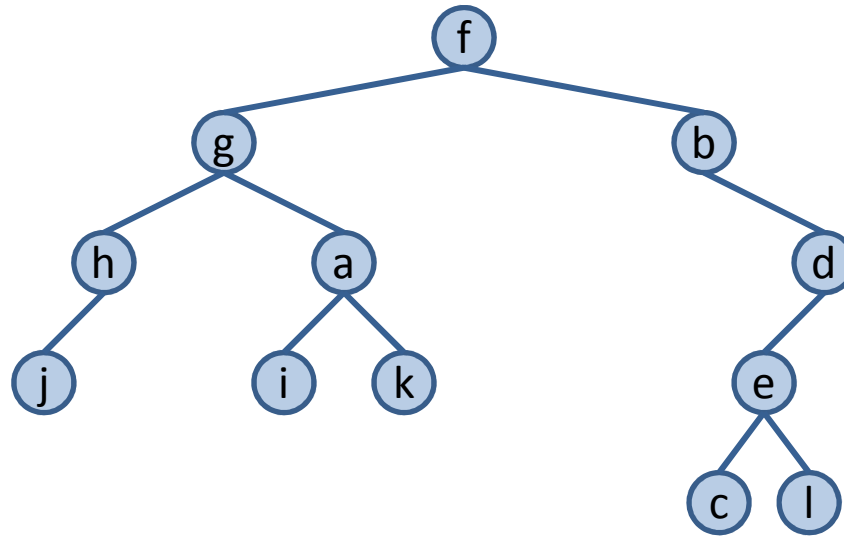
	maxNodos-1
	*

Implementación vectorial de árboles binarios

Inserción y eliminación

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

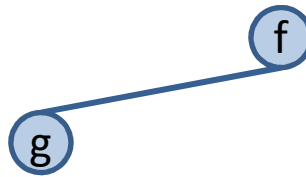
numNodos 1

	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



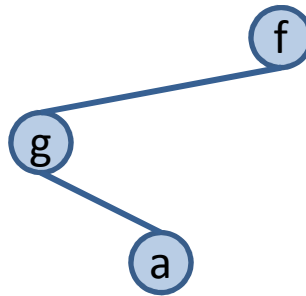
numNodos 2

	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



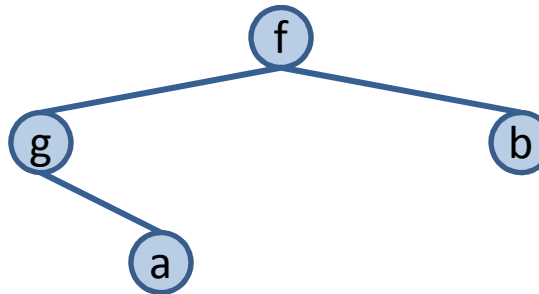
numNodos 3

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

maxNodos-1

Implementación vectorial de árboles binarios

Inserción y eliminación eficientes

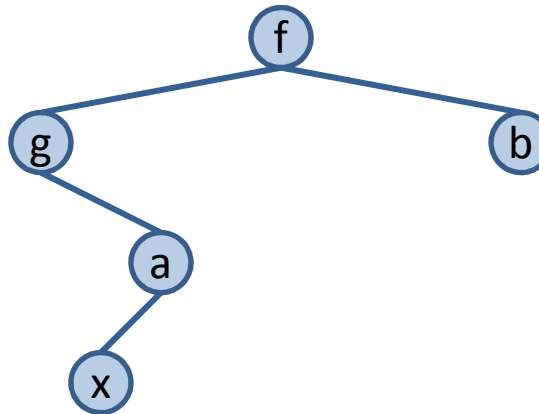


numNodos 4

[illegible]

Implementación vectorial de árboles binarios

Inserción y eliminación eficientes

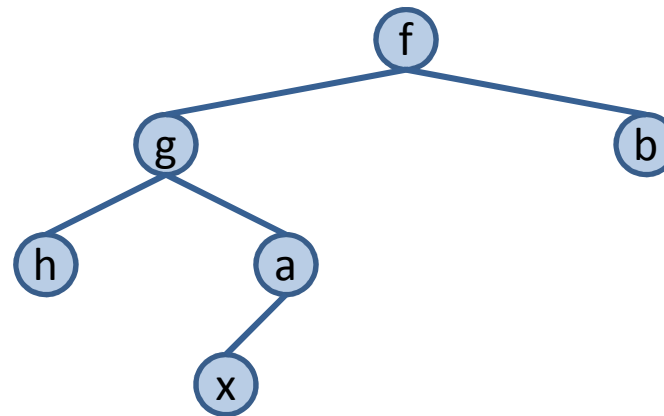


numNodos 5

[illegible]

Implementación vectorial de árboles binarios

Inserción y eliminación eficientes

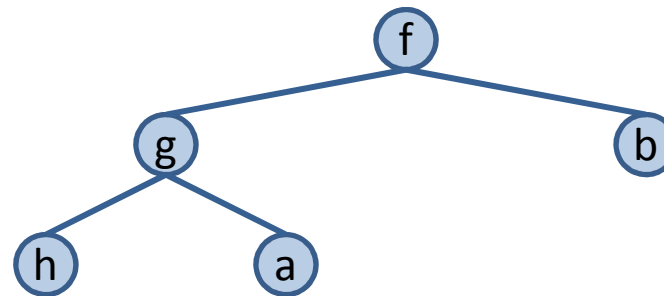


numNodos 6

[illegible]

Implementación vectorial de árboles binarios

Inserción y eliminación eficientes

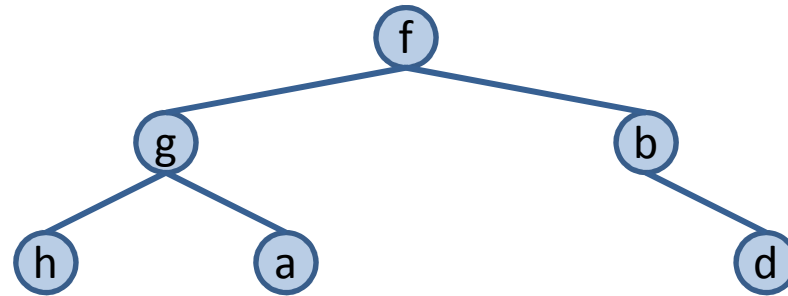


numNodos 5

[illegible]

Implementación vectorial de árboles binarios

Inserción y eliminación eficientes

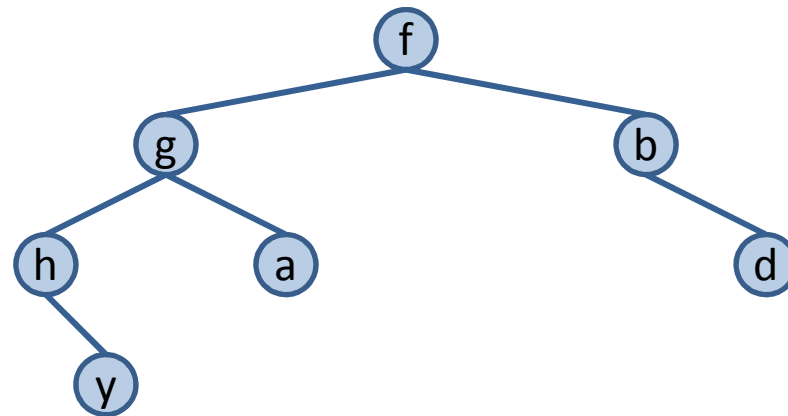


numNodos 6

[illegible]

Implementación vectorial de árboles binarios

Inserción y eliminación eficientes

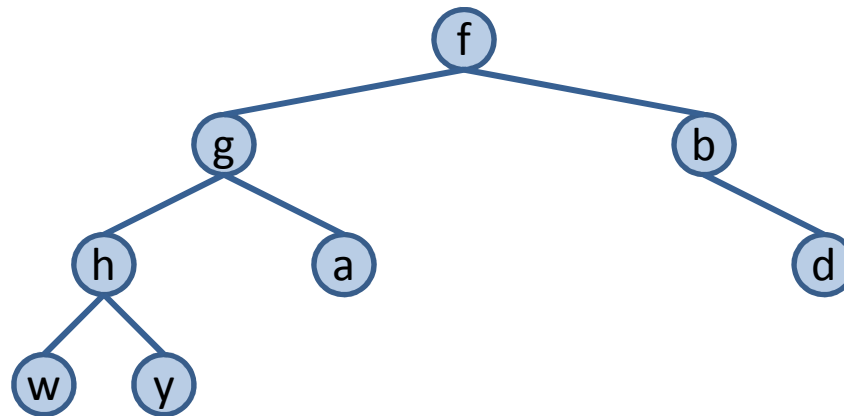


numNodos 7

	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

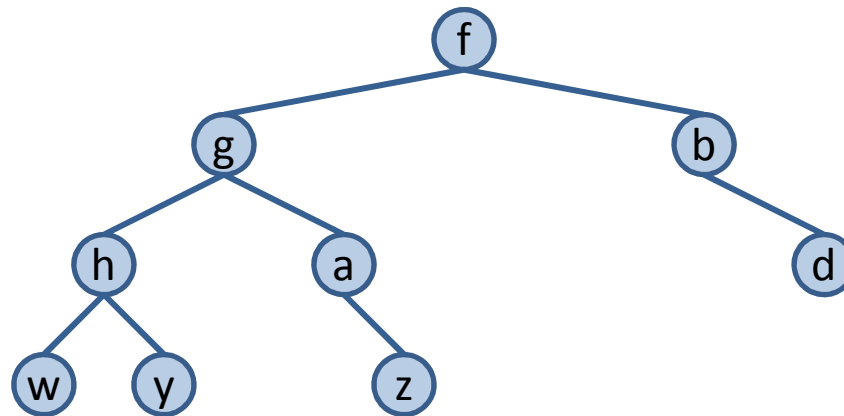


numNodos 8

	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

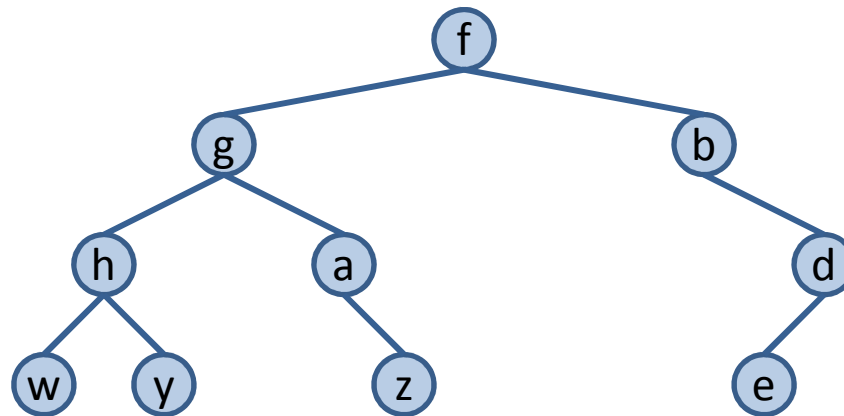


numNodos 9

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	maxNodos-1
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	*	*	*	*						

Implementación vectorial de árboles binarios

Inserción y eliminación eficientes

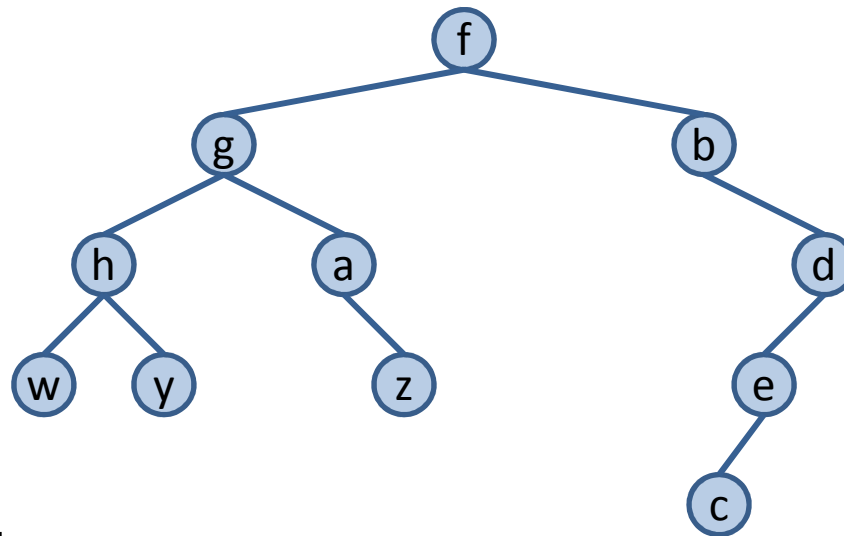


numNodos 10

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

Implementación vectorial de árboles binarios

Inserción y eliminación eficientes

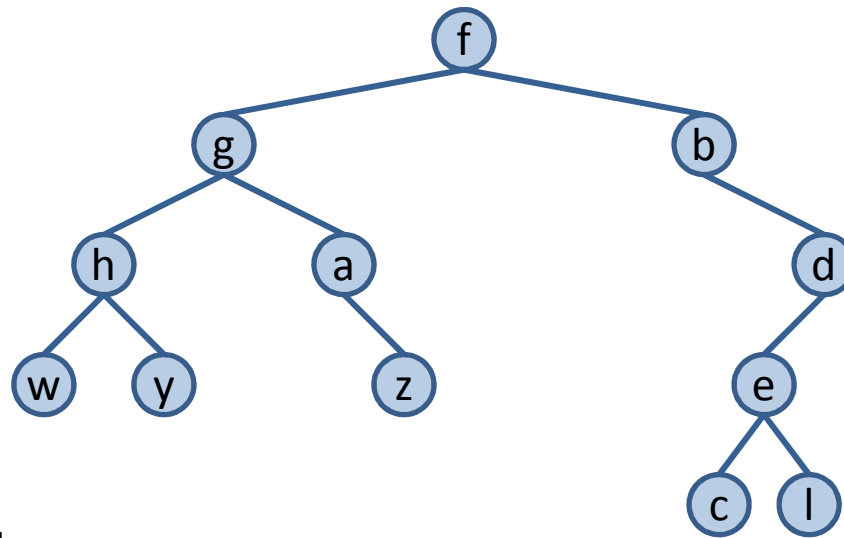


numNodos **11**

	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

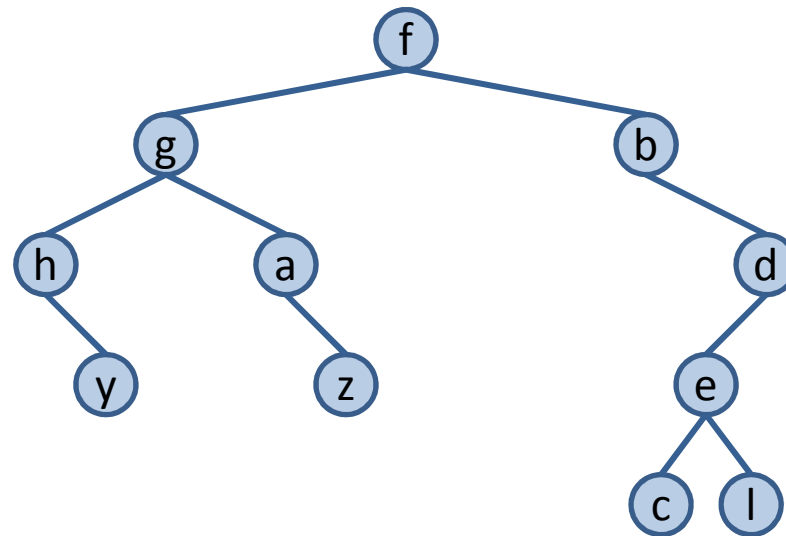


numNodos 12

	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

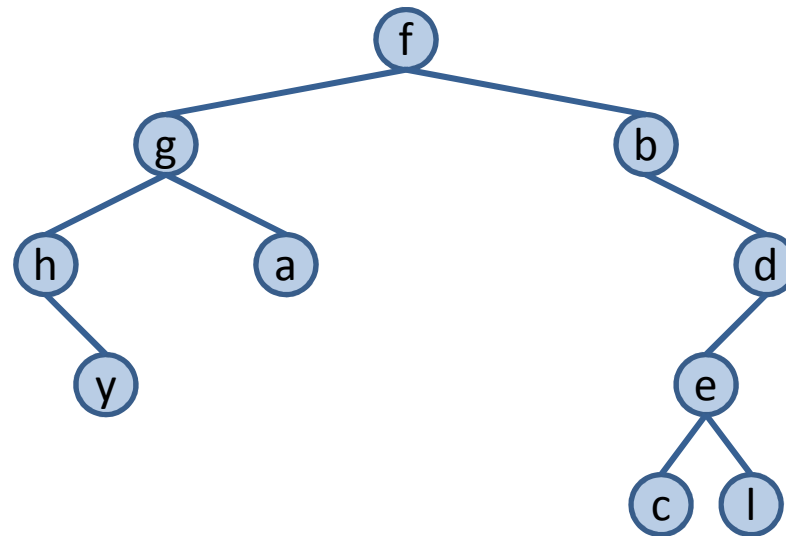


numNodos 11

	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

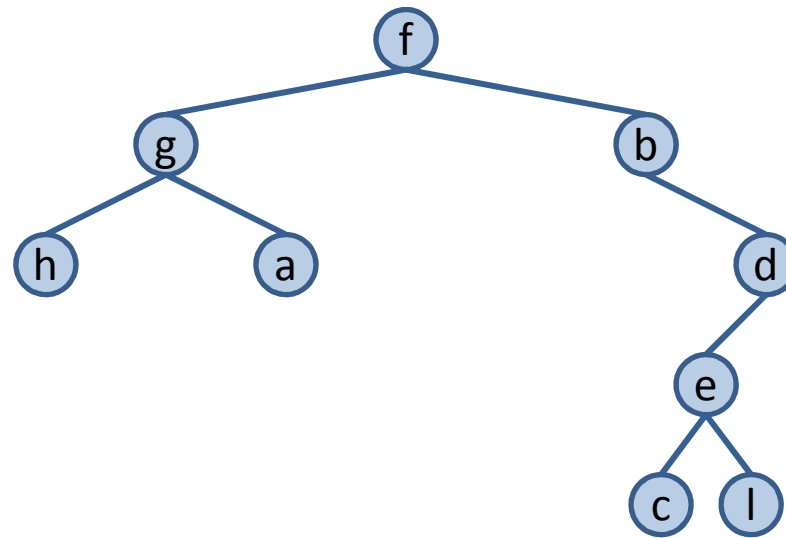


numNodos 10

	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

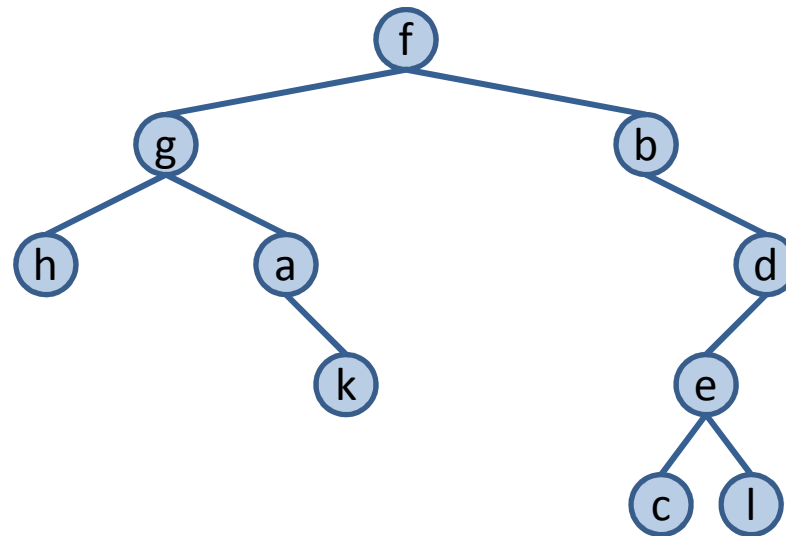


numNodos 9

	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

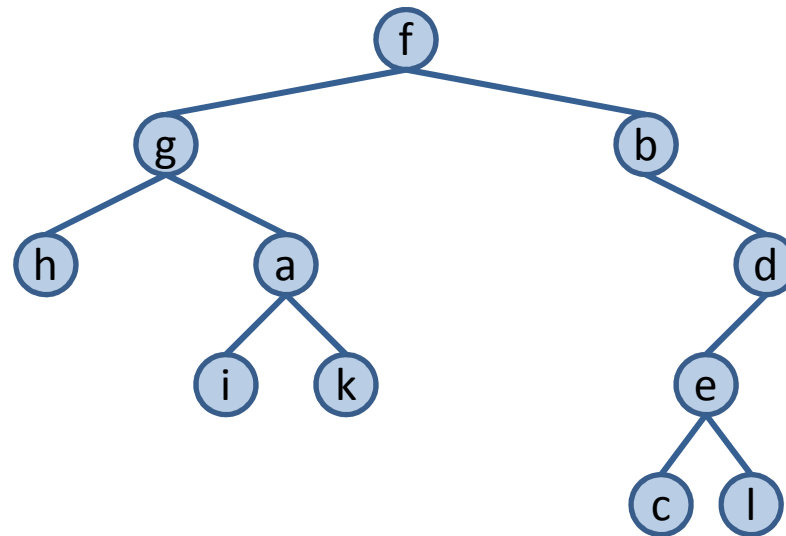


numNodos 10

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	maxNodos-1	
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	*	*	*	*	*				

Implementación vectorial de árboles binarios

Inserción y eliminación eficientes



numNodos 11

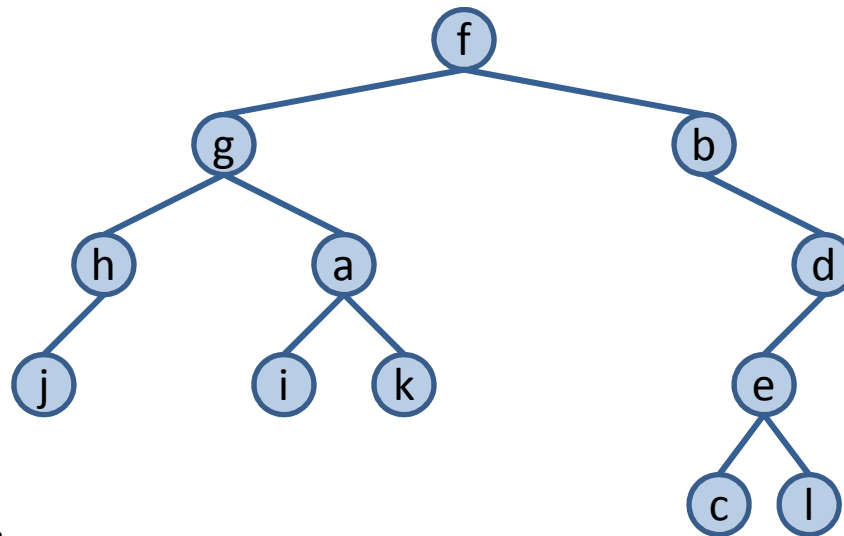
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	maxNodos-1
elto	f	g	a	b	h	d	e	l	c	k	i	l			
padre	*	0	1	0	1	3	5	6	6	2	2	9			
hizq	1	4	10	*	*	6	8	*	*	*	*	*			
hder	3	2	9	5	*	*	7	*	*	*	*	*			

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 int nodo; // índice de la matriz
                        // entre 0 y maxNodos-1
    static const nodo NODO_NULO;
    explicit Abin(size_t maxNodos);           // constructor
    void insertarRaizB(const T& e);
    void insertarHijoIzqdoB(nodo n, const T& e);
    void insertarHijoDrchoB(nodo n, const T& e);
    void eliminarHijoIzqdoB(nodo n);
    void eliminarHijoDrchoB(nodo n);
    void eliminarRaizB();
    ~Abin();                                   // destructor
    bool arbolVacioB() const;
    const T& elemento(nodo n) const; // acceso a elto, lectura
    T& elemento(nodo n);             // acceso a elto, lectura/escritura

```

```

    nodo raizB() const;
    nodo padreB(nodo n) const;
    nodo hijoIzqdoB(nodo n) const;
    nodo hijoDrchoB(nodo n) const;
    Abin(const Abin<T>& a);                // ctor. de copia
    Abin<T>& operator =(const Abin<T>& a);  // asignación
private:
    struct celda {
        T elto;
        nodo padre, hizq, hder;
    };
    celda *nodos;    // vector de nodos
    int maxNodos;    // tamaño del vector
    int numNodos;    // número de nodos del árbol
};

/* Definición del nodo nulo */
template <typename T>
const typename Abin<T>::nodo Abin<T>::NODO_NULO(-1);

```



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

```
template <typename T>
void Abin<T>::insertarRaizB(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>
void Abin<T>::insertarHijoIzqdoB(Abin<T>::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>
void Abin<T>::insertarHijoDrchoB(Abin<T>::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>::eliminarHijoIzqdoB(Abin<T>::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>::eliminarHijoDrchoB(Abin<T>::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>::eliminarRaizB()
{
    assert(numNodos == 1);
    numNodos = 0;
}
```

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



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

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

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

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

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

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

```
template <typename T> inline
typename Abin<T>::nodo Abin<T>::hijoDrchoB(Abin<T>::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];
}
```

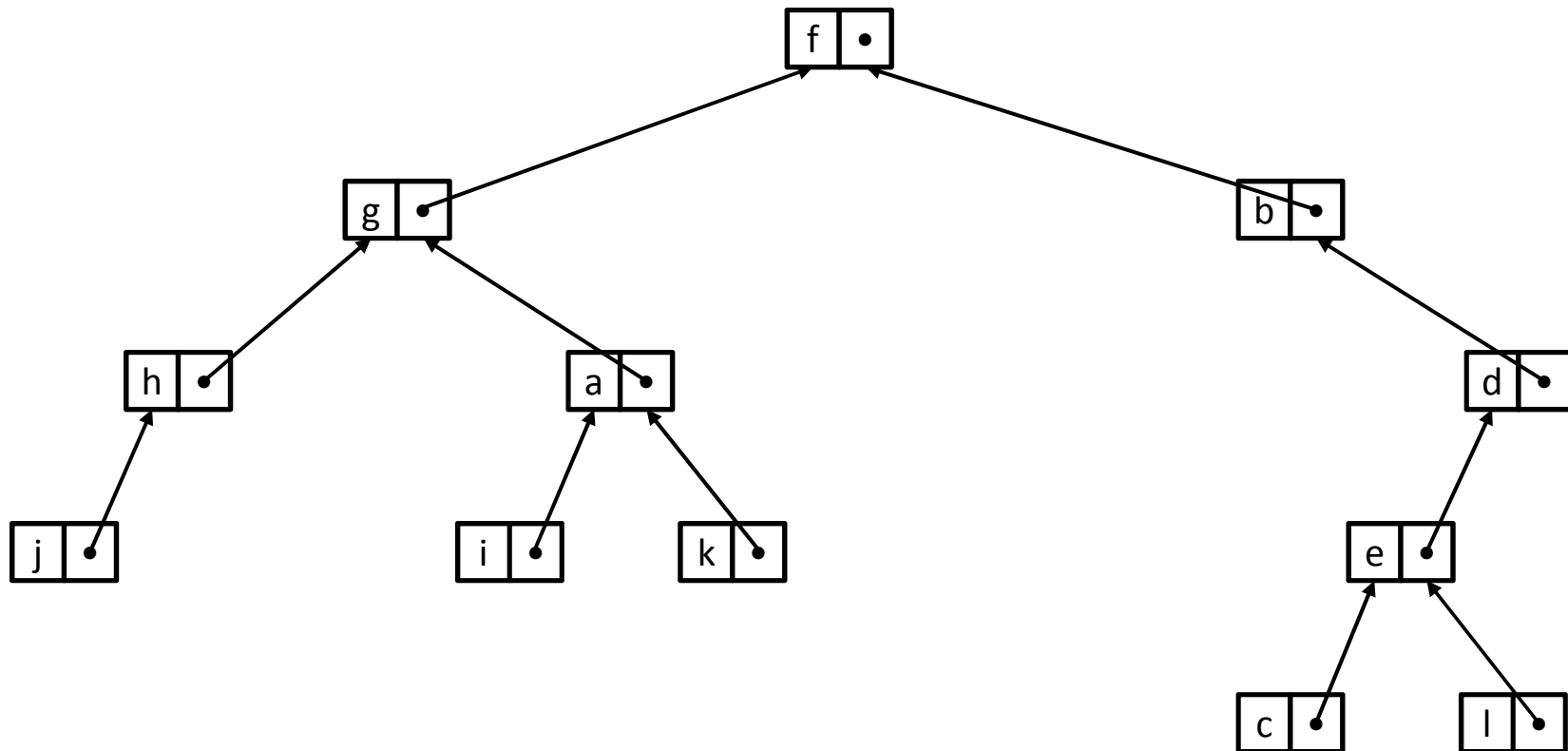
```

template <typename T>
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;
}

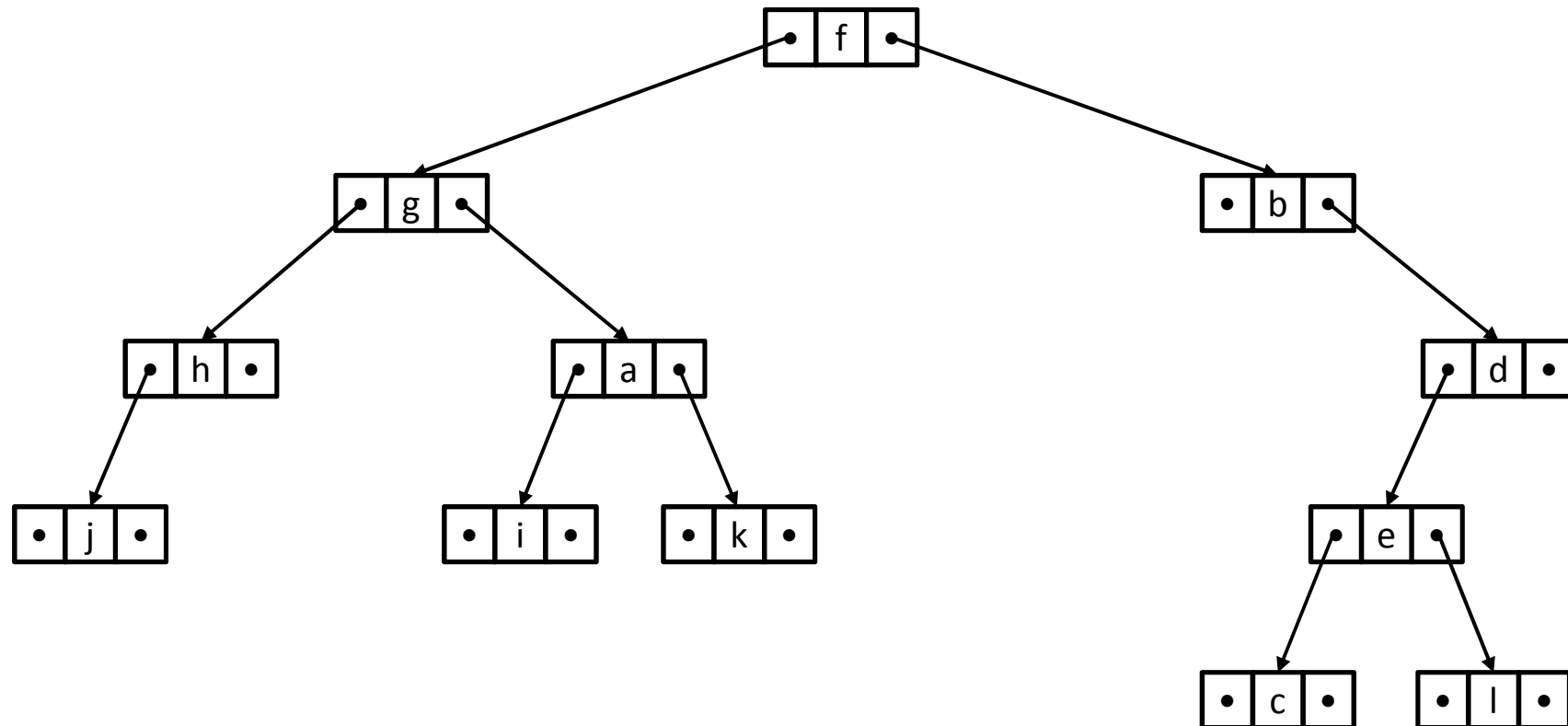
#endif // ABIN_VEC0_H

```

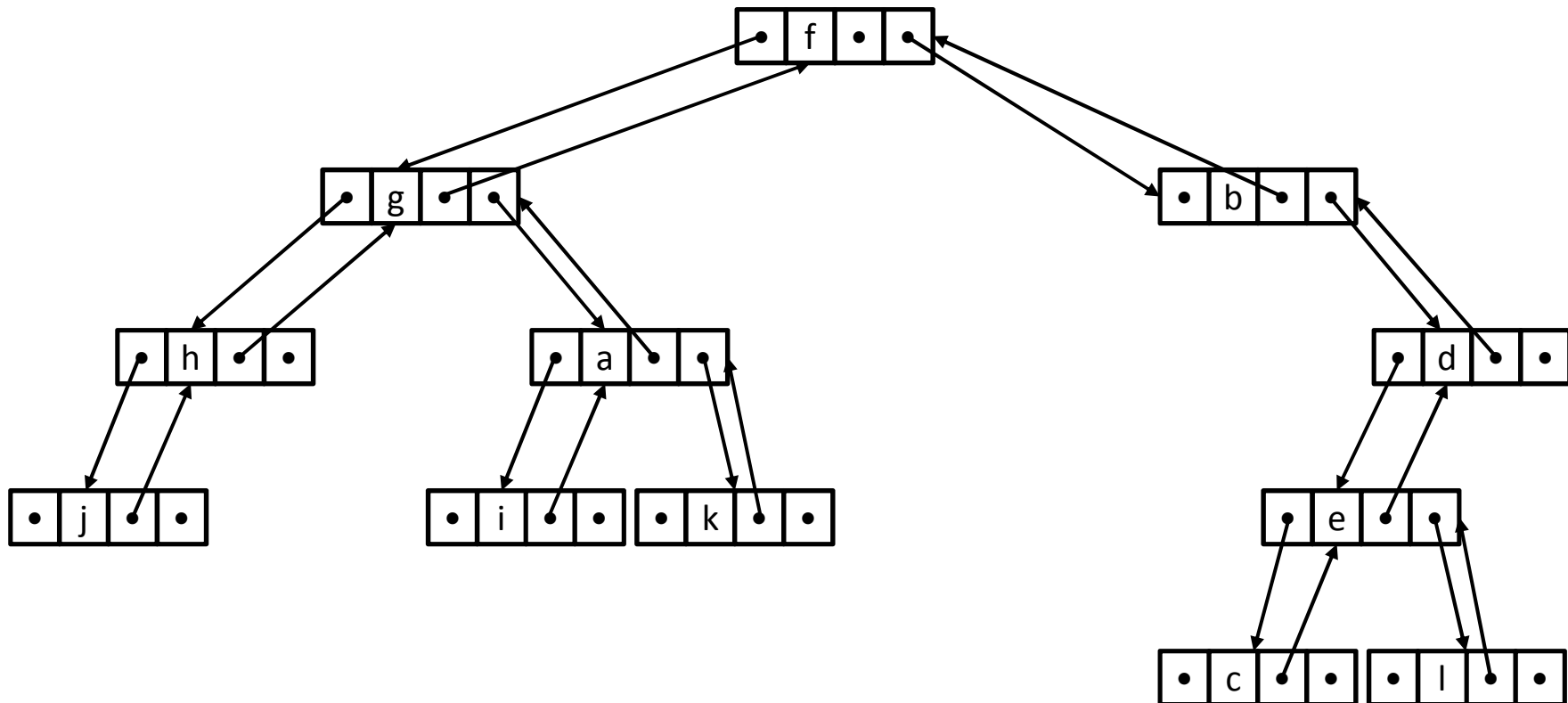
Implementación de un árbol binario usando celdas enlazadas



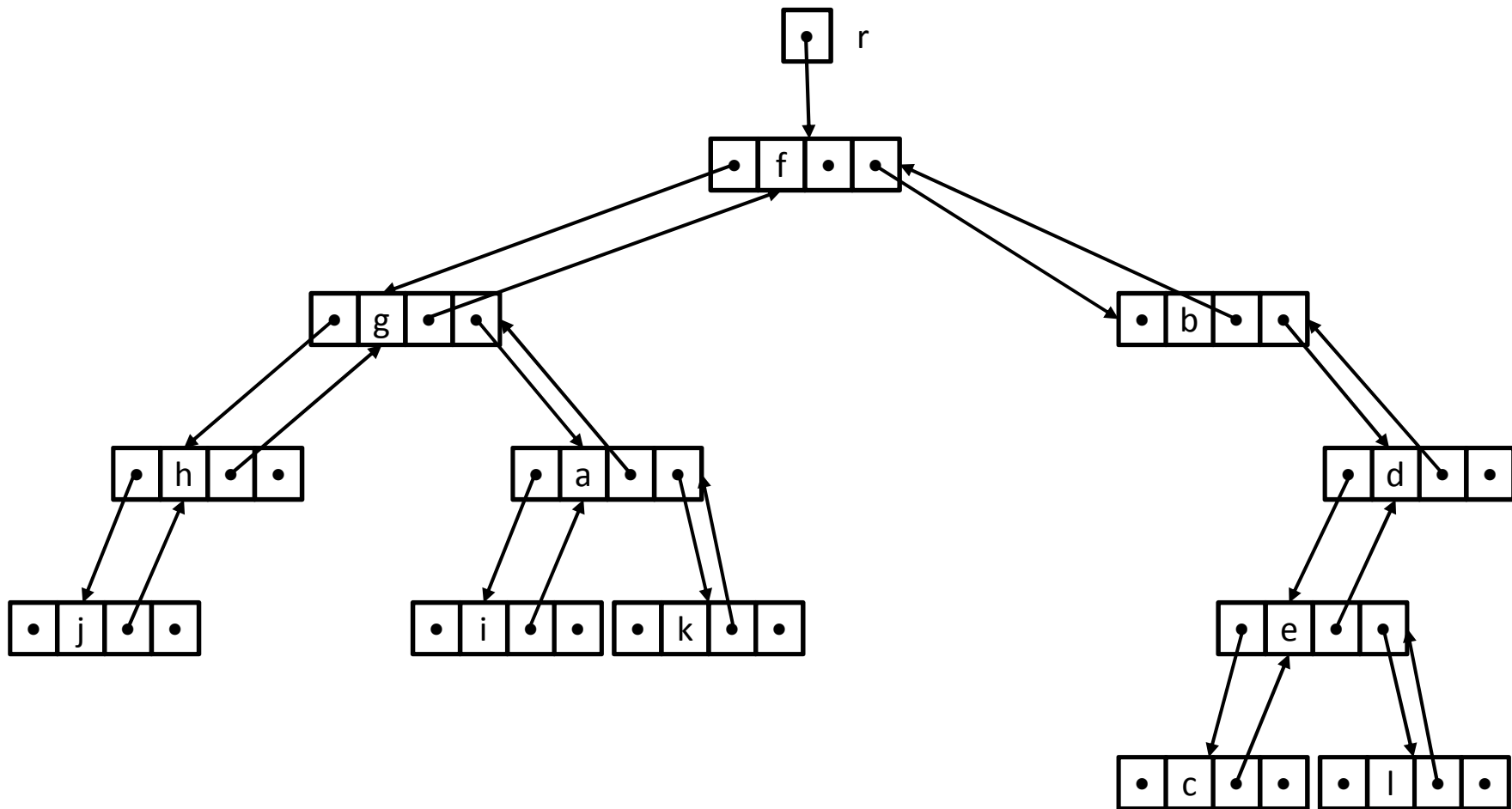
Implementación de un árbol binario usando celdas enlazadas



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;
    Abin();           // constructor
    void insertarRaizB(const T& e);
    void insertarHijoIzqdoB(nodo n, const T& e);
    void insertarHijoDrchoB(nodo n, const T& e);
    void eliminarHijoIzqdoB(nodo n);
    void eliminarHijoDrchoB(nodo n);
    void eliminarRaizB();
    ~Abin();          // destructor
    bool arbolVacioB() const;
    const T& elemento(nodo n) const; // acceso a elto, lectura
    T& elemento(nodo n);             // acceso a elto, lectura/escritura
    nodo raizB() const;

```

```

nodo padreB(nodo n) const;
nodo hijoIzqdoB(nodo n) const;
nodo hijoDrchoB(nodo n) const;
Abin(const Abin<T>& a); // ctor. de copia
Abin<T>& operator =(const Abin<T>& a); //asignación de árboles
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(0);

```

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



```
template <typename T>
inline void Abin<T>::insertarRaizB (const T& e)
{
    assert(r == NODO_NULO);    // árbol vacío
    r = new celda(e);
}
```

```
template <typename T>
inline void Abin<T>::insertarHijoIzqdoB(Abin<T>::nodo n,
    const T& e)
{
    assert(n != NODO_NULO);
    assert(n->hizq == NODO_NULO);    // no existe hijo
    n->hizq = new celda(e, n);
}
```

```

template <typename T> inline
void Abin<T>::insertarHijoDrchoB(Abin<T>::nodo n, const T& e)
{
    assert(n != NODO_NULO);
    assert(n->hder == NODO_NULO);    // no existe hijo
    n->hder = new celda(e, n);
}

```

```

template <typename T>
inline void Abin<T>::eliminarHijoIzqdoB(Abin<T>::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;
}

```

```

template <typename T>
inline void Abin<T>::eliminarHijoDrchoB(Abin<T>::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>::eliminarRaizB()
{
    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 Abin<T>::~~Abin()  
{  
    destruirNodos(r); // vacía el árbol  
}
```

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

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

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

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

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



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

```
template <typename T> inline
typename Abin<T>::nodo Abin<T>::hijoDrchoB(Abin<T>::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);
}
```

```
template <typename T>
Abin<T>& Abin<T>::operator =(const Abin<T>& a)
{
    if (this != &a)          // evitar autoasignación
    {
        this->~Abin();      // vaciar el árbol
        r = copiar(a.r);
    }
    return *this;
}
```

```
// Métodos privados
```

```
// Destruye un nodo y todos sus descendientes
```

```
template <typename T>
```

```
void Abin<T>::destruirNodos(Abin<T>::nodo& n)
```

```
{
```

```
    if (n != NODO_NULO)
```

```
    {
```

```
        destruirNodos(n->hizq);
```

```
        destruirNodos(n->hder);
```

```
        delete n;
```

```
        n = NODO_NULO;
```

```
    }
```

```
}
```

// Devuelve una copia de un nodo y todos sus descendientes

```
template <typename T>
```

```
typename Abin<T>::nodo Abin<T>::copiar(Abin<T>::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
```