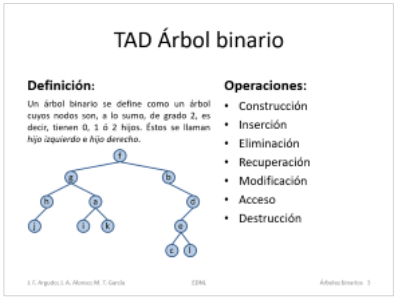
Estructuras de Datos no Lineales

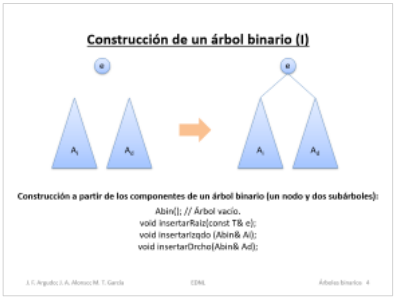
1.1. Árboles binarios

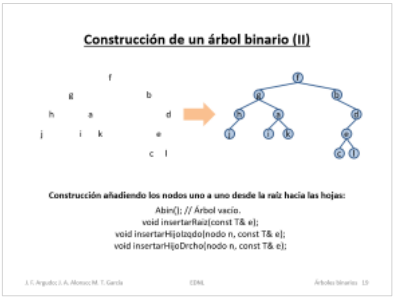
José Fidel Argudo Argudo
José Antonio Alonso de la Huerta
Mª Teresa García Horcajadas

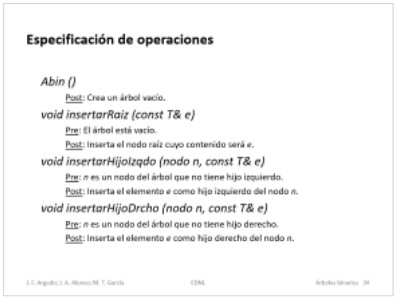
Versión 2.3

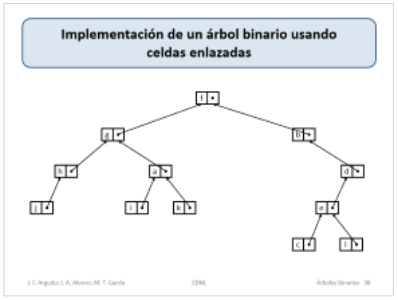
Contenido

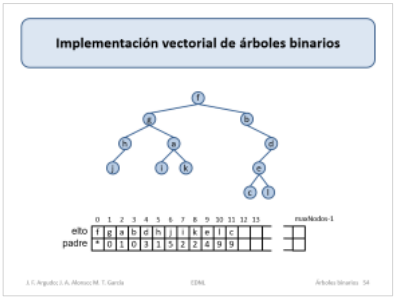

TAD Árbol binario
Definición:
Un árbol binario se define como un árbol cuyos nodos son, a lo sumo, de grado 2, es decir, tienen 0, 1 ó 2 hijos. Éstos se llaman hijo izquierdo e hijo derecho.
Operaciones:
Construcción
Inserción
Eliminación
Recuperación
Modificación
Acceso
Destrucción

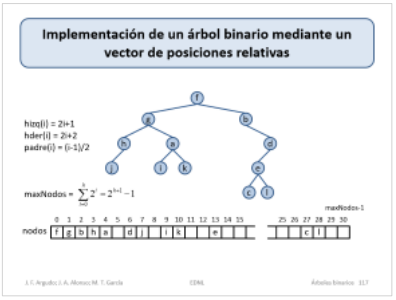

Construcción de un árbol binario (I)
Construcción a partir de los componentes de un árbol binario (un nodo y dos subárboles):
Abin(); // Árbol vacío.
void insertarRaiz(const T& e);
void insertarIzqdo (Abin& Ai);
void insertarDrcho(Abin& Ad);


Construcción de un árbol binario (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);


Especificación de operaciones
Abin ()
Post: Crea 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.


Implementación de un árbol binario usando celdas enlazadas

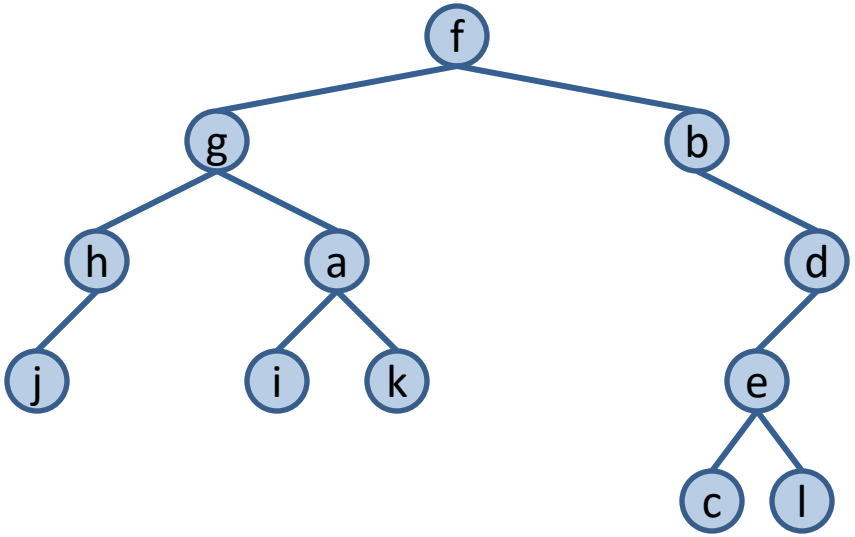

Implementación vectorial de árboles binarios
elto
padre
maxNodos-1


Implementación de un árbol binario mediante un vector de posiciones relativas
hizq(i) = 2i+1
hder(i) = 2i+2
padre(i) = (i-1)/2
maxNodos = $\sum_{i=0}^{h} 2^i = 2^{h+1} - 1$
nodos
maxNodos-1

TAD Árbol binario

Definición:

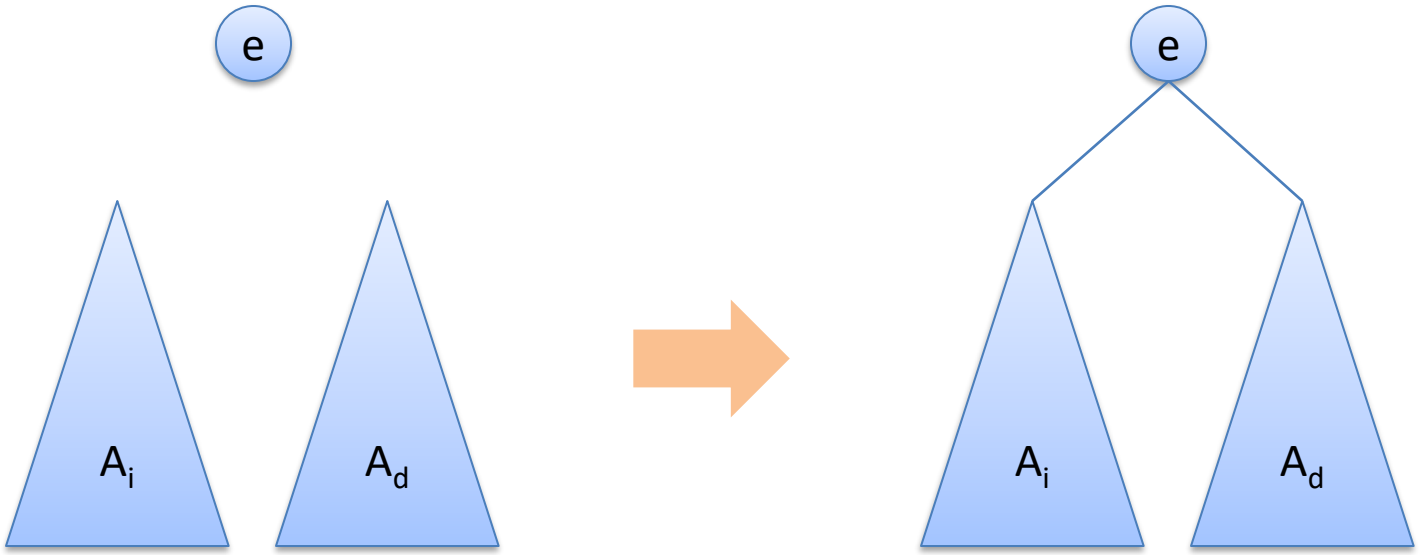
Un árbol binario se define como un árbol cuyos nodos son, a lo sumo, de grado 2, es decir, tienen 0, 1 ó 2 hijos. Éstos se llaman *hijo izquierdo* e *hijo derecho*.



Operaciones:

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

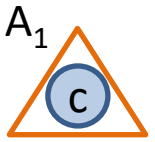
Construcción de un árbol binario (I)



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

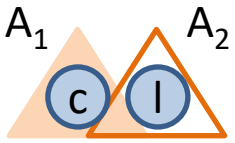
```
Abin(); // Árbol vacío.
void insertarRaiz(const T& e);
void insertarIzqdo (Abin& Ai);
void insertarDrcho(Abin& Ad);
```

Construcción de un árbol binario (I)



Abin A_1;
A_1.insertarRaiz('c');

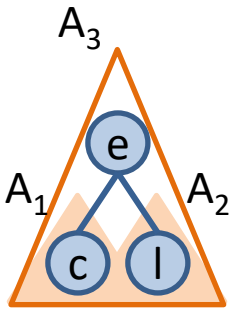
Construcción de un árbol binario (I)



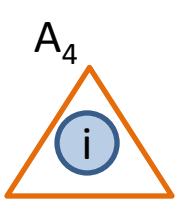
Abin A_2;
A_2.insertarRaiz('l');

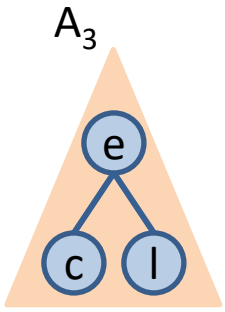
Construcción de un árbol binario (I)

Abin A_3; A_3.insertarRaiz('e');
A_3.insertarIzqdo(A_1);
A_3.insertarDrcho(A_2);



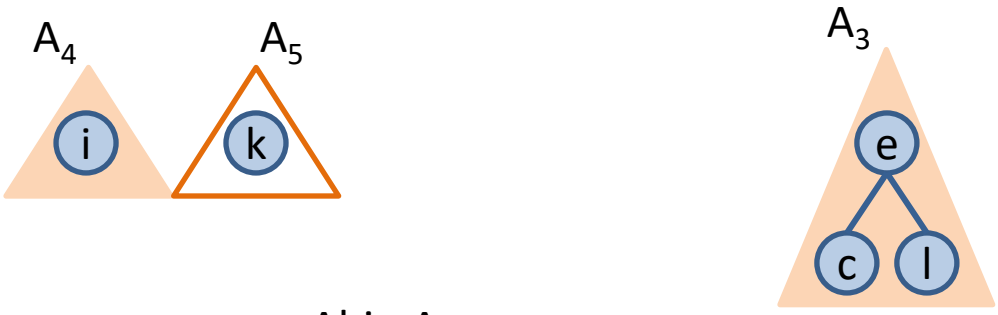
Construcción de un árbol binario (I)





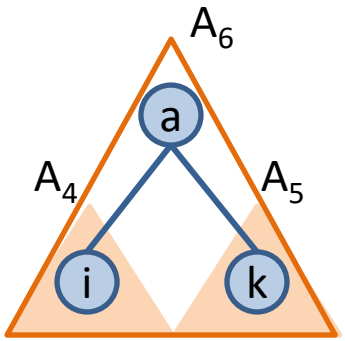
Abin A_4;
A_4.insertarRaiz('i');

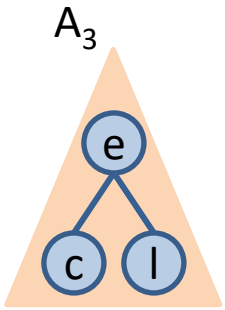
Construcción de un árbol binario (I)



Abin A_5;
A_5.insertarRaiz('k');

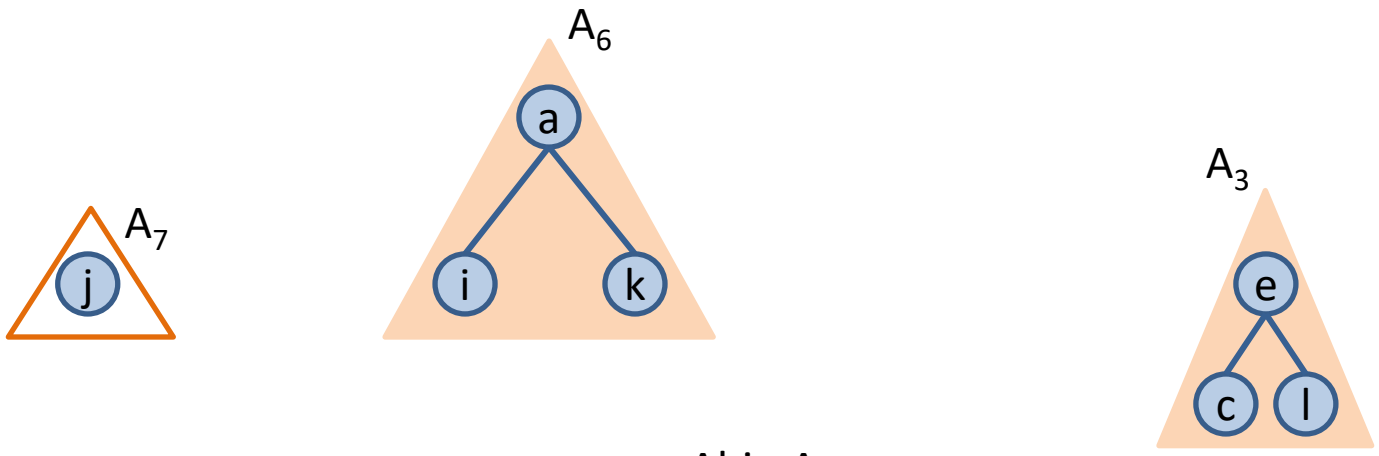
Construcción de un árbol binario (I)





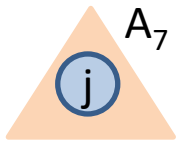
Abin A_6; A_6.insertarRaiz('a');
A_6.insertarIzqdo(A_4);
A_6.insertarDrcho(A_5);

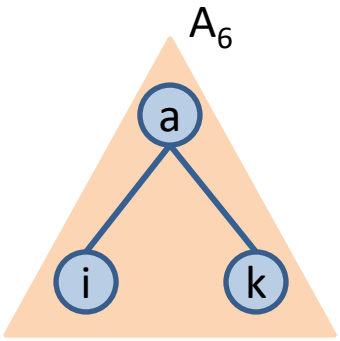
Construcción de un árbol binario (I)

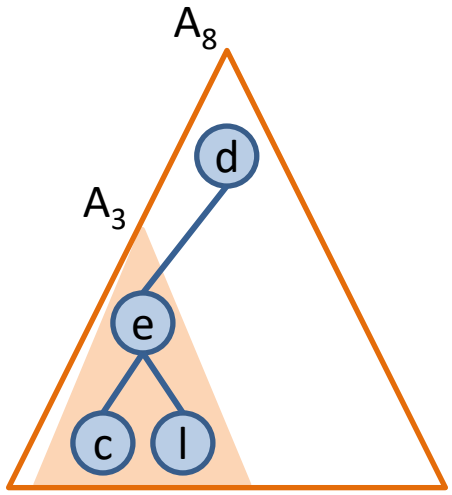


Abin A_7;
A_7.insertarRaiz('j');

Construcción de un árbol binario (I)

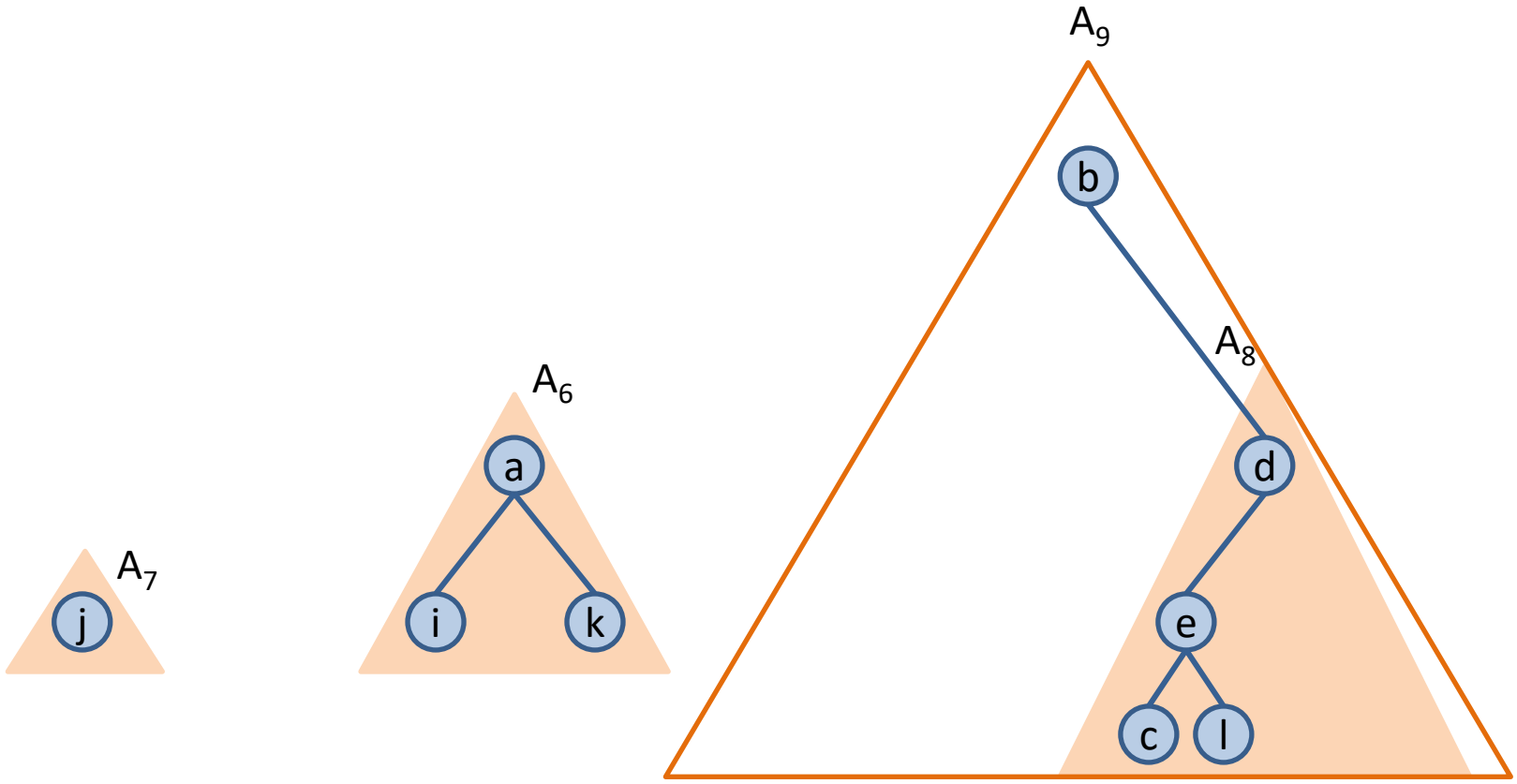






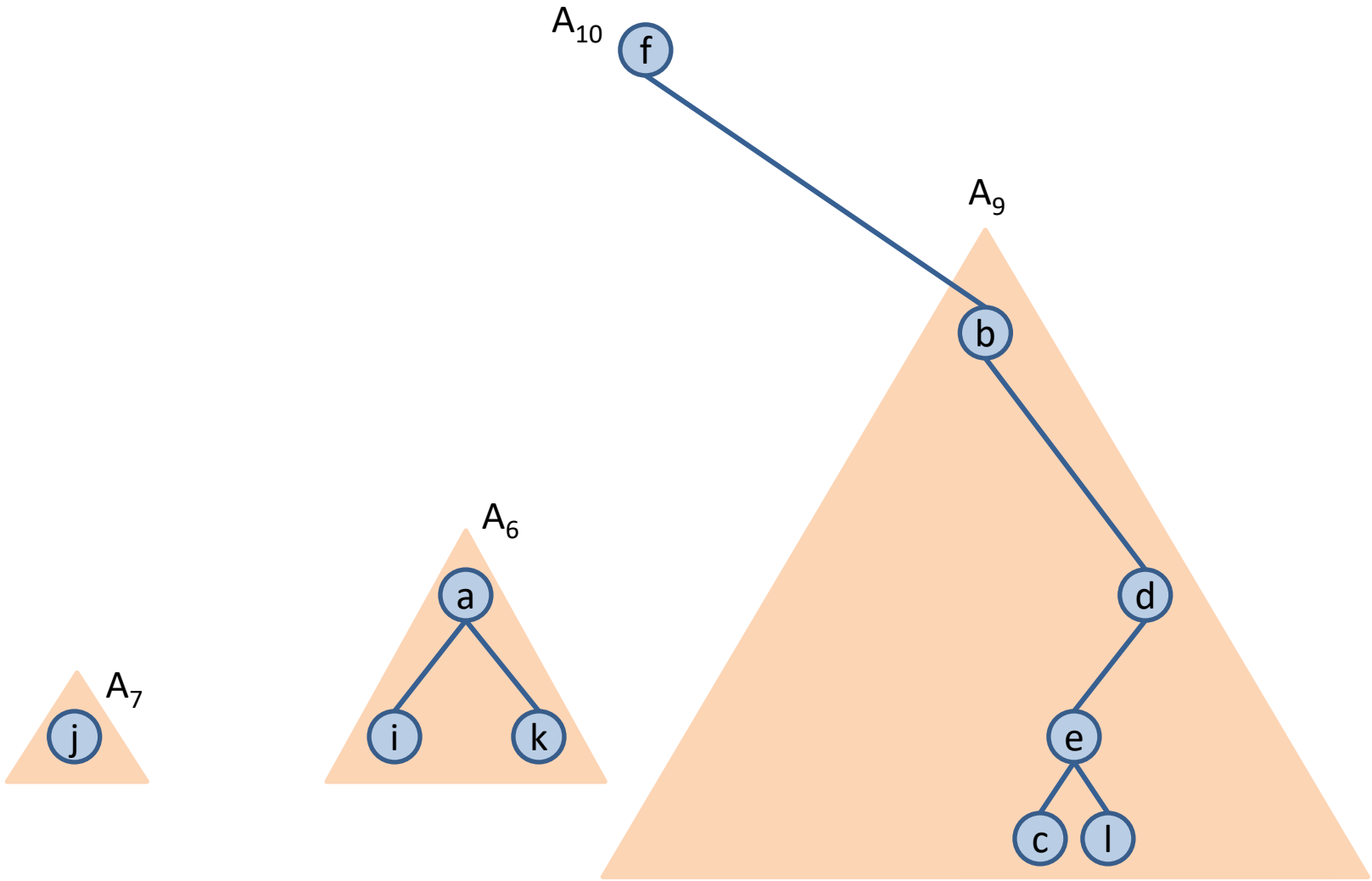
Abin A_8;
A_8.insertarRaiz('d');
A_8.insertarIzqdo(A_3);

Construcción de un árbol binario (I)



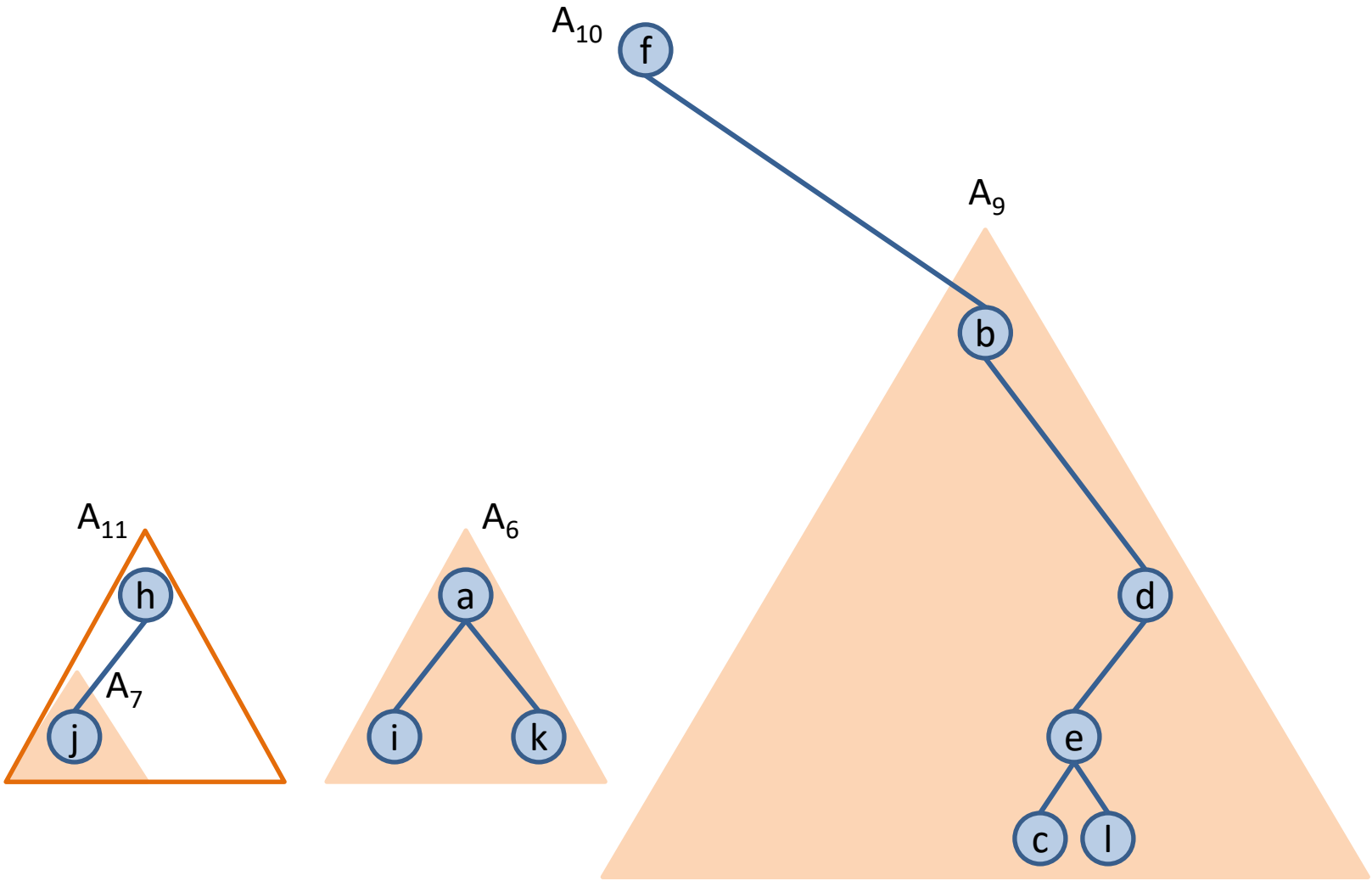
Abin A_9; A_9.insertarRaiz('b');
A_9.insertarDrcho(A_8);

Construcción de un árbol binario (I)



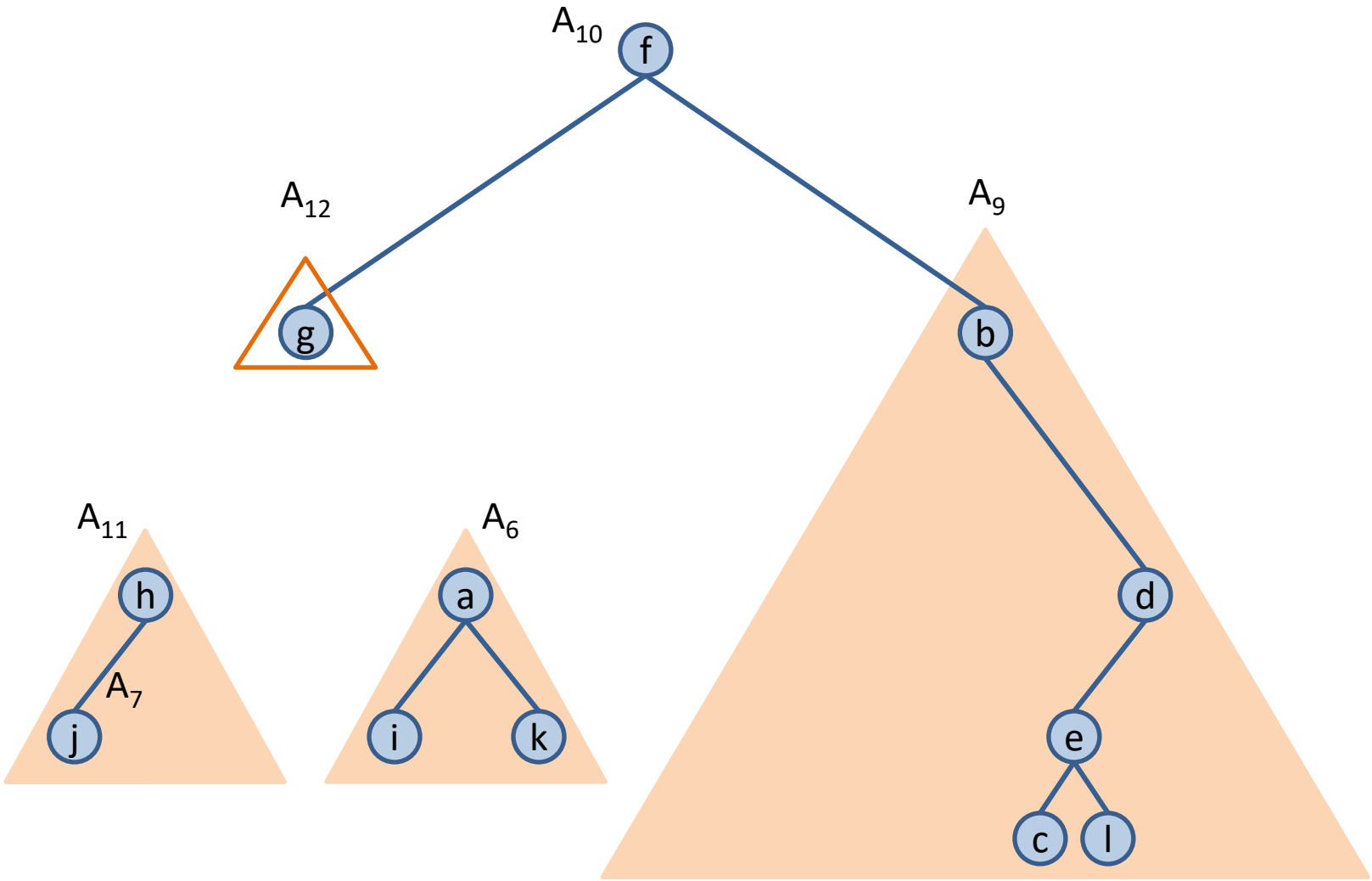
Abin A_{10}; A_{10}.insertarRaiz('f');
A_{10}.insertarDrcho(A_9);

Construcción de un árbol binario (I)



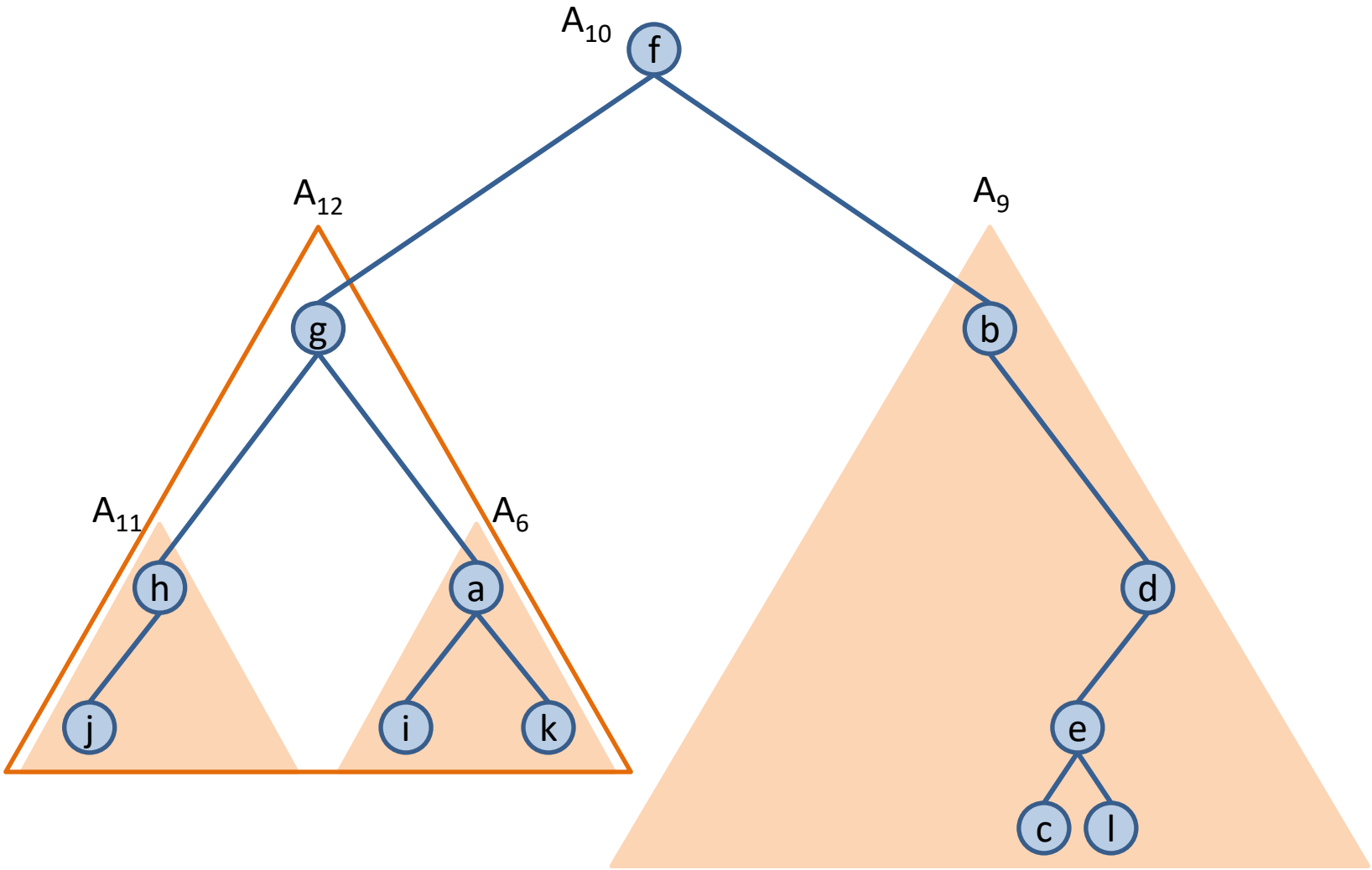
Abin A_{11}; A_{11}.insertarRaiz('h');
A_{11}.insertarIzqdo(A_7);

Construcción de un árbol binario (I)



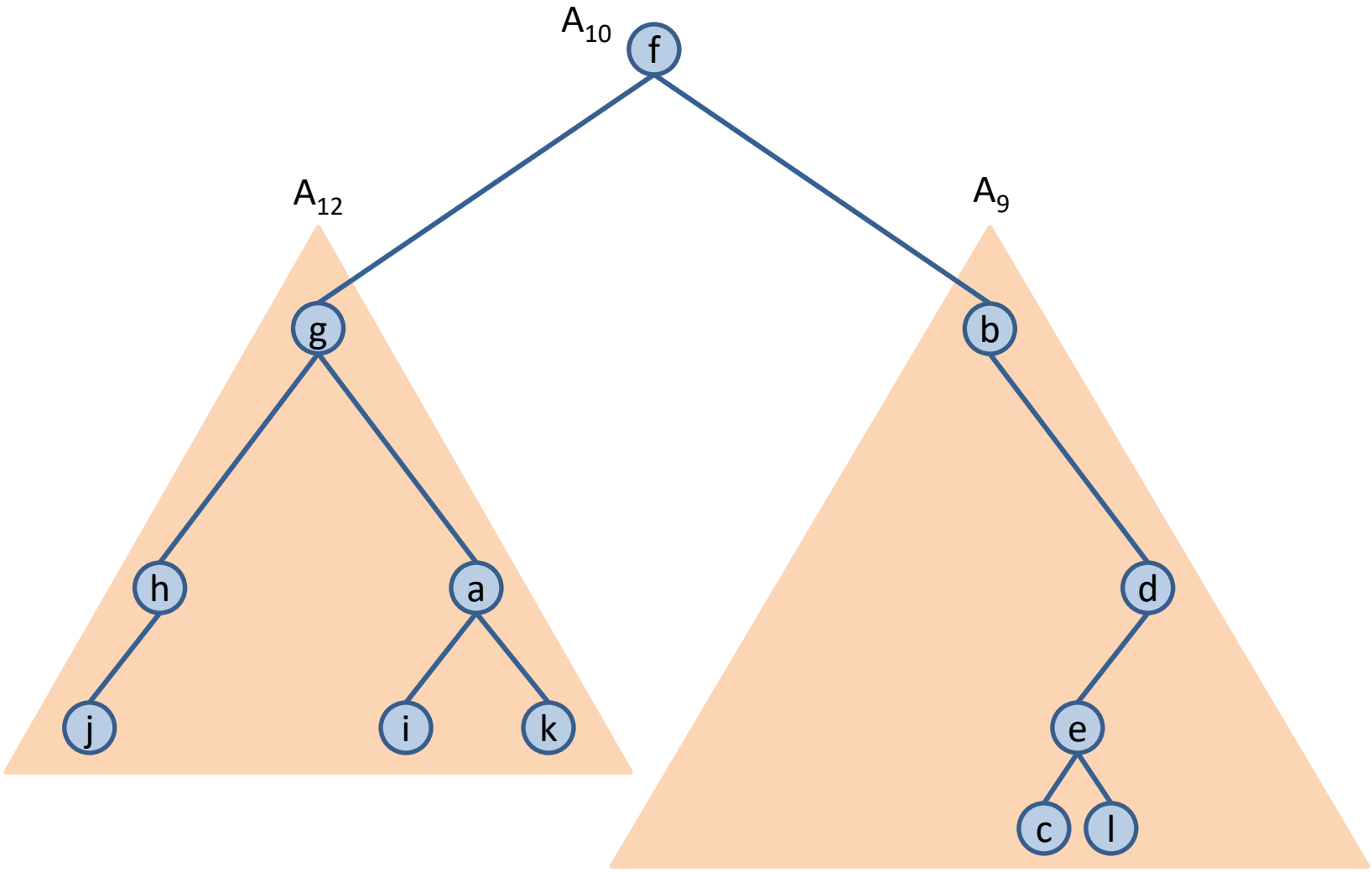
Abin A_{12}; A_{12}.insertarRaiz('g');
A_{10}.insertarIzqdo(A_{12});

Construcción de un árbol binario (I)

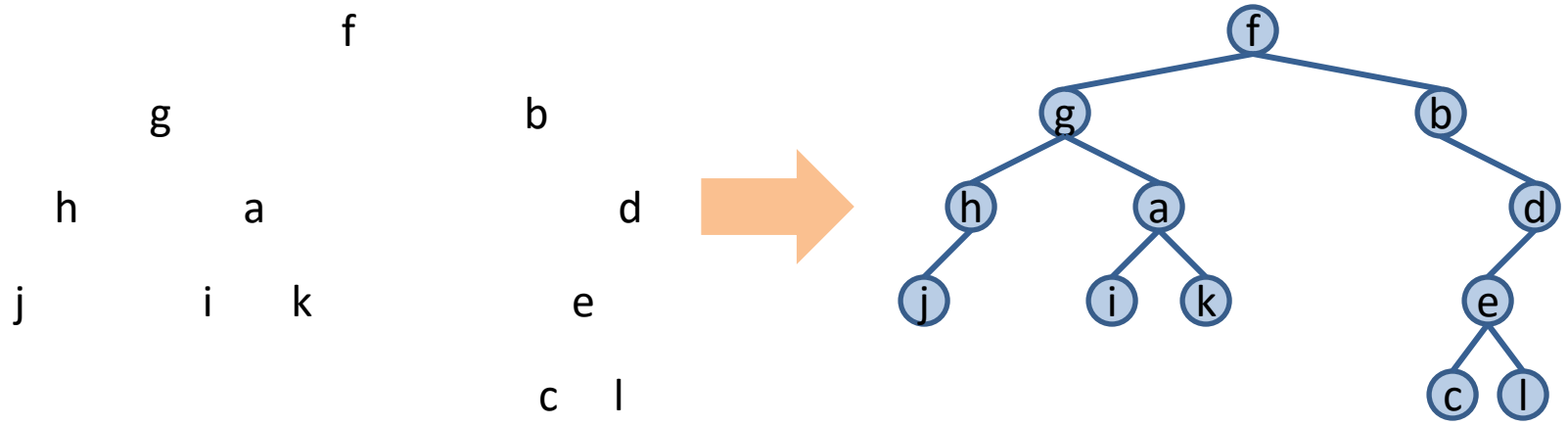


A_{12}.insertarIzqdo(A_{11});
A_{12}.insertarDrcho(A_6);

Construcción de un árbol binario (I)



Construcción de un árbol binario (II)

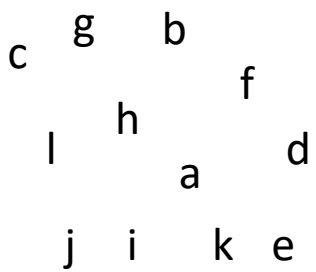


Construcción añadiendo los nodos uno a uno desde la raíz hacia las hojas:

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)

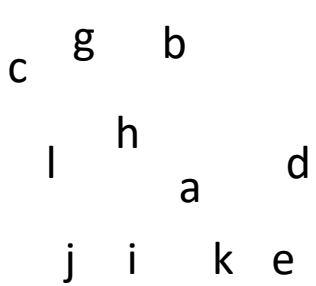
Creación del árbol binario A como un contenedor vacío.

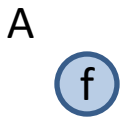


A

Abin A;

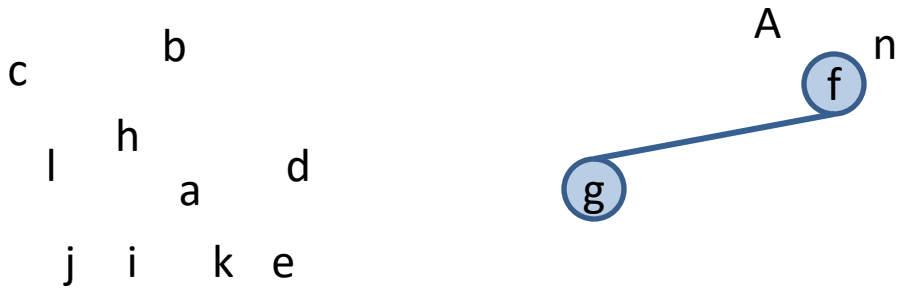
Construcción de un árbol binario (II)





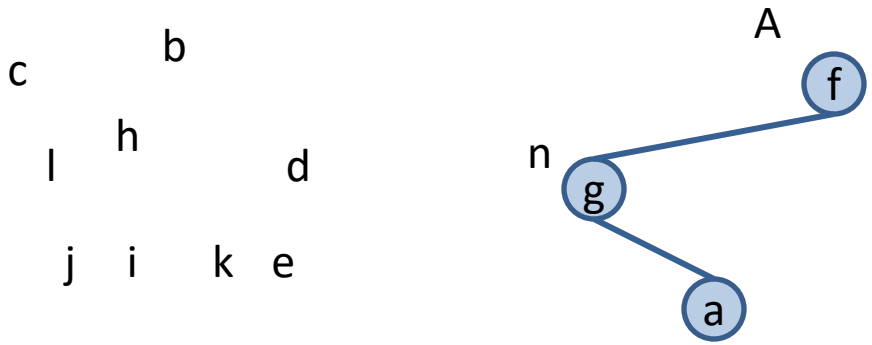
A.insertarRaiz('f');

Construcción de un árbol binario (II)



A.insertarHijoIzqdo(n, ‘g’);

Construcción de un árbol binario (II)



A.insertarHijoDrcho(n, ‘a’);

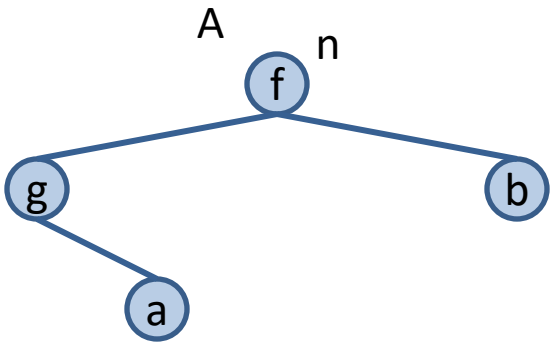
Construcción de un árbol binario (II)

c

h

l d

j i k e



A.insertarHijoDrcho(n, ‘b’);

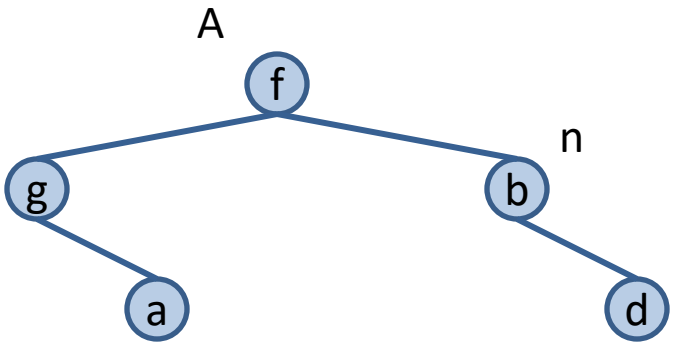
Construcción de un árbol binario (II)

c

h

l

j i k e



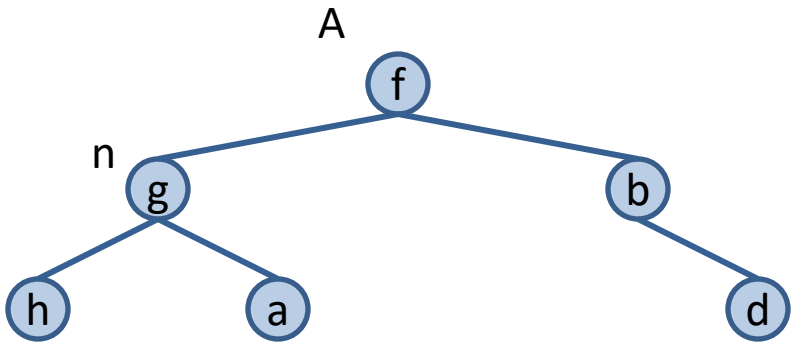
A.insertarHijoDrcho(n, ‘d’);

Construcción de un árbol binario (II)

c

l

j i k e



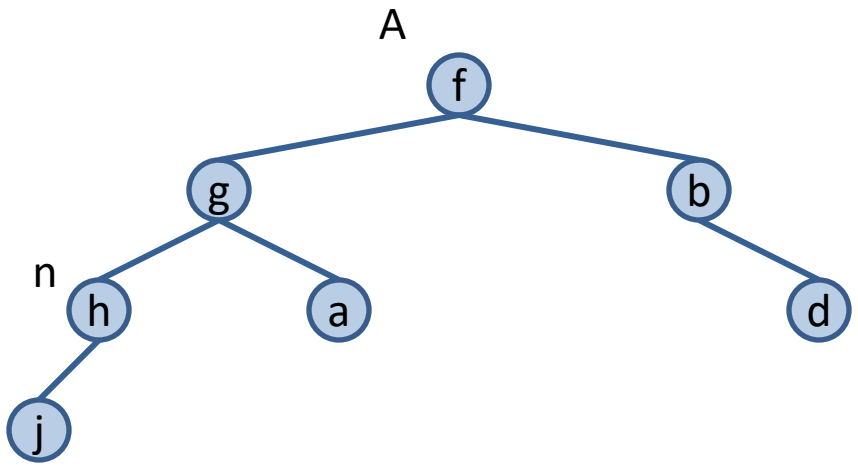
A.insertarHijoIzqdo(n, ‘h’);

Construcción de un árbol binario (II)

c

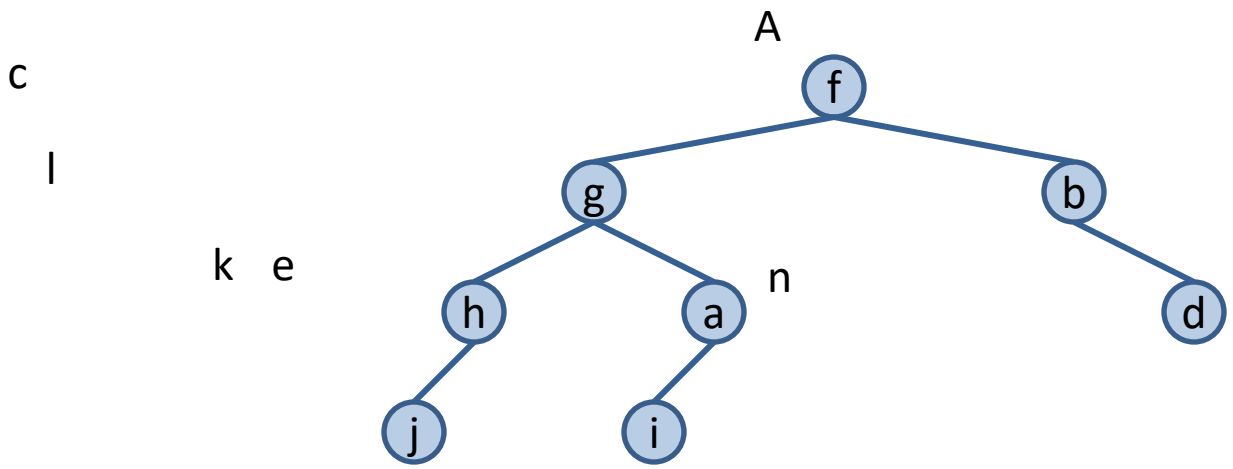
l

i k e



A.insertarHijoIzqdo(n, ‘j’);

Construcción de un árbol binario (II)



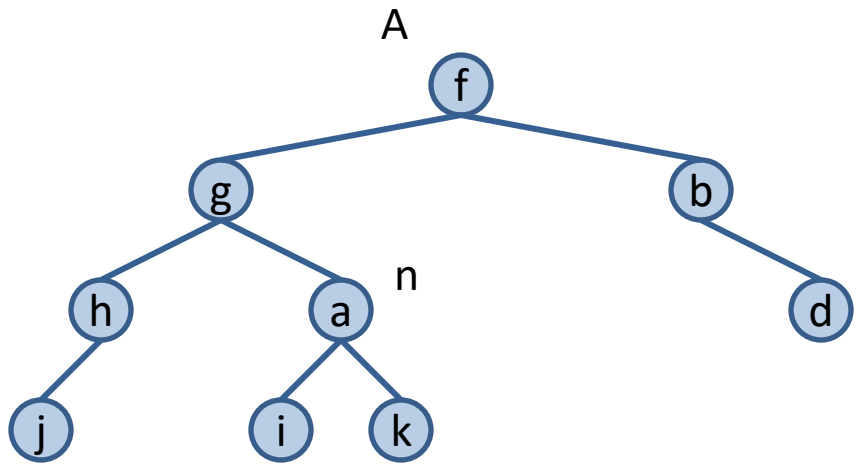
A.insertarHijoIzqdo(n, ‘i’);

Construcción de un árbol binario (II)

c

l

e

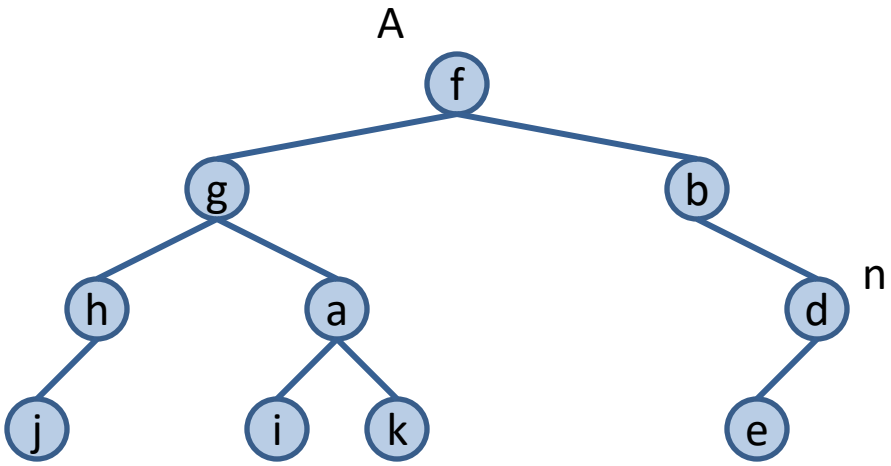


A.insertarHijoDrcho(n, ‘k’);

Construcción de un árbol binario (II)

c

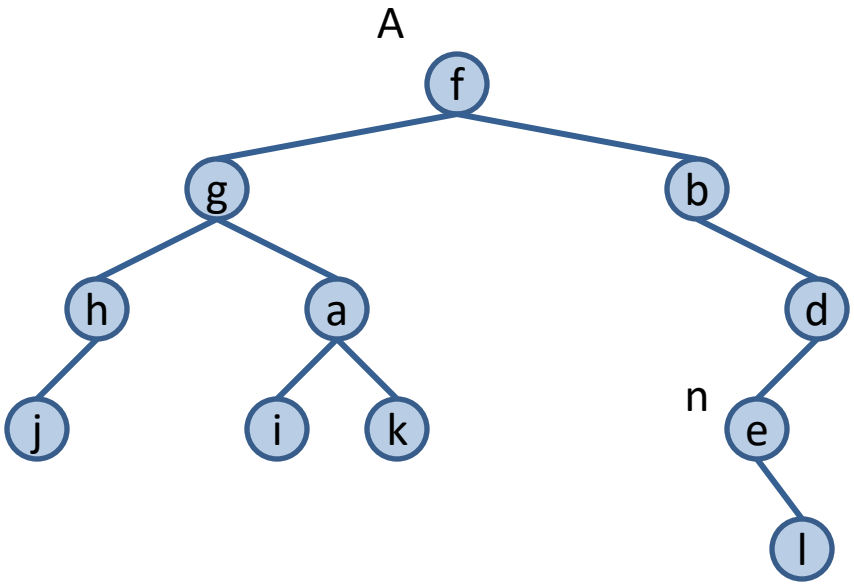
l



A.insertarHijoIzqdo(n, ‘e’);

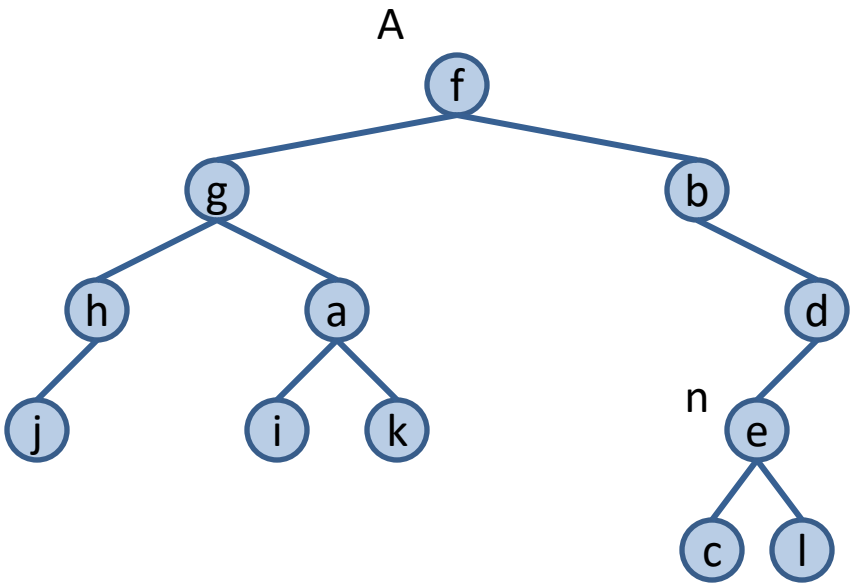
Construcción de un árbol binario (II)

c



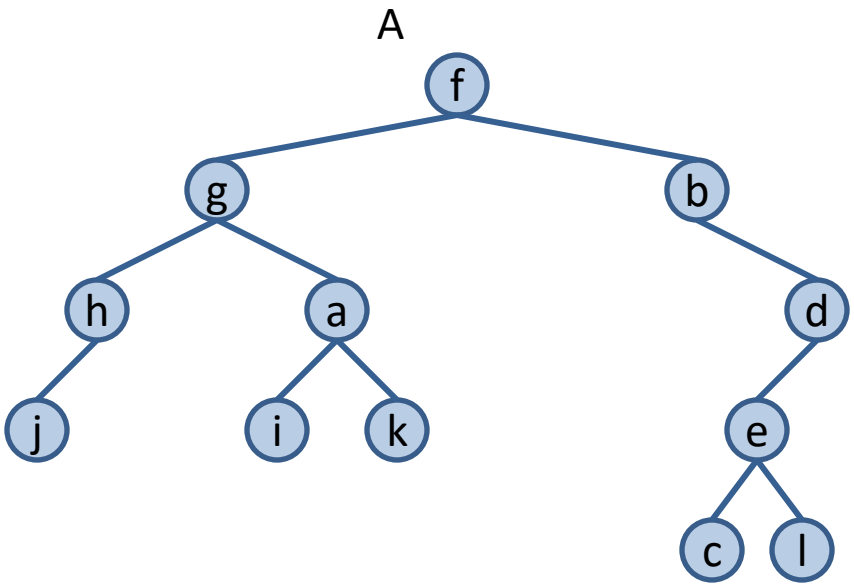
A.insertarHijoDrcho(n, ‘l’);

Construcción de un árbol binario (II)



A.insertarHijoIzqdo(n, ‘c’);

Construcción de un árbol binario (II)



Especificación de operaciones

Abin ()

Post: Crea 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 *hijoIzqdo(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 *hijoDrcho(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

bool vacio () const

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

size_t tama () const

Post: Devuelve el número de elementos que contiene el árbol.

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.

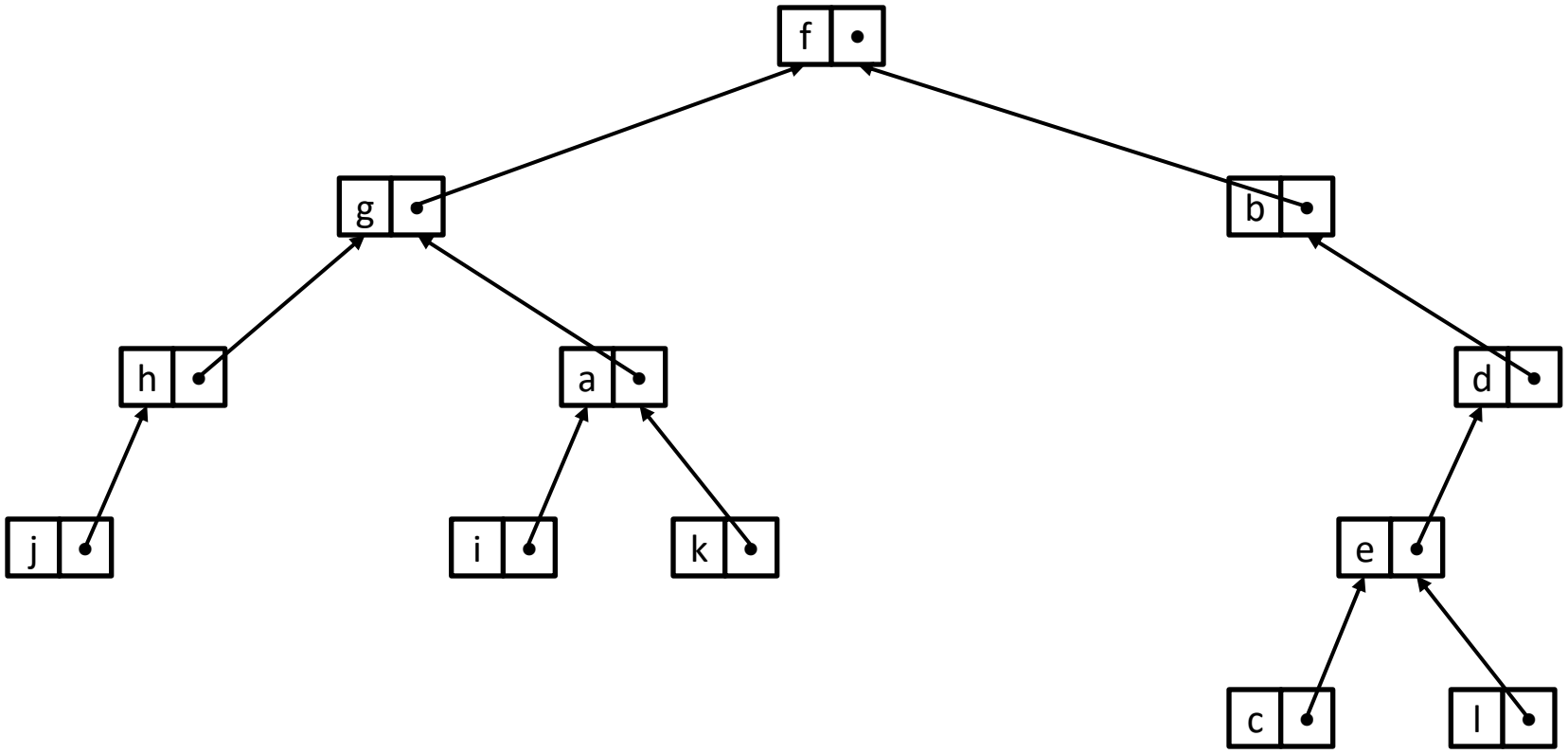
Post: Devuelve el nodo hijo izquierdo del nodo *n*. Si no existe, devuelve *NODO_NULO*.

nodo hijoDrcho (nodo n) const

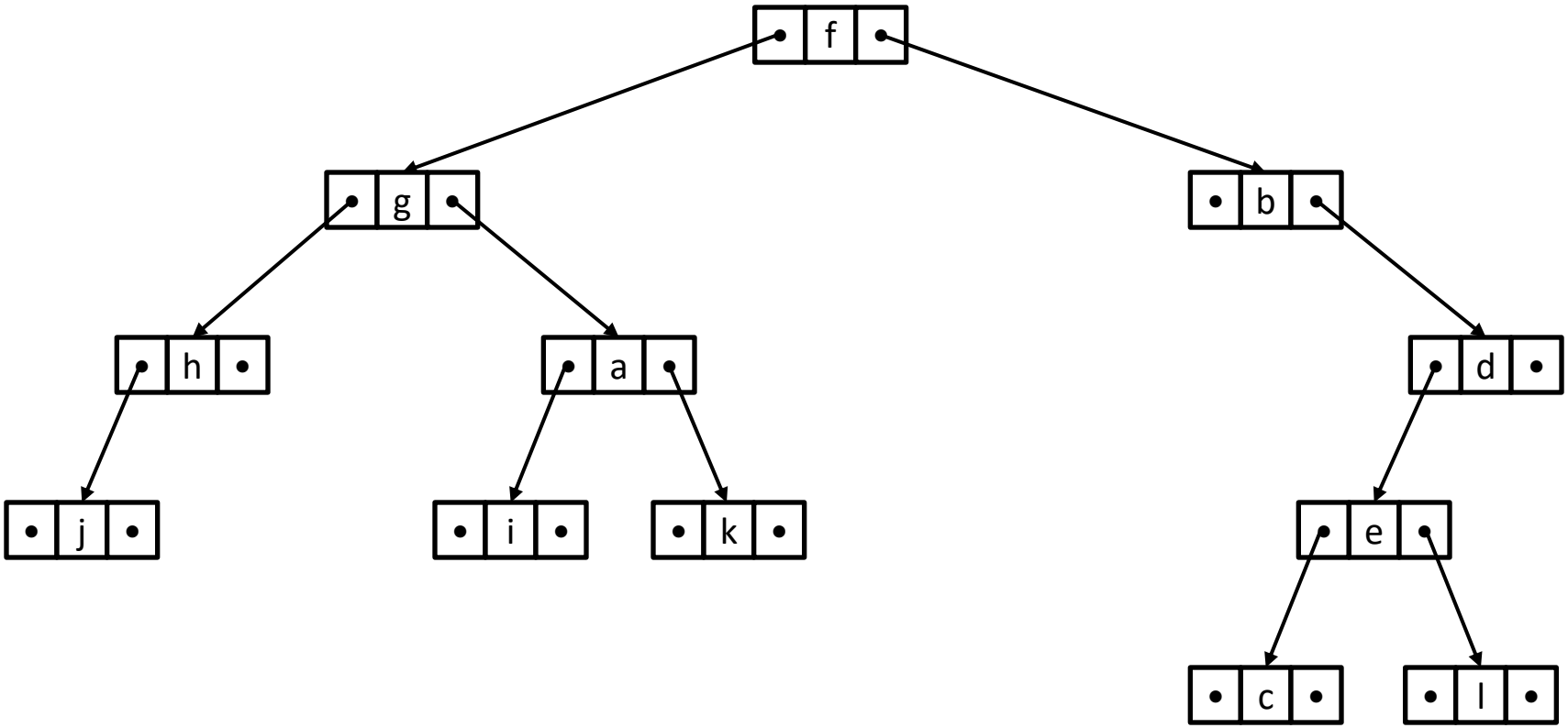
Pre: *n* es un nodo del árbol.

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

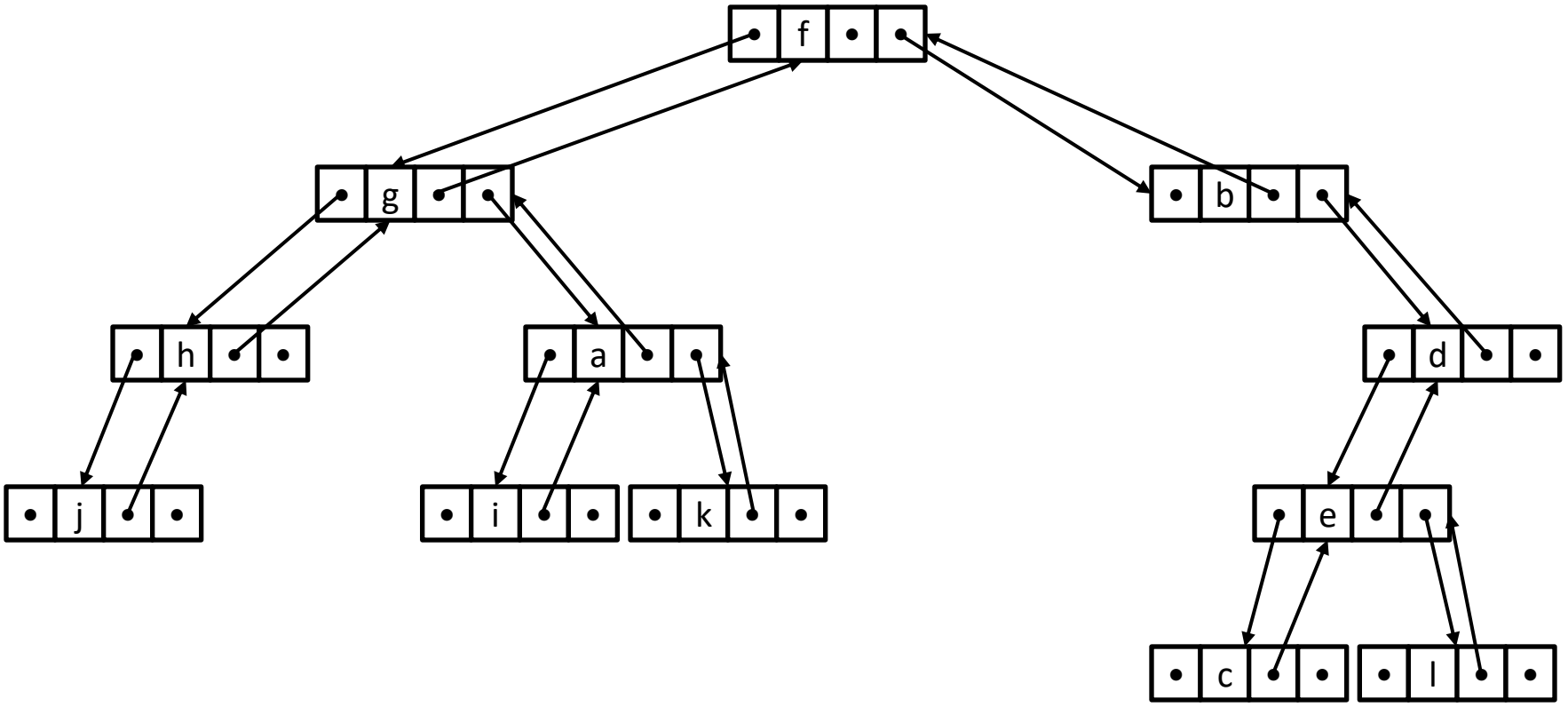
Implementación de un árbol binario usando celdas enlazadas



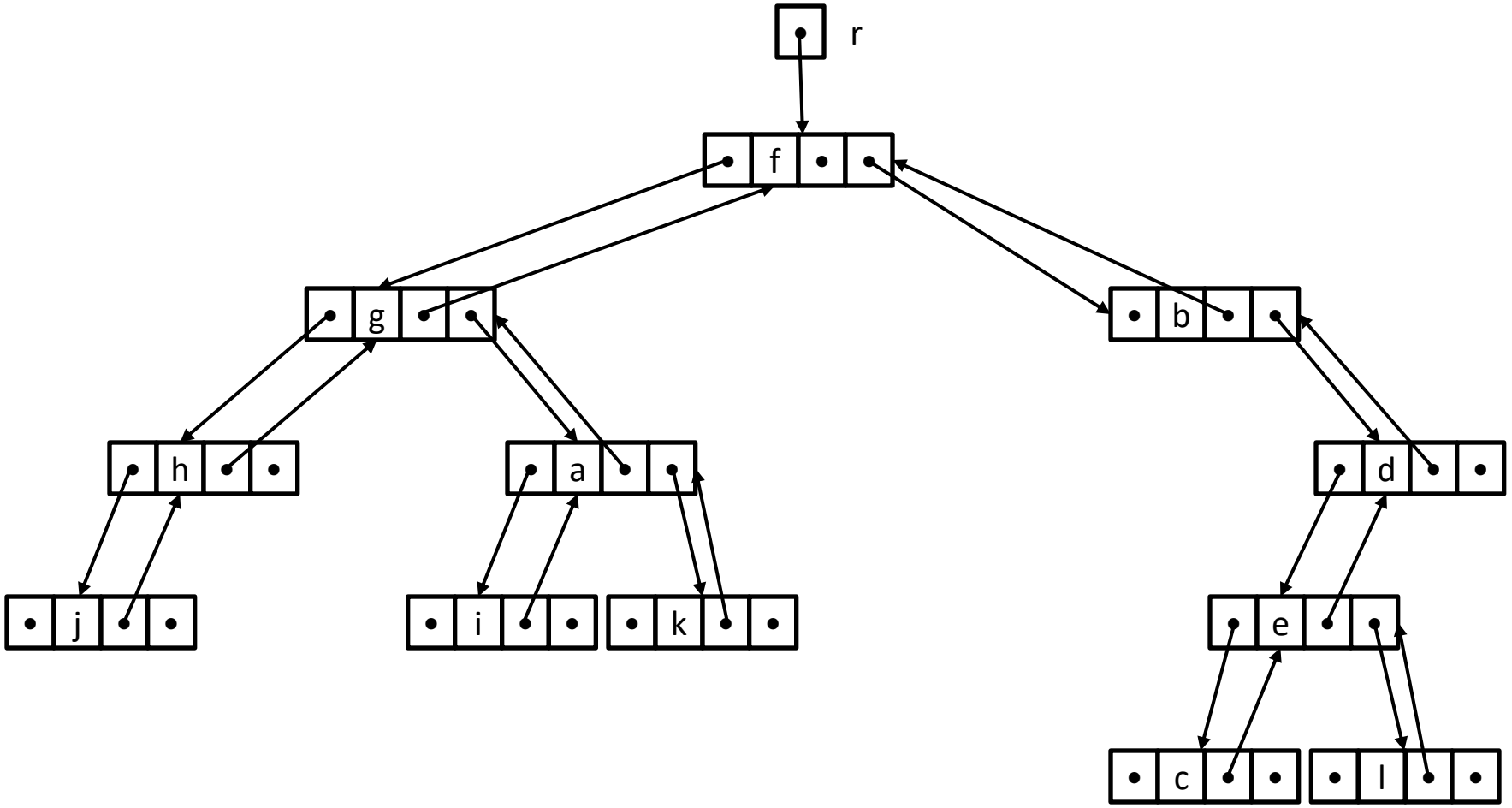
Implementación de un árbol binario usando celdas enlazadas



Implementación de un árbol binario usando celdas enlazadas



Implementación de un árbol binario usando celdas enlazadas



```
#ifndef ABIN_H
#define ABIN_H
#include <cassert>
#include <cstddef>   // size_t
#include <utility>   // swap

template <typename T> class Abin {
   struct celda;   // Declaración adelantada privada
public:
   typedef celda* nodo;
   static const nodo NODO_NULO;

   Abin();
   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 vacio() const;
size_t tama() 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& A);                  // Ctor. de copia
Abin& operator =(const Abin& A);   // Asig. de árboles
~Abin();                              // Destructor
```

```
private:
   struct celda {
      T elto;
      nodo padre, hizq, hder;
   };

   nodo r;           // Raíz del árbol
   size_t numNodos; // Tamaño del árbol

   nodo copiar(nodo n);
   void destruir(nodo& n);
}; // class Abin

// Definición del nodo nulo
template <typename T>
const typename Abin<T>::nodo Abin<T>::NODO_NULO{nullptr};
```

```
template <typename T>
inline Abin<T>::Abin() : r{NODO_NULO}, numNodos{0} {}

template <typename T>
inline void Abin<T>::insertarRaiz(const T& e)
{
   assert(vacio());
   r = new celda{e};
   numNodos = 1;
}

template <typename T>
inline void Abin<T>::insertarHijoIzqdo(nodo n, const T& e)
{
   assert(n != NODO_NULO);
   assert(n->hizq == NODO_NULO);
   n->hizq = new celda{e, n};
   ++numNodos;
}
```

```
template <typename T>
inline void Abin<T>::insertarHijoDrcho(nodo n, const T& e)
{
   assert(n != NODO_NULO);
   assert(n->hder == NODO_NULO);
   n->hder = new celda{e, n};
   ++numNodos;
}

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 &&   // y es
          n->hizq->hder == NODO_NULO);    // hoja.
   delete n->hizq;
   n->hizq = NODO_NULO;
   --numNodos;
}
```

```
template <typename T>
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 &&   // y es
          n->hder->hder == NODO_NULO);    // hoja
   delete n->hder;
   n->hder = NODO_NULO;
   --numNodos;
}
template <typename T>
inline void Abin<T>::eliminarRaiz()
{
   assert(numNodos == 1);
   delete r;
   r = NODO_NULO;
   numNodos = 0;
}
```

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

template <typename T>
inline size_t Abin<T>::tama() const
{ return numNodos; }

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;
}
```

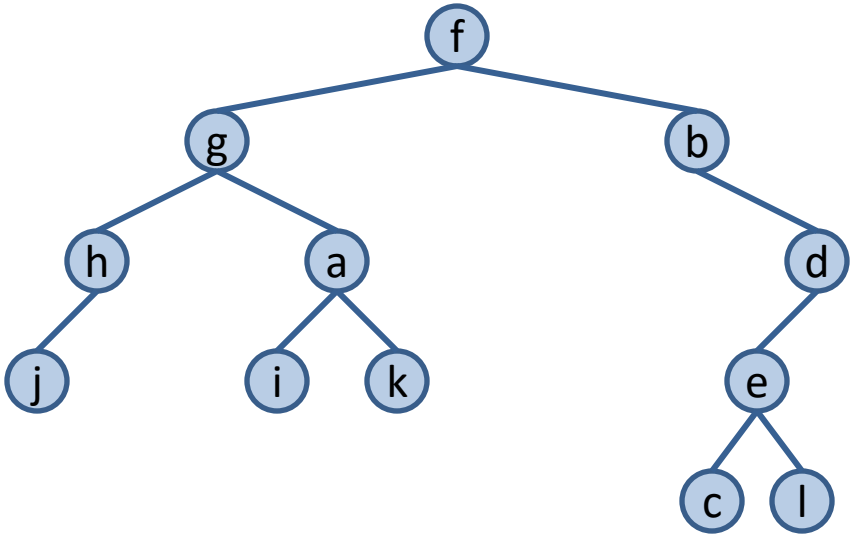
```
template <typename T> inline
typename Abin<T>::nodo Abin<T>::hijoIzqdo(nodo n) const
{
   assert(n != NODO_NULO);
   return n->hizq;
}

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& A) : Abin{}
{
   r = copiar(A.r); // Copiar raíz y descendientes.
   numNodos = A.numNodos;
}

template <typename T>
inline Abin<T>& Abin<T>::operator =(const Abin& A)
{
   Abin B{A};
   std::swap(r, B.r);
   std::swap(numNodos, B.numNodos);
   return *this;
}
```

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

/*--------------------------------------------------------*/
/* Métodos privados                                       */
/*--------------------------------------------------------*/
// Destruye un nodo y todos sus descendientes
template <typename T>
void Abin<T>::destruir(nodo& n)
{
   if (n != NODO_NULO)
   {
      destruir(n->hizq);
      destruir(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(nodo n)
{
   nodo m = NODO_NULO;
   if (n != NODO_NULO) {
      Abin A; // Contiene los nodos copiados.
              // Si la copia no se completa, A es destruido.
      A.r = new celda{n->elto};    // Copiar n en raíz.
      A.r->hizq = copiar(n->hizq); // Copiar subárbol izqdo.
      if (A.r->hizq != NODO_NULO) A.r->hizq->padre = A.r;
      A.r->hder = copiar(n->hder); // Copiar subárbol drcho.
      if (A.r->hder != NODO_NULO) A.r->hder->padre = A.r;
      m = A.r;
      A.r = NODO_NULO; // Evita destruir la copia.
   }
   return m;
}

#endif // ABIN_H
```

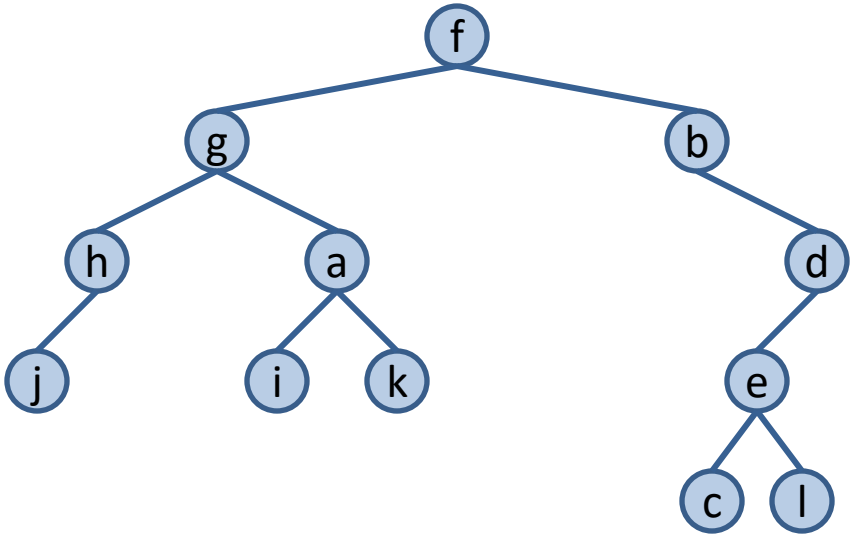
Implementación vectorial de árboles binarios



	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

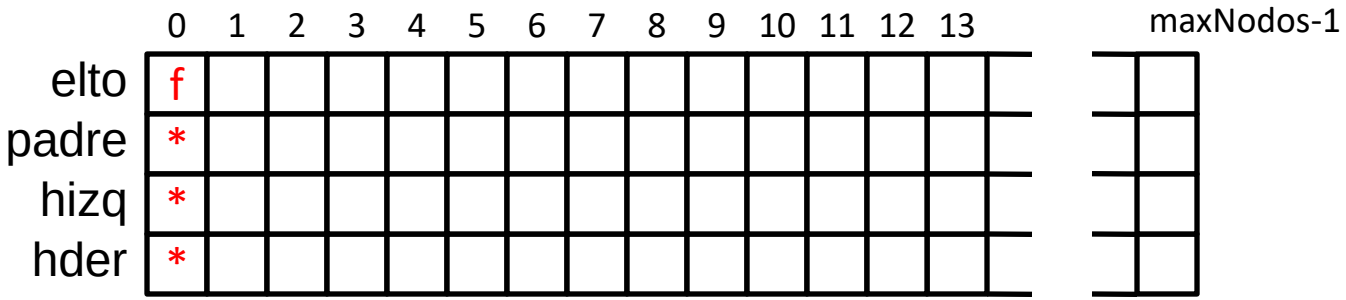


	0	1	2	3	4	5	6	7	8	9	10	11	12	13
elto	f	g	a	b	d	h	j	i	k	e	l	c		
padre	*	0	1	0	3	1	5	2	2	4	9	9		
hizq	1	5	7	*	9	6	*	*	*	11	*	*		
hder	3	2	8	4	*	*	*	*	*	10	*	*		

maxNodos-1

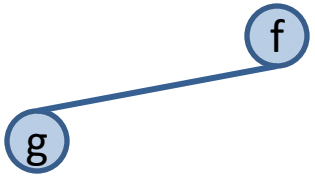
Implementación vectorial de árboles binarios

Inserción y eliminación



Implementación vectorial de árboles binarios

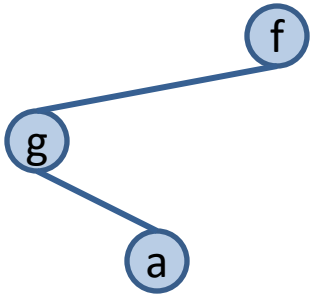
Inserción y eliminación



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

Implementación vectorial de árboles binarios

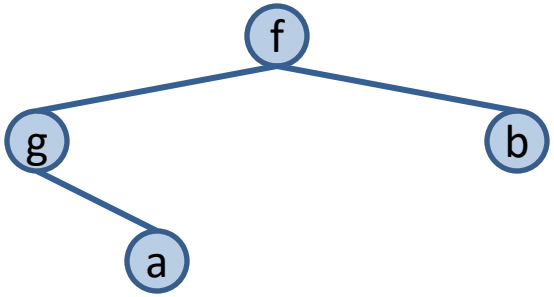
Inserción y eliminación



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

Implementación vectorial de árboles binarios

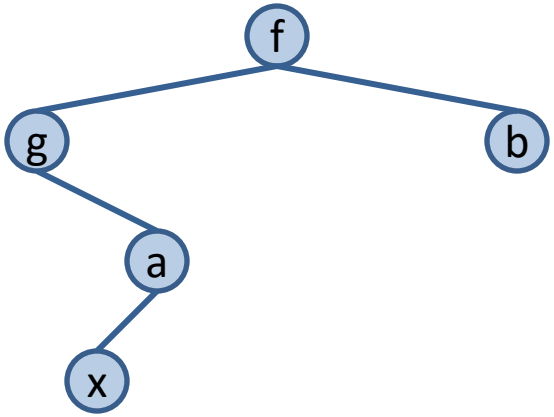
Inserción y eliminación



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

Implementación vectorial de árboles binarios

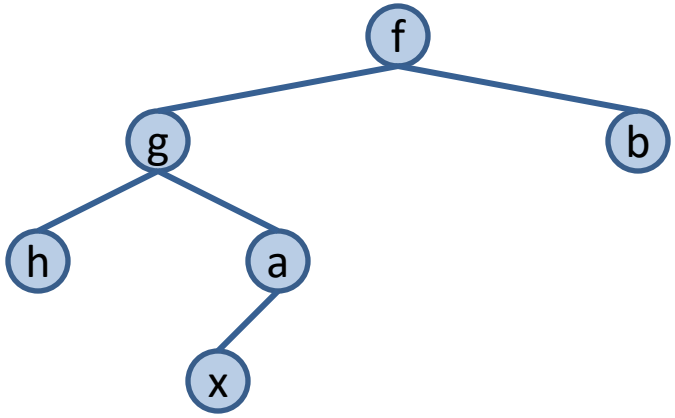
Inserción y eliminación



	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

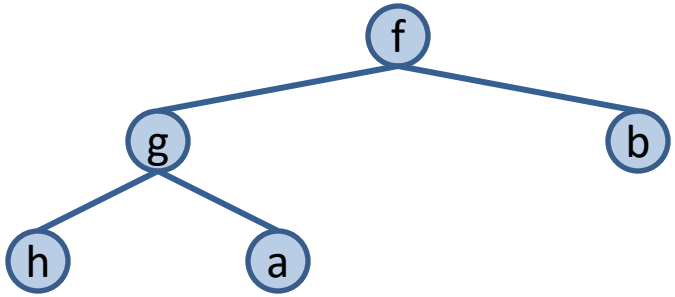


	0	1	2	3	4	5	6	7	8	9	10	11	12	13
elto	f	g	a	b	x	h								
padre	*	0	1	0	2	1								
hizq	1	5	4	*	*	*								
hder	3	2	*	*	*	*								

maxNodos-1

Implementación vectorial de árboles binarios

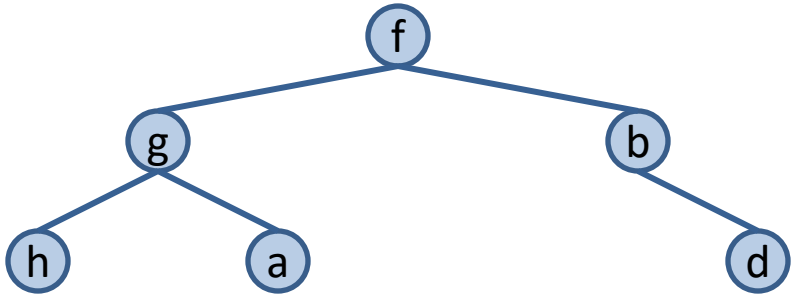
Inserción y eliminación



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

Implementación vectorial de árboles binarios

Inserción y eliminación

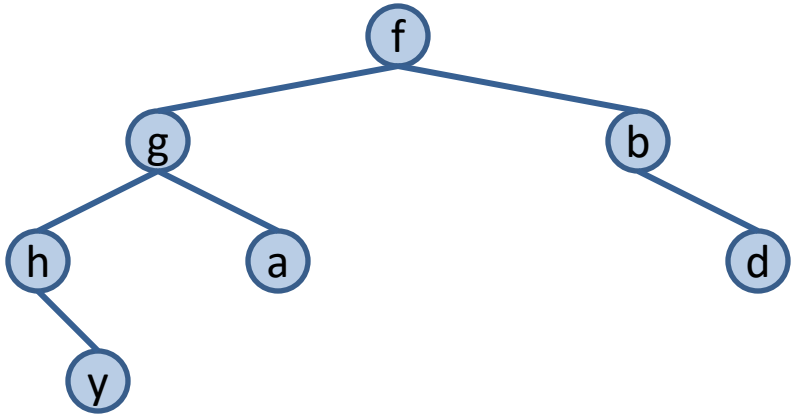


	0	1	2	3	4	5	6	7	8	9	10	11	12	13
elto	f	g	a	b	d	h								
padre	*	0	1	0	3	1								
hizq	1	5	*	*	*	*								
hder	3	2	*	4	*	*								

maxNodos-1

Implementación vectorial de árboles binarios

Inserción y eliminación

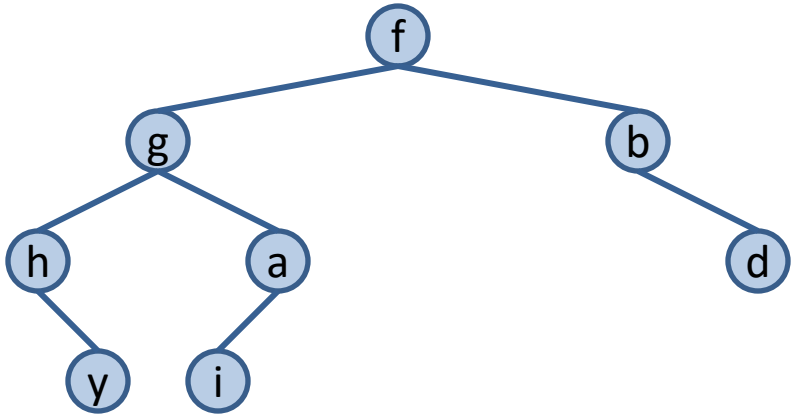


	0	1	2	3	4	5	6	7	8	9	10	11	12	13
elto	f	g	a	b	d	h	y							
padre	*	0	1	0	3	1	5							
hizq	1	5	*	*	*	*	*							
hder	3	2	*	4	*	6	*							

maxNodos-1

Implementación vectorial de árboles binarios

Inserción y eliminación

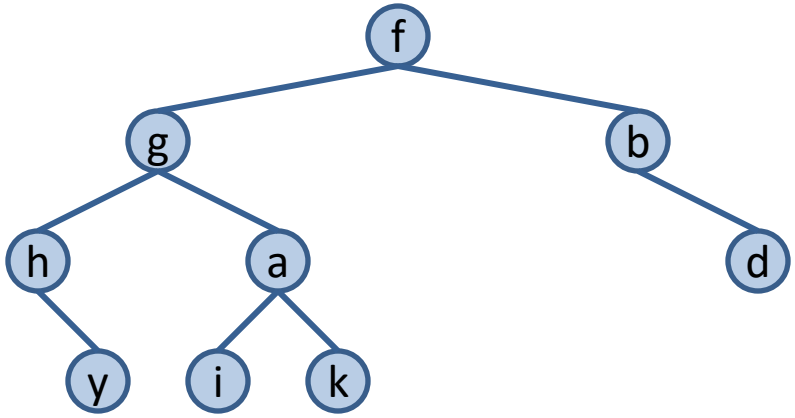


	0	1	2	3	4	5	6	7	8	9	10	11	12	13
elto	f	g	a	b	d	h	y	i						
padre	*	0	1	0	3	1	5	2						
hizq	1	5	7	*	*	*	*	*						
hder	3	2	*	4	*	6	*	*						

maxNodos-1

Implementación vectorial de árboles binarios

Inserción y eliminación

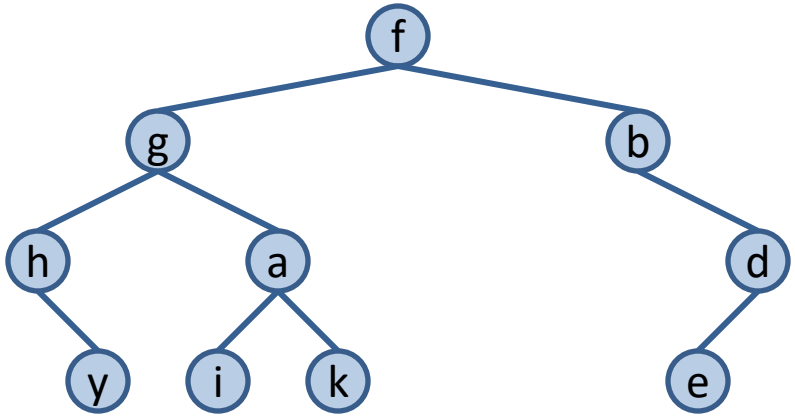


	0	1	2	3	4	5	6	7	8	9	10	11	12	13
elto	f	g	a	b	d	h	y	i	k					
padre	*	0	1	0	3	1	5	2	2					
hizq	1	5	7	*	*	*	*	*	*					
hder	3	2	8	4	*	6	*	*	*					

maxNodos-1

Implementación vectorial de árboles binarios

Inserción y eliminación

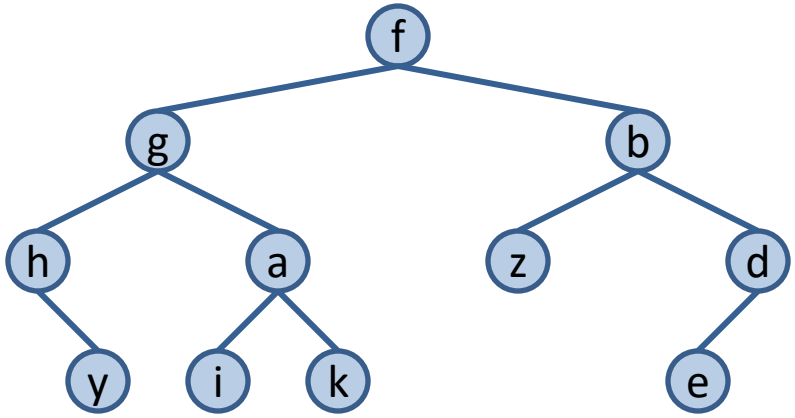


	0	1	2	3	4	5	6	7	8	9	10	11	12	13
elto	f	g	a	b	d	h	y	i	k	e				
padre	*	0	1	0	3	1	5	2	2	4				
hizq	1	5	7	*	9	*	*	*	*	*				
hder	3	2	8	4	*	6	*	*	*	*				

maxNodos-1

Implementación vectorial de árboles binarios

Inserción y eliminación

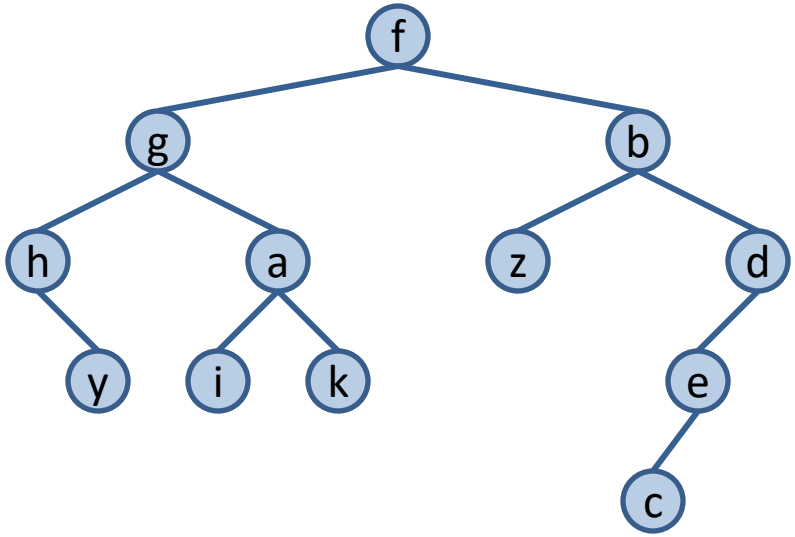


	0	1	2	3	4	5	6	7	8	9	10	11	12	13
elto	f	g	a	b	d	h	y	i	k	e	z			
padre	*	0	1	0	3	1	5	2	2	4	3			
hizq	1	5	7	10	9	*	*	*	*	*	*			
hder	3	2	8	4	*	6	*	*	*	*	*			

maxNodos-1

Implementación vectorial de árboles binarios

Inserción y eliminación

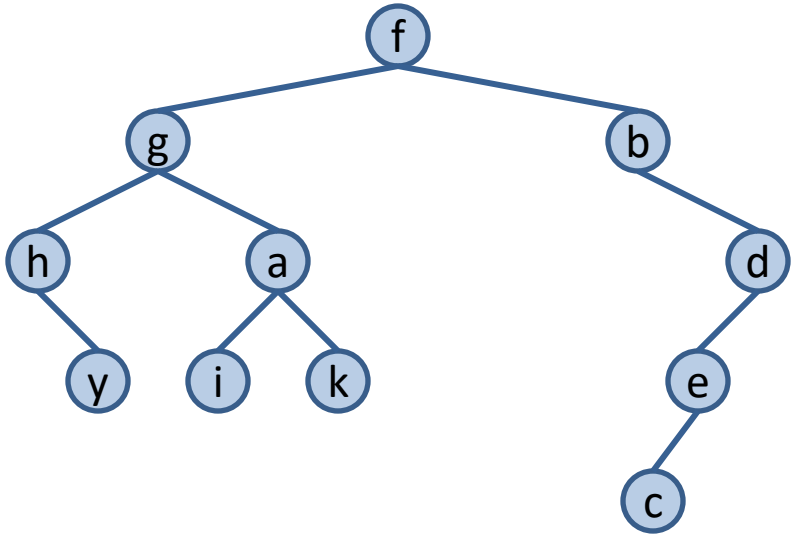


	0	1	2	3	4	5	6	7	8	9	10	11	12	13
elto	f	g	a	b	d	h	y	i	k	e	z	c		
padre	*	0	1	0	3	1	5	2	2	4	3	9		
hizq	1	5	7	10	9	*	*	*	*	11	*	*		
hder	3	2	8	4	*	6	*	*	*	*	*	*		

maxNodos-1

Implementación vectorial de árboles binarios

Inserción y eliminación

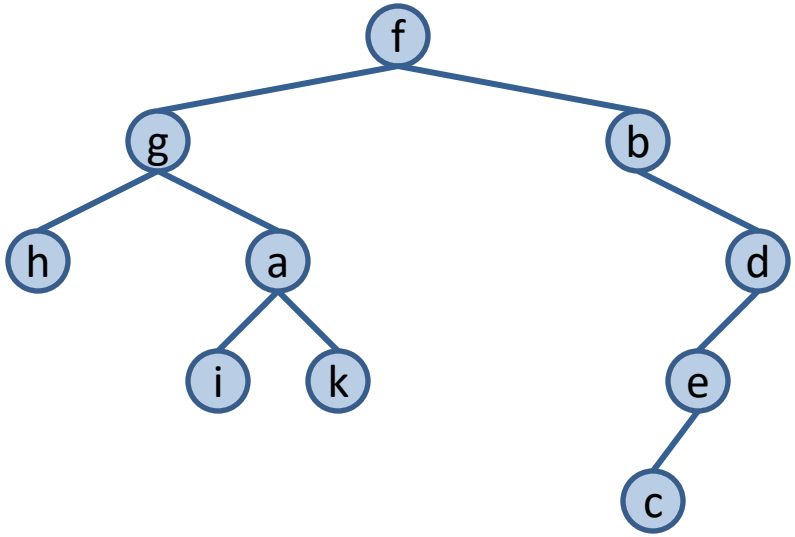


	0	1	2	3	4	5	6	7	8	9	10	11	12	13
elto	f	g	a	b	d	h	y	i	k	e		c		
padre	*	0	1	0	3	1	5	2	2	4		9		
hizq	1	5	7	*	9	*	*	*	*	11		*		
hder	3	2	8	4	*	6	*	*	*	*		*		

maxNodos-1

Implementación vectorial de árboles binarios

Inserción y eliminación

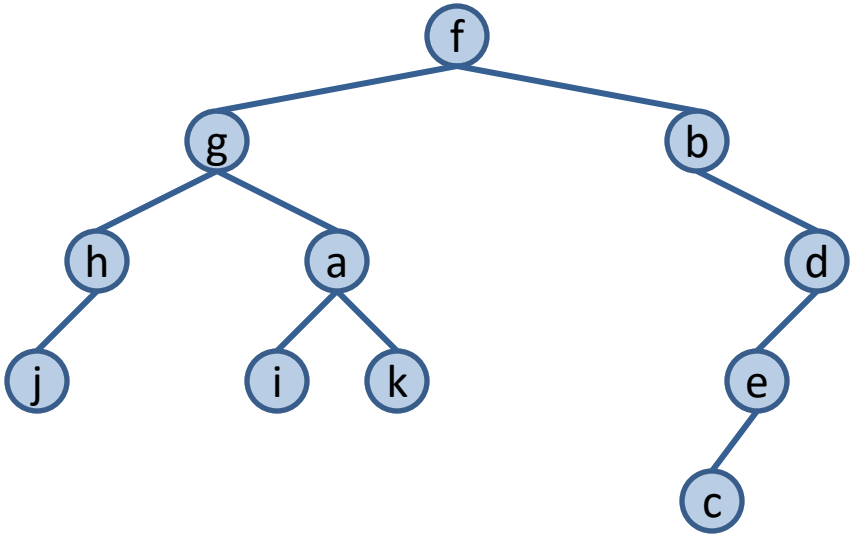


	0	1	2	3	4	5	6	7	8	9	10	11	12	13
elto	f	g	a	b	d	h		i	k	e		c		
padre	*	0	1	0	3	1		2	2	4		9		
hizq	1	5	7	*	9	*		*	*	11		*		
hder	3	2	8	4	*	*		*	*	*		*		

maxNodos-1

Implementación vectorial de árboles binarios

Inserción y eliminación

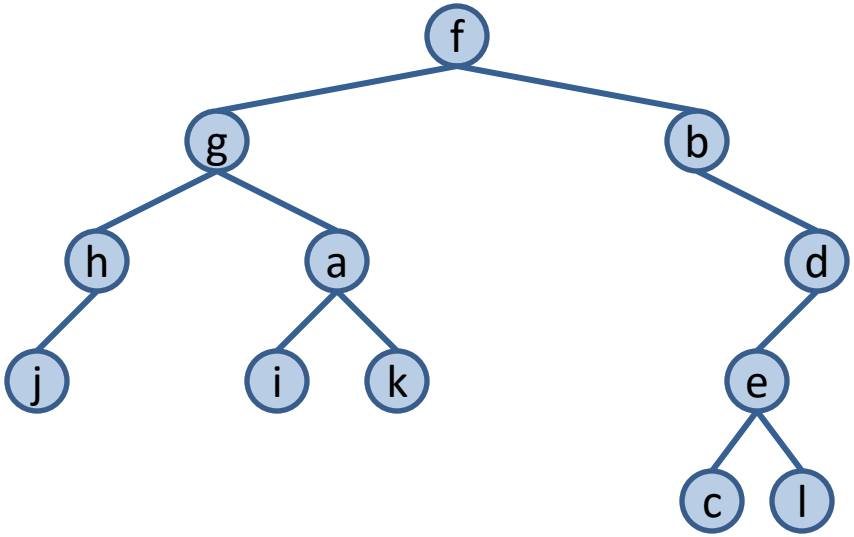


	0	1	2	3	4	5	6	7	8	9	10	11	12	13
elto	f	g	a	b	d	h	j	i	k	e		c		
padre	*	0	1	0	3	1	5	2	2	4		9		
hizq	1	5	7	*	9	6	*	*	*	11		*		
hder	3	2	8	4	*	*	*	*	*	*		*		

maxNodos-1

Implementación vectorial de árboles binarios

Inserción y eliminación



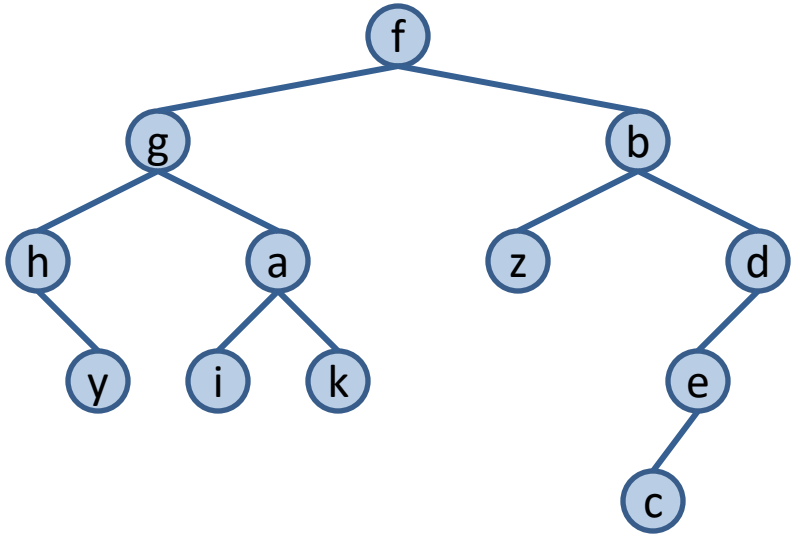
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
elto	f	g	a	b	d	h	j	i	k	e	l	c		
padre	*	0	1	0	3	1	5	2	2	4	9	9		
hizq	1	5	7	*	9	6	*	*	*	11	*	*		
hder	3	2	8	4	*	*	*	*	*	10	*	*		

maxNodos-1

Implementación vectorial de árboles binarios

Inserción y eliminación

(distinción entre celdas libres y ocupadas)



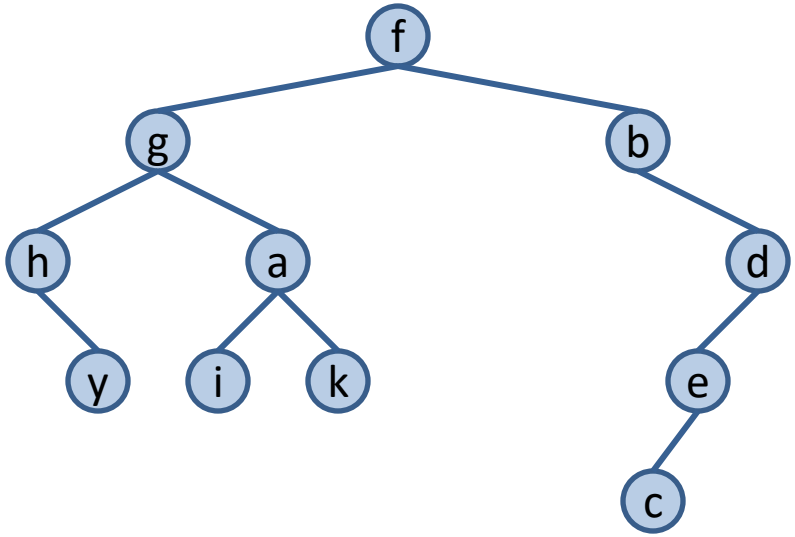
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
elto	f	g	a	b	d	h	y	i	k	e	z	c		
padre	*	0	1	0	3	1	5	2	2	4	3	9		
hizq	1	5	7	10	9	*	*	*	*	11	*	*		
hder	3	2	8	4	*	6	*	*	*	*	*	*		

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

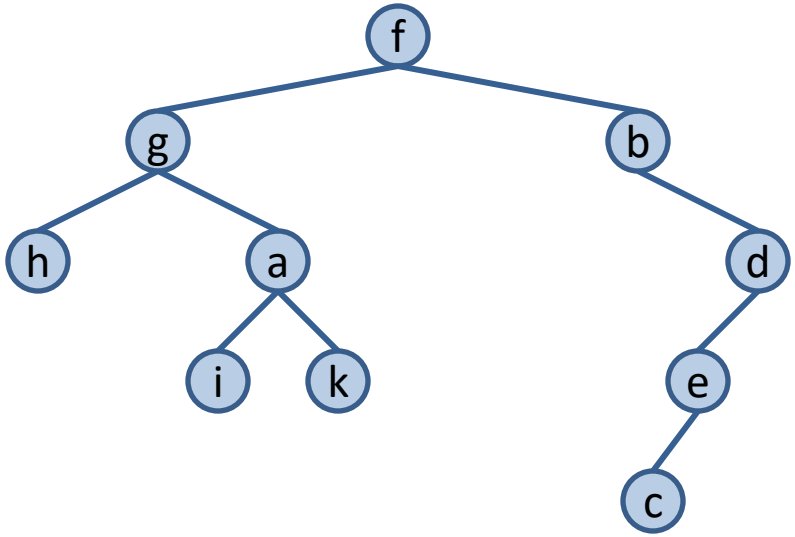
maxNodos-1

	*

Implementación vectorial de árboles binarios

Inserción y eliminación

(distinción entre celdas libres y ocupadas)



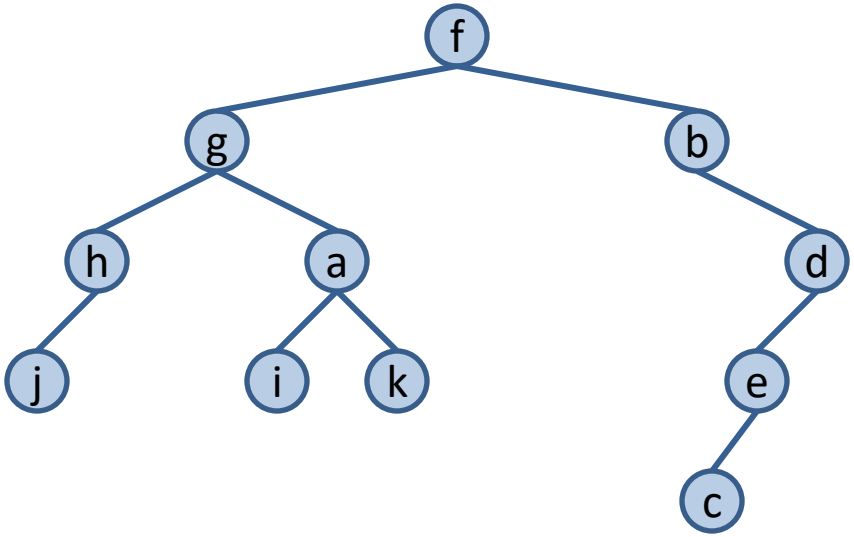
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
elto	f	g	a	b	d	h	y	i	k	e	z	c		
padre	0	0	1	0	3	1	*	2	2	4	*	9	*	*
hizq	1	5	7	*	9	*	*	*	*	11	*	*		
hder	3	2	8	4	*	*	*	*	*	*	*	*		

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	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	*	*	*	*	*	*	*	*		

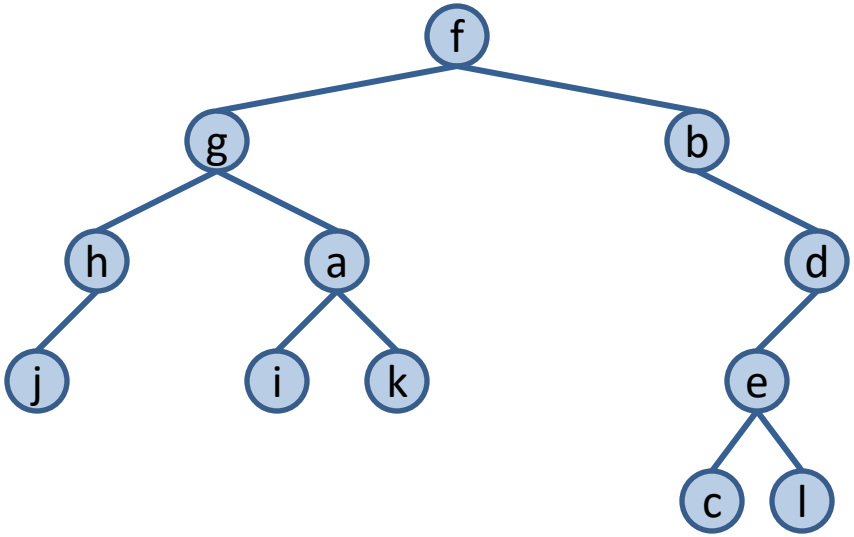
maxNodos-1	
	*

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

	*

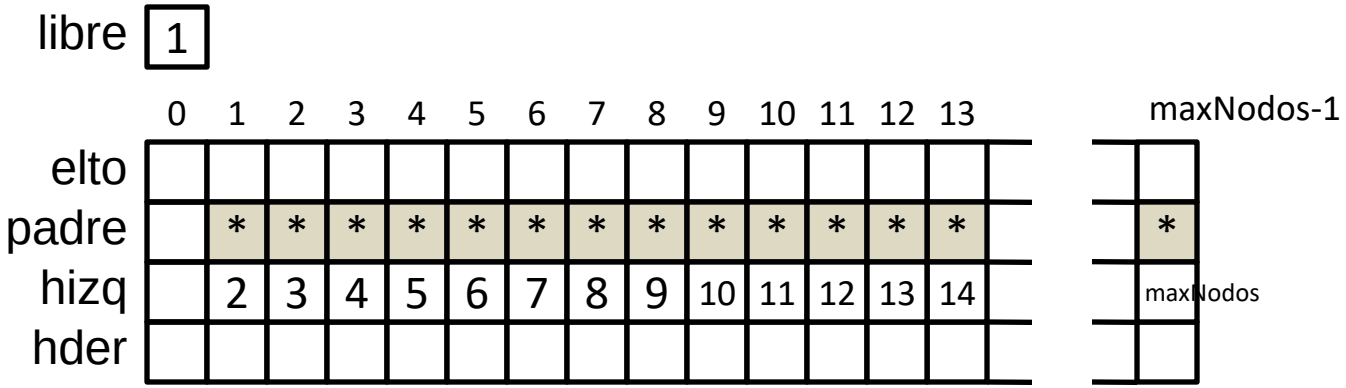
```
template <typename T> class Abin {
public:
   typedef size_t nodo;
   static const nodo NODO_NULO;
   // …
private:
   struct celda {
      T elto;
      nodo padre, hizq, hder;
   };

   celda* nodos;       // Vector de celdas
   size_t maxNodos,    // Tamaño del vector
          numNodos;    // Tamaño del árbol
};

// 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) :
   nodos{new celda[maxNodos]},
   maxNodos{maxNodos}, numNodos{0}
{
   for (nodo n = 1; n < maxNodos; ++n) // Marcar celdas libres.
      nodos[n].padre = NODO_NULO;
}

template <typename T>
void Abin<T>::insertarHijoIzqdo(nodo n, const T& e)
{
   // Comprobar precondiciones…
   // Buscar celda libre.
   nodo hizqdo = 1;
   while (nodos[hizqdo].padre != NODO_NULO) ++hizqdo;
   // Añadir el nuevo nodo.
   nodos[n].hizq = hizqdo;
   nodos[hizqdo] = {e, n, NODO_NULO, NODO_NULO};
   ++numNodos;
}
```

```
template <typename T>
inline void Abin<T>::eliminarHijoDrcho(nodo n)
{
   // Comprobar precondiciones.
   // …
   // Eliminar el nodo.
   nodos[nodos[n].hder].padre = NODO_NULO; // Marcar celda libre.
   nodos[n].hder = NODO_NULO;
   --numNodos;
}
```

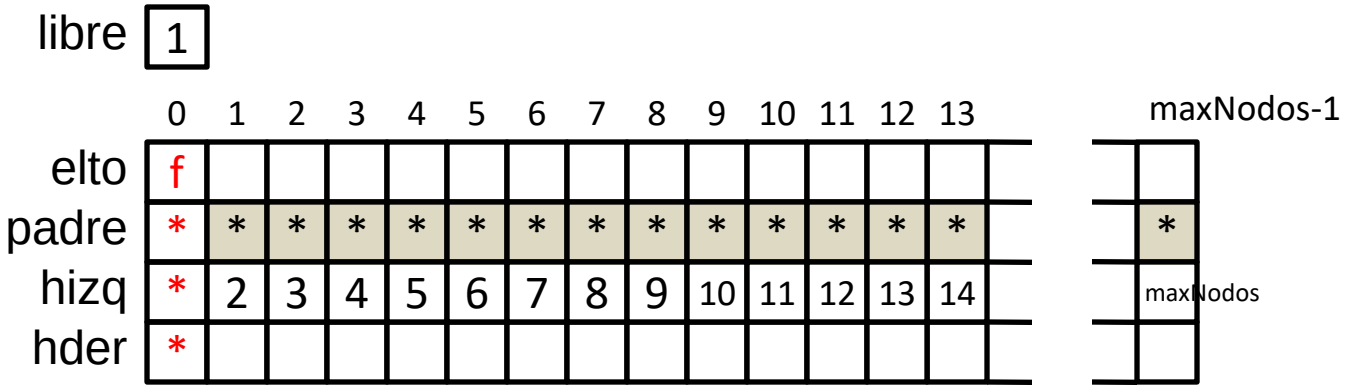
Implementación vectorial de árboles binarios

Inserción y eliminación eficientes



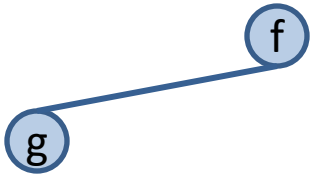
Implementación vectorial de árboles binarios

Inserción y eliminación eficientes



Implementación vectorial de árboles binarios

Inserción y eliminación eficientes



libre 2

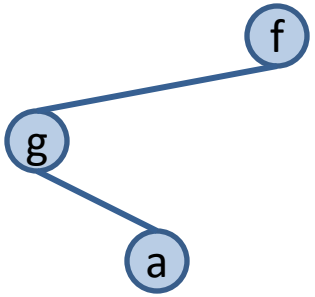
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
elto	f	g												
padre	*	0	*	*	*	*	*	*	*	*	*	*	*	*
hizq	1	*	3	4	5	6	7	8	9	10	11	12	13	14
hder	*	*												

maxNodos-1

	*
maxNodos	

Implementación vectorial de árboles binarios

Inserción y eliminación eficientes



libre 3

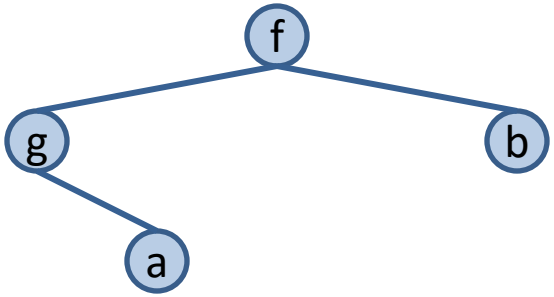
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
elto	f	g	a											
padre	*	0	1	*	*	*	*	*	*	*	*	*	*	*
hizq	1	*	*	4	5	6	7	8	9	10	11	12	13	14
hder	*	2	*											

maxNodos-1

	*
maxNodos	

Implementación vectorial de árboles binarios

Inserción y eliminación eficientes



libre 4

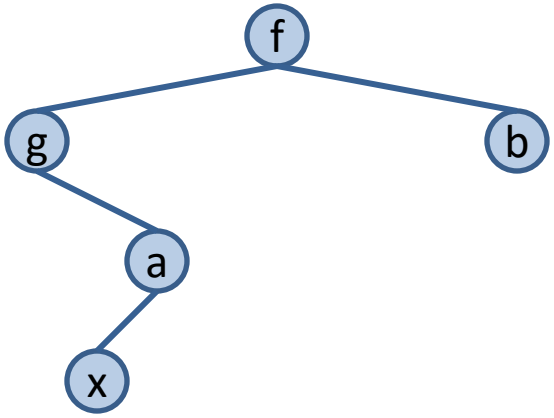
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
elto	f	g	a	b										
padre	*	0	1	0	*	*	*	*	*	*	*	*	*	*
hizq	1	*	*	*	5	6	7	8	9	10	11	12	13	14
hder	3	2	*	*										

maxNodos-1

	*
maxNodos	

Implementación vectorial de árboles binarios

Inserción y eliminación eficientes



libre 5

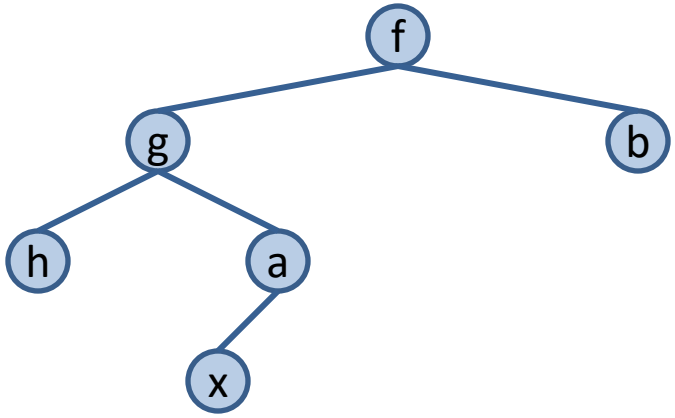
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
elto	f	g	a	b	x									
padre	*	0	1	0	2	*	*	*	*	*	*	*	*	*
hizq	1	*	4	*	*	6	7	8	9	10	11	12	13	14
hder	3	2	*	*	*									

maxNodos-1

	*
maxNodos	

Implementación vectorial de árboles binarios

Inserción y eliminación eficientes



libre 6

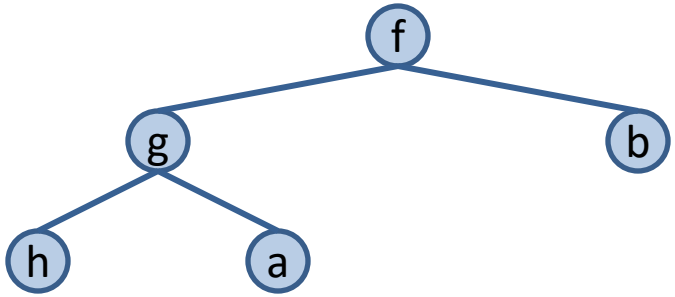
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
elto	f	g	a	b	x	h								
padre	*	0	1	0	2	1	*	*	*	*	*	*	*	*
hizq	1	5	4	*	*	*	7	8	9	10	11	12	13	14
hder	3	2	*	*	*	*								

maxNodos-1

	*
maxNodos	

Implementación vectorial de árboles binarios

Inserción y eliminación eficientes



libre 4

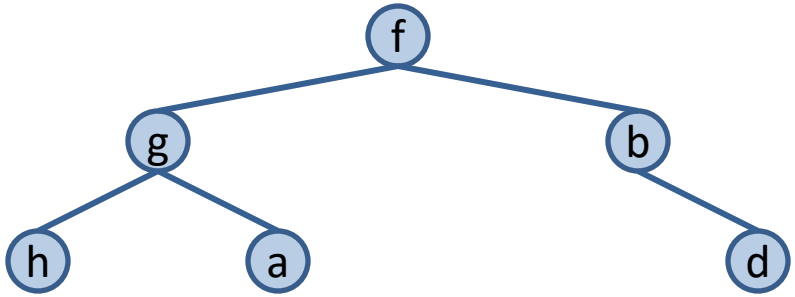
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
elto	f	g	a	b	x	h								
padre	*	0	1	0	*	1	*	*	*	*	*	*	*	*
hizq	1	5	*	*	6	*	7	8	9	10	11	12	13	14
hder	3	2	*	*	*	*								

maxNodos-1

	*
maxNodos	

Implementación vectorial de árboles binarios

Inserción y eliminación eficientes



libre 6

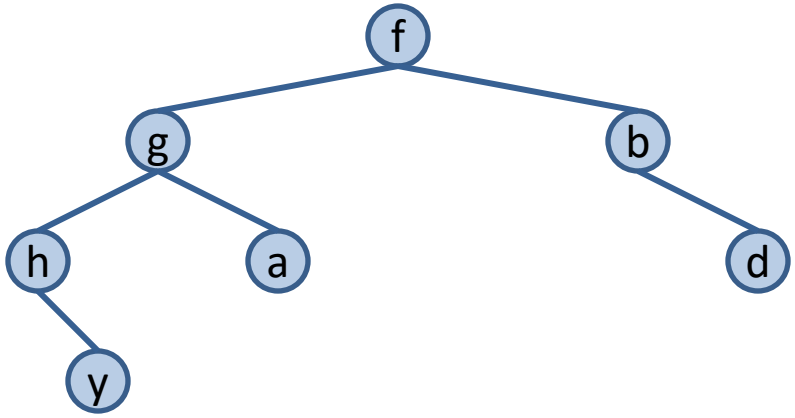
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
elto	f	g	a	b	d	h								
padre	*	0	1	0	3	1	*	*	*	*	*	*	*	*
hizq	1	5	*	*	*	*	7	8	9	10	11	12	13	14
hder	3	2	*	4	*	*								

maxNodos-1

	*
	maxNodos

Implementación vectorial de árboles binarios

Inserción y eliminación eficientes



libre 7

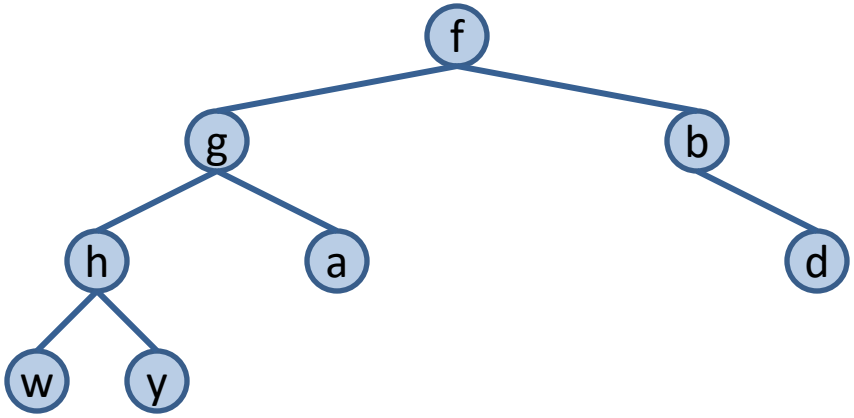
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
elto	f	g	a	b	d	h	y							
padre	*	0	1	0	3	1	5	*	*	*	*	*	*	*
hizq	1	5	*	*	*	*	*	8	9	10	11	12	13	14
hder	3	2	*	4	*	6	*							

maxNodos-1

*
maxNodos

Implementación vectorial de árboles binarios

Inserción y eliminación eficientes



libre 8

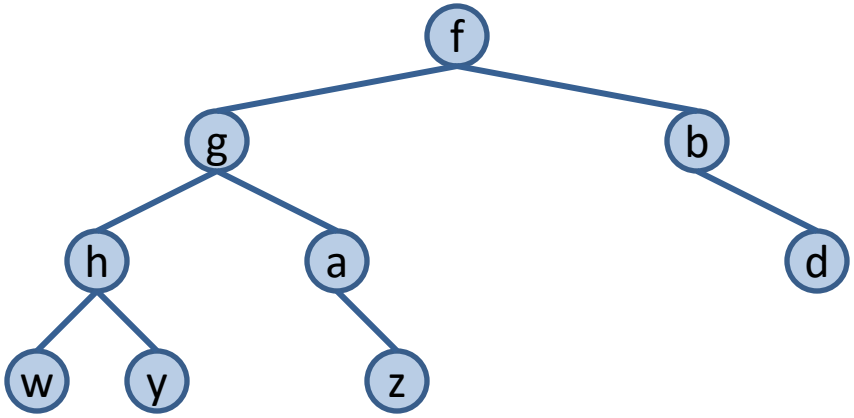
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
elto	f	g	a	b	d	h	y	w						
padre	*	0	1	0	3	1	5	5	*	*	*	*	*	*
hizq	1	5	*	*	*	7	*	*	9	10	11	12	13	14
hder	3	2	*	4	*	6	*	*						

maxNodos-1

	*
maxNodos	

Implementación vectorial de árboles binarios

Inserción y eliminación eficientes



libre 9

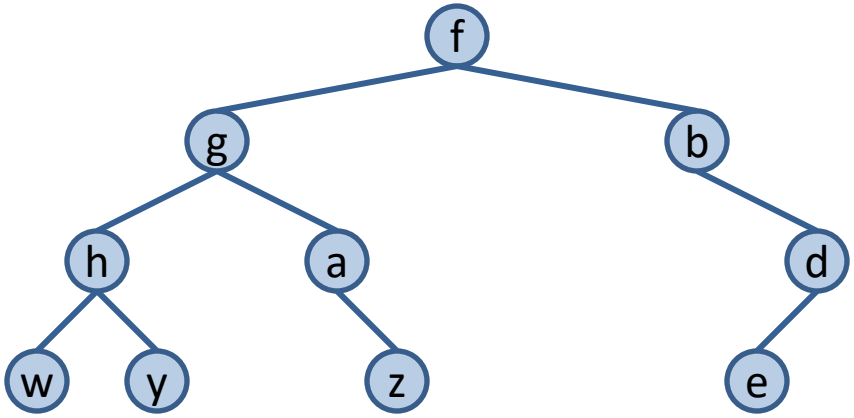
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
elto	f	g	a	b	d	h	y	w	z					
padre	*	0	1	0	3	1	5	5	2	*	*	*	*	*
hizq	1	5	*	*	*	7	*	*	*	10	11	12	13	14
hder	3	2	8	4	*	6	*	*	*					

maxNodos-1

	*
maxNodos	

Implementación vectorial de árboles binarios

Inserción y eliminación eficientes



libre 10

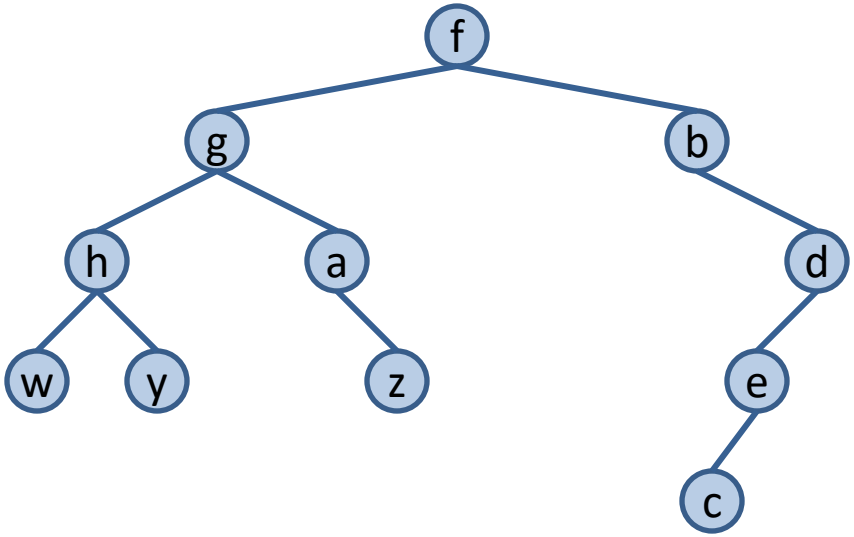
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
elto	f	g	a	b	d	h	y	w	z	e				
padre	*	0	1	0	3	1	5	5	2	4	*	*	*	*
hizq	1	5	*	*	9	7	*	*	*	*	11	12	13	14
hder	3	2	8	4	*	6	*	*	*	*				

maxNodos-1

*
maxNodos

Implementación vectorial de árboles binarios

Inserción y eliminación eficientes



libre 11

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
elto	f	g	a	b	d	h	y	w	z	e	c			
padre	*	0	1	0	3	1	5	5	2	4	9	*	*	*
hizq	1	5	*	*	9	7	*	*	*	10	*	12	13	14
hder	3	2	8	4	*	6	*	*	*	*	*			

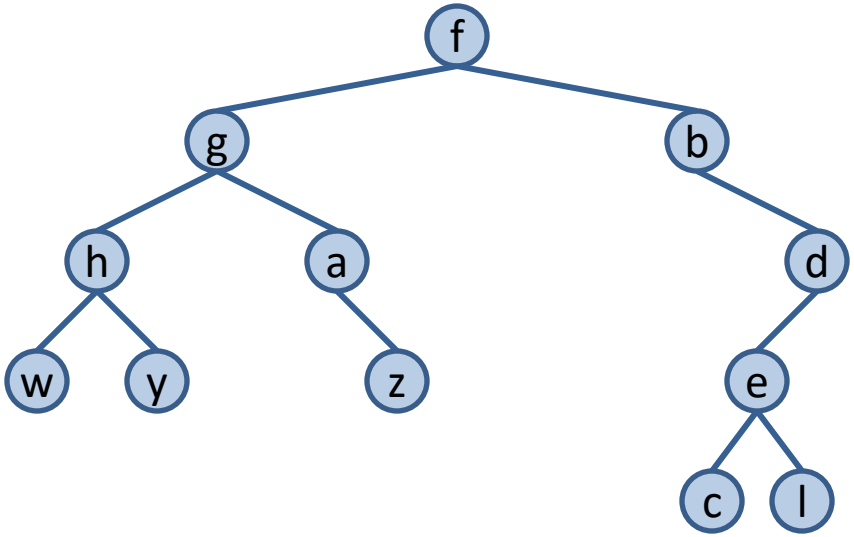
maxNodos-1

*

maxNodos

Implementación vectorial de árboles binarios

Inserción y eliminación eficientes



libre 12

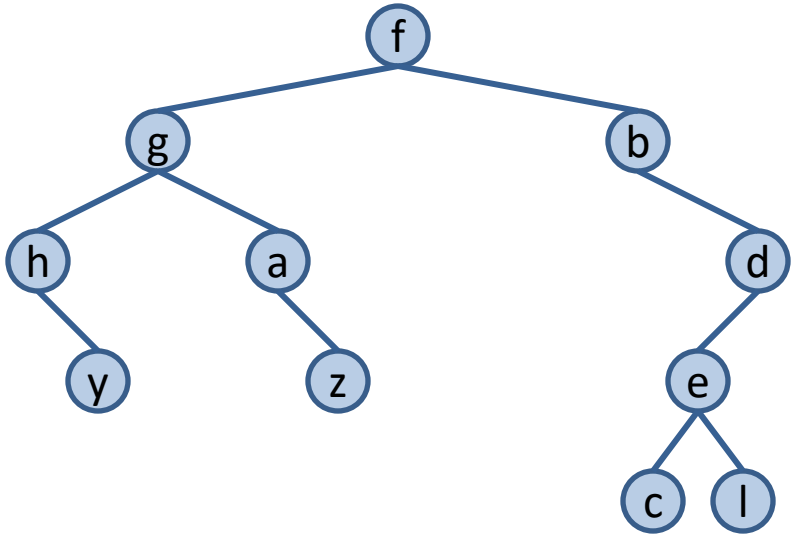
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
elto	f	g	a	b	d	h	y	w	z	e	c	l		
padre	*	0	1	0	3	1	5	5	2	4	9	9	*	*
hizq	1	5	*	*	9	7	*	*	*	10	*	*	13	14
hder	3	2	8	4	*	6	*	*	*	11	*	*		

maxNodos-1

*
maxNodos

Implementación vectorial de árboles binarios

Inserción y eliminación eficientes



libre 7

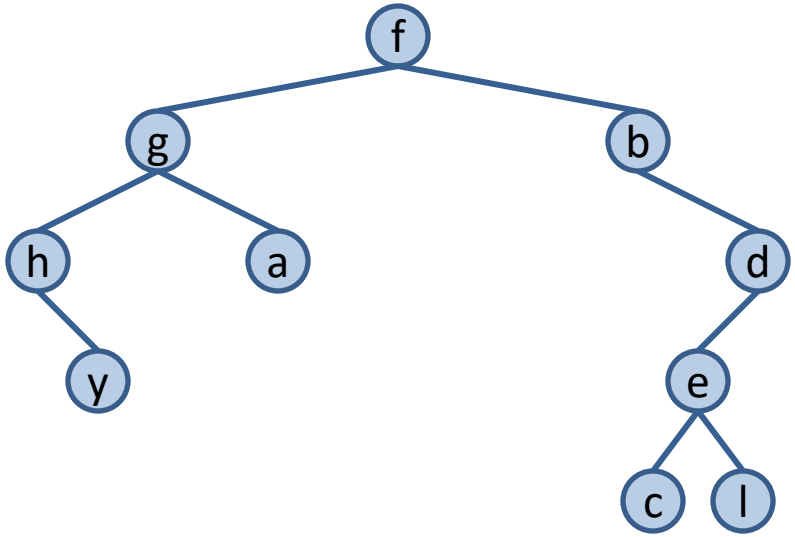
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
elto	f	g	a	b	d	h	y	w	z	e	c	l		
padre	*	0	1	0	3	1	5	*	2	4	9	9	*	*
hizq	1	5	*	*	9	*	*	12	*	10	*	*	13	14
hder	3	2	8	4	*	6	*	*	*	11	*	*		

maxNodos-1

	*
	maxNodos

Implementación vectorial de árboles binarios

Inserción y eliminación eficientes



libre 8

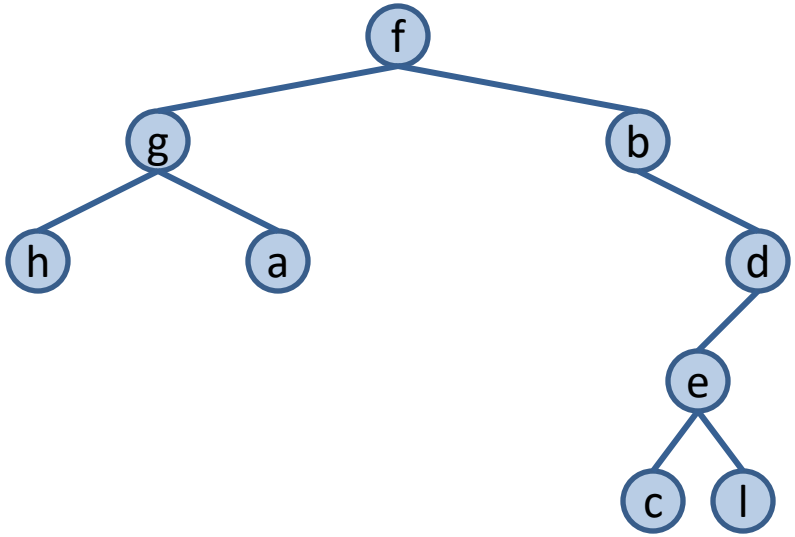
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
elto	f	g	a	b	d	h	y	w	z	e	c	l			
padre	*	0	1	0	3	1	5	*	*	4	9	9	*	*	
hizq	1	5	*	*	9	*	*	12	7	10	*	*	13	14	
hder	3	2	*	4	*	6	*	*	*	11	*	*			

maxNodos-1

	*
	maxNodos

Implementación vectorial de árboles binarios

Inserción y eliminación eficientes



libre 6

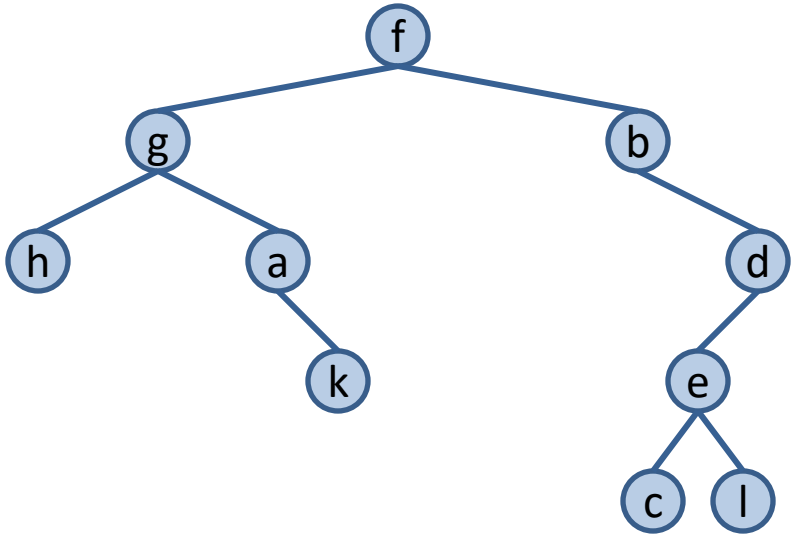
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
elto	f	g	a	b	d	h	y	w	z	e	c	l		
padre	*	0	1	0	3	1	*	*	*	4	9	9	*	*
hizq	1	5	*	*	9	*	8	12	7	10	*	*	13	14
hder	3	2	*	4	*	*	*	*	*	11	*	*		

maxNodos-1

	*
	maxNodos

Implementación vectorial de árboles binarios

Inserción y eliminación eficientes



libre 8

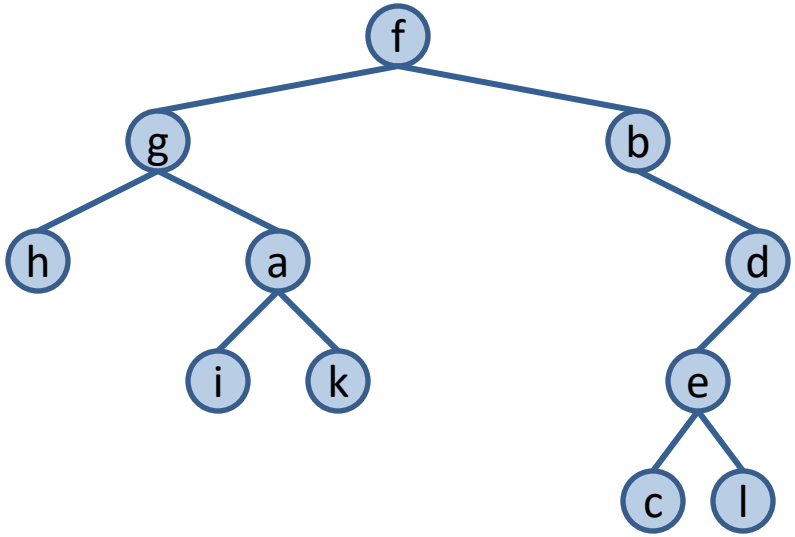
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
elto	f	g	a	b	d	h	k	w	z	e	c	l			
padre	*	0	1	0	3	1	2	*	*	4	9	9	*	*	
hizq	1	5	*	*	9	*	*	12	7	10	*	*	13	14	
hder	3	2	6	4	*	*	*	*	*	11	*	*			

maxNodos-1

	*
	maxNodos

Implementación vectorial de árboles binarios

Inserción y eliminación eficientes



libre 7

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
elto	f	g	a	b	d	h	k	w	i	e	c	l		
padre	*	0	1	0	3	1	2	*	2	4	9	9	*	*
hizq	1	5	8	*	9	*	*	12	*	10	*	*	13	14
hder	3	2	6	4	*	*	*	*	*	11	*	*		

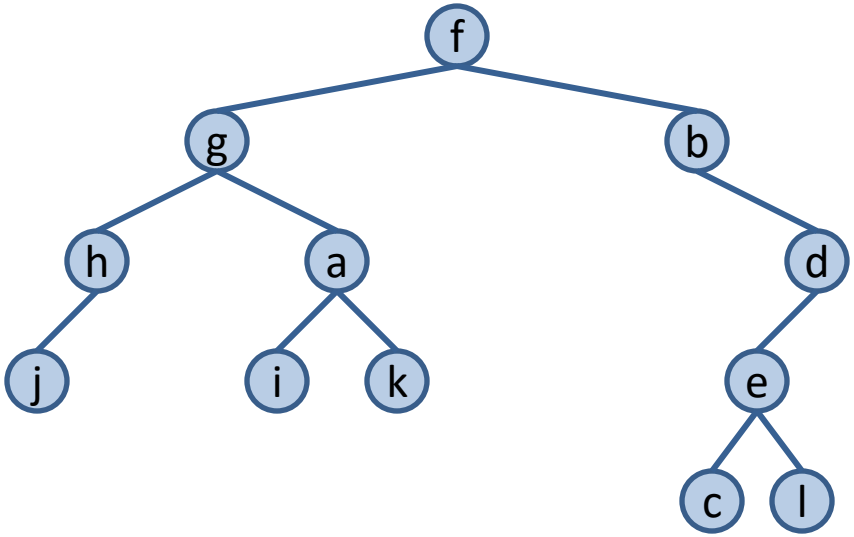
maxNodos-1

	*
maxNodos	

Implementación vectorial de árboles binarios

Inserción y eliminación eficientes

Inserción *O(1)*
Eliminación *O(1)*



libre 12

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
elto	f	g	a	b	d	h	k	j	i	e	c	l			
padre	*	0	1	0	3	1	2	5	2	4	9	9	*	*	
hizq	1	5	8	*	9	7	*	*	*	10	*	*	13	14	
hder	3	2	6	4	*	*	*	*	*	11	*	*			

maxNodos-1

	*
	maxNodos

```
#ifndef ABIN_VEC0_H
#define ABIN_VEC0_H
#include <cassert>
#include <cstddef>   // size_t
#include <cstdint>   // SIZE_MAX
#include <utility>   // swap

template <typename T> class Abin {
public:
   typedef size_t nodo;
   static const nodo NODO_NULO;
   explicit Abin(size_t maxNodos = 0);
   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 vacio() const;
   size_t tama() const;
   size_t tamaMax() const; // Requerido por la implementación
```

```
   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& A);                  // Ctor. de copia
   Abin& operator =(const Abin& A);      // Asig. de árboles
   ~Abin();                              // Destructor
private:
   struct celda {
      T elto;
      nodo padre, hizq, hder;
   };
   celda* nodos;       // Vector de celdas
   size_t maxNodos,    // Tamaño del vector
          numNodos;    // Tamaño del árbol
   nodo libre;         // Lista de celdas libres

   bool valido(nodo n) const;
}; // class Abin
```

```
// Definición del nodo nulo
template <typename T>
const typename Abin<T>::nodo Abin<T>::NODO_NULO{SIZE_MAX};

//
// Método privado
//
template <typename T>
inline bool Abin<T>::valido(nodo n) const
{ // Comprobar si n es un nodo del árbol.
   return !vacio() &&
          n < maxNodos &&                    // Celda del vector
          (n == 0 || nodos[n].padre != NODO_NULO); // ocupada.
}
```

```
template <typename T>
Abin<T>::Abin(size_t maxNodos) :
    nodos{new celda[maxNodos]},
    maxNodos{maxNodos},
    numNodos{0}
{
    if (maxNodos > 1) { // Crear la lista de celdas
        // libres enlazadas por el campo hizq.
        libre = 1;
        for (nodo n = 1; n < maxNodos; ++n) {
            // Añadir celda n+1 a la lista.
            nodos[n].hizq = n + 1;
            #ifndef NDEBUG
            // Sólo para comprobar precondiciones.
            nodos[n].padre = NODO_NULO;// Marcar celda libre.
            #endif
        }
    }
}
```

```
template <typename T>
inline void Abin<T>::insertarRaiz(const T& e)
{
   assert(maxNodos > 0);
   assert(vacio());
   nodos[0] = {e, NODO_NULO, NODO_NULO, NODO_NULO};
   numNodos = 1;
}
```

```
template <typename T>
inline void Abin<T>::insertarHijoIzqdo(nodo n, const T& e)
{
   assert(tama() < tamaMax());
   assert(valido(n));
   assert(nodos[n].hizq == NODO_NULO);
   // Sacar la primera celda de la lista de libres.
   nodo hizqdo = libre;
   libre = nodos[libre].hizq;
   // Añadir el nuevo nodo.
   nodos[n].hizq = hizqdo;
   nodos[hizqdo] = {e, n, NODO_NULO, NODO_NULO};
   ++numNodos;
}
```

```
template <typename T>
inline void Abin<T>::insertarHijoDrcho(nodo n, const T& e)
{
    assert(tama() < tamaMax());
    assert(valido(n));
    assert(nodos[n].hder == NODO_NULO);
    // Sacar la primera celda de la lista de libres.
    nodo hdrcho = libre;
    libre = nodos[libre].hizq;
    // Añadir el nuevo nodo.
    nodos[n].hder = hdrcho;
    nodos[hdrcho] = {e, n, NODO_NULO, NODO_NULO};
    ++numNodos;
}
```

```
template <typename T>
inline void Abin<T>::eliminarHijoIzqdo(nodo n)
{
   assert(valido(n));
   nodo hizqdo = nodos[n].hizq;
   assert(hizqdo != NODO_NULO);  // Existe hijo izqdo.
   assert(nodos[hizqdo].hizq == NODO_NULO &&  // y es
          nodos[hizqdo].hder == NODO_NULO);   // hoja.
   nodos[n].hizq = NODO_NULO;
   // Añadir hizqdo al inicio de la lista de libres.
   nodos[hizqdo].hizq = libre;
   libre = hizqdo;
   #ifndef NDEBUG
   nodos[hizqdo].padre = NODO_NULO; // Marcar celda libre.
   #endif
   --numNodos;
}
```

```
template <typename T>
inline void Abin<T>::eliminarHijoDrcho(nodo n)
{
   assert(valido(n));
   nodo hdrcho = nodos[n].hder;
   assert(hdrcho != NODO_NULO);  // Existe hijo drcho.
   assert(nodos[hdrcho].hizq == NODO_NULO &&  // y es
          nodos[hdrcho].hder == NODO_NULO);   // hoja.
   nodos[n].hder = NODO_NULO;
   // Añadir hizqdo al inicio de la lista de libres.
   nodos[hdrcho].hizq = libre;
   libre = hdrcho;
   #ifndef NDEBUG
   nodos[hdrcho].padre = NODO_NULO; // Marcar celda libre.
   #endif
   --numNodos;
}
```

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

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

template <typename T>
inline size_t Abin<T>::tama() const
{ return numNodos; }

template <typename T>
inline size_t Abin<T>::tamaMax() const
{ return maxNodos; }
```

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

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

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

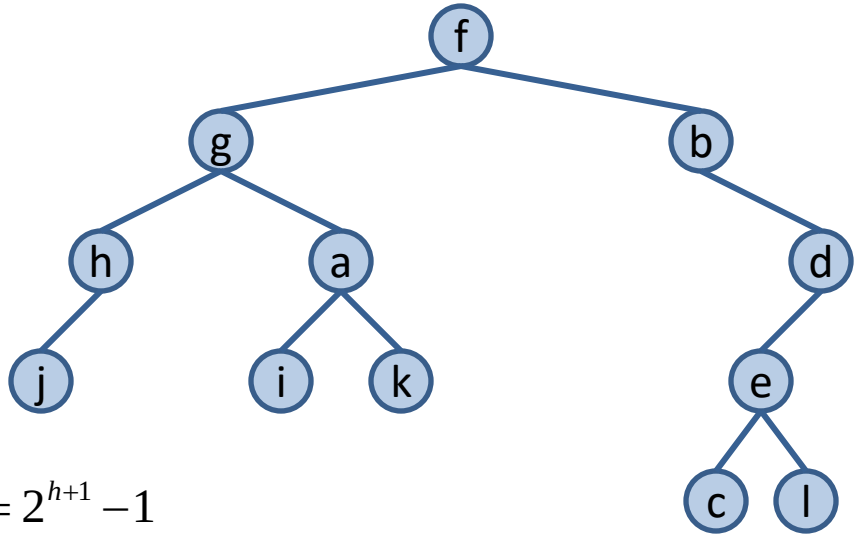
```
template <typename T>
inline typename Abin<T>::nodo Abin<T>::padre(nodo n) const
{
   assert(valido(n));
   return nodos[n].padre;
}

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

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

```
template <typename T>
Abin<T>::Abin(const Abin& A) : Abin{A.maxNodos}
{
   if (!A.vacio()) {
      for (nodo n = 0; n < maxNodos; ++n)
         nodos[n] = A.nodos[n];
      numNodos = A.numNodos;
      libre = A.libre;
   }
}

template <typename T>
inline Abin<T>& Abin<T>::operator =(const Abin& A)
{
   Abin B{A};
   std::swap(nodos, B.nodos);
   std::swap(maxNodos, B.maxNodos);
   std::swap(numNodos, B.numNodos);
   std::swap(libre, B.libre);
   return *this;
}
```

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

#endif // ABIN_VEC0_H
```

Implementación de un árbol binario mediante un vector de posiciones relativas

hizq(i) = 2i+1
hder(i) = 2i+2
padre(i) = (i-1)/2



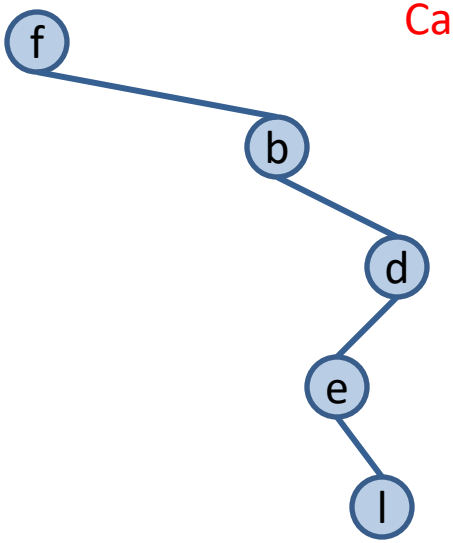
maxNodos = $\sum_{i=0}^{h} 2^i = 2^{h+1} - 1$

maxNodos-1

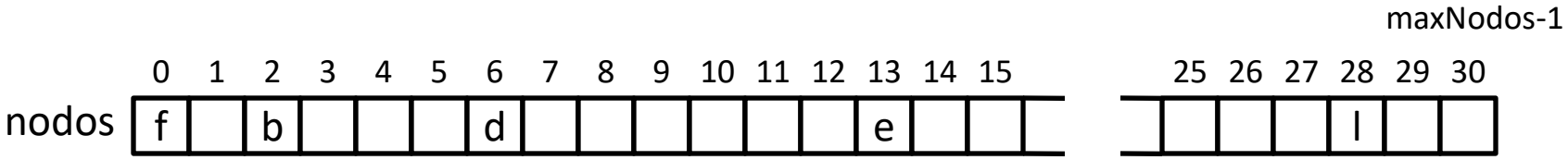
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15		25	26	27	28	29	30
nodos	f	g	b	h	a		d	j		i	k			e						c	l		

Implementación de un árbol binario mediante un vector de posiciones relativas

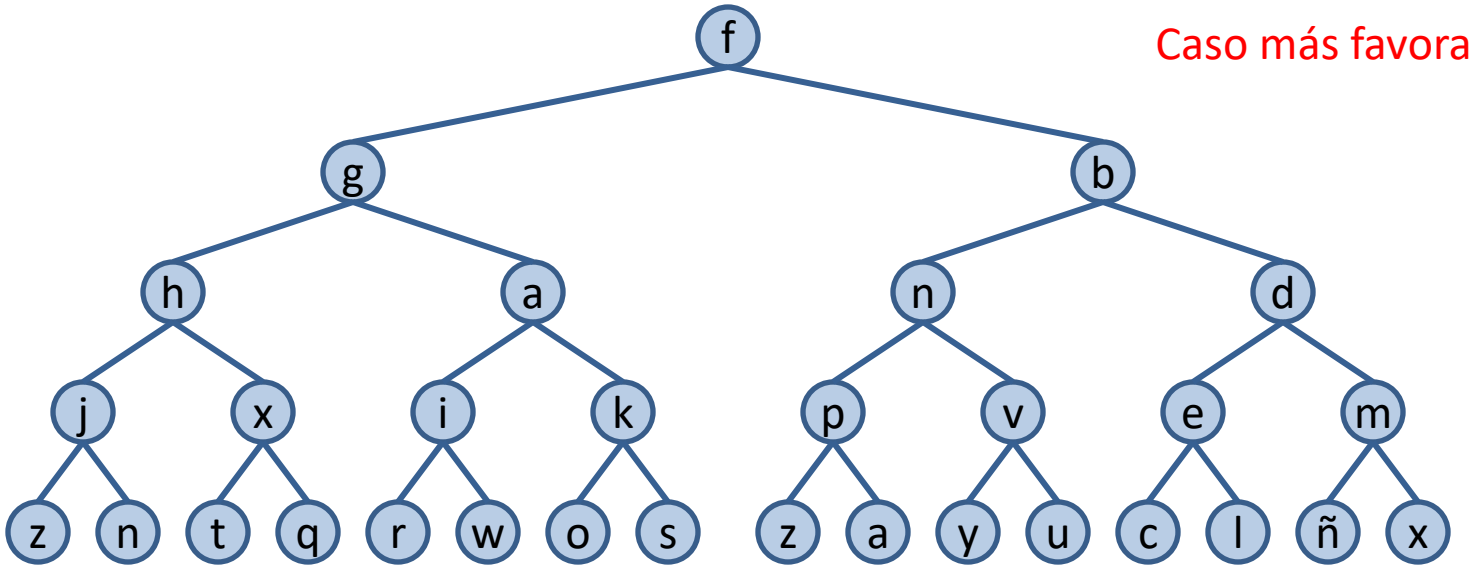
Caso más desfavorable



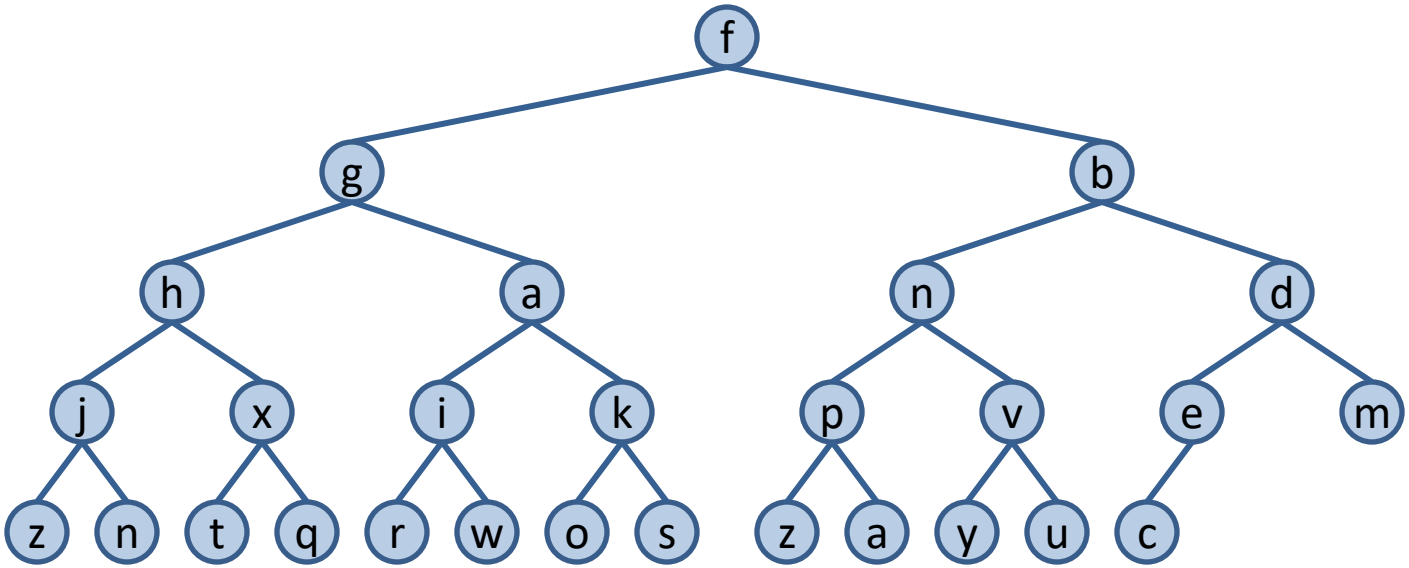
Sea un árbol de altura máxima *h*. La ausencia de un nodo en el nivel $n \leq h$ provocará $2^{h-n+1}-1$ posiciones libres en el vector.



Implementación de un árbol binario mediante un vector de posiciones relativas



Implementación de un árbol binario mediante un vector de posiciones relativas

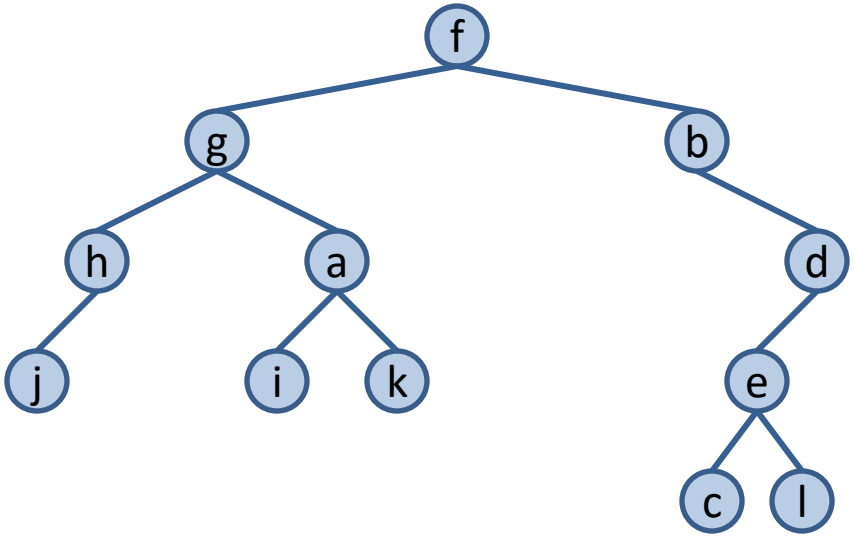


maxNodos-1

	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	@	@

Un valor del tipo de los elementos del árbol no significativo en la aplicación se utilizará para marcar las posiciones libres del vector.

```
#ifndef ABIN_VEC1_H
#define ABIN_VEC1_H
#include <cassert>
#include <cstddef>   // size_t
#include <cstdint>   // SIZE_MAX
#include <utility>   // swap

template <typename T> class Abin {
public:
   typedef size_t nodo;
   static const nodo NODO_NULO;

   explicit Abin(size_t maxNodos = 0, 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 vacio() const;
   size_t tama() const;
   size_t tamaMax() const; // Requerido por la implementación
   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& A);                    // Ctor. de copia
   Abin& operator =(const Abin& A);        // Asignación de árboles
   ~Abin();                                // Destructor
private:
   T* nodos;           // Vector de elementos
   size_t maxNodos,    // Tamaño del vector
          numNodos;    // Tamaño del árbol
   T ELTO_NULO;        // Marca celdas libres

   bool valido(nodo n) const;
}; // class Abin
```

```
// Definición del nodo nulo
template <typename T>
const typename Abin<T>::nodo Abin<T>::NODO_NULO{SIZE_MAX};

//
// Método privado
//
template <typename T>
inline bool Abin<T>::valido(nodo n) const
{ // Comprobar si n es un nodo del árbol.
   return n < maxNodos &&    // Celda del vector
          !(nodos[n] == ELTO_NULO);   // ocupada.
}
```

```
template <typename T>
Abin<T>::Abin(size_t maxNodos, const T& e_nulo) :
   nodos{new T[maxNodos]},
   maxNodos{maxNodos},
   numNodos{0},
   ELTO_NULO{e_nulo}
{
   // Marcar libres todas las celdas.
   for (nodo n = 0; n < maxNodos; ++n)
      nodos[n] = ELTO_NULO;
}

template <typename T>
inline void Abin<T>::insertarRaiz(const T& e)
{
   assert(maxNodos > 0);
   assert(vacio());
   nodos[0] = e;
   numNodos = 1;
}
```

```
template <typename T>
inline void Abin<T>::insertarHijoIzqdo(nodo n, const T& e)
{
   assert(valido(n));
   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;
   ++numNodos;
}

template <typename T>
inline void Abin<T>::insertarHijoDrcho(nodo n, const T& e)
{
   assert(valido(n));
   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;
   ++numNodos;
}
```

```
template <typename T>
inline void Abin<T>::eliminarHijoIzqdo(nodo n)
{
   assert(valido(n));
   assert(valido(2*n+1));
   assert(!valido(4*n+3) && !valido(4*n+4));
   nodos[2*n+1] = ELTO_NULO;
   --numNodos;
}

template <typename T>
inline void Abin<T>::eliminarHijoDrcho(nodo n)
{
   assert(valido(n));
   assert(valido(2*n+2));
   assert(!valido(4*n+5) && !valido(4*n+6));
   nodos[2*n+2] = ELTO_NULO;
   --numNodos;
}
```

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

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

template <typename T>
inline size_t Abin<T>::tama() const
{ return numNodos; }

template <typename T>
inline size_t Abin<T>::tamaMax() const
{ return maxNodos; }
```

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

template <typename T>
inline T& Abin<T>::elemento(nodo n)
{
   assert(valido(n));
   return nodos[n];
}

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

```
template <typename T>
inline typename Abin<T>::nodo Abin<T>::padre(nodo n) const
{
   assert(valido(n));
   return n ? (n-1)/2 : NODO_NULO;
}

template <typename T>
inline typename Abin<T>::nodo Abin<T>::hijoIzqdo(nodo n) const
{
   assert(valido(n));
   return valido(2*n+1) ? 2*n+1 : NODO_NULO;
}

template <typename T>
inline typename Abin<T>::nodo Abin<T>::hijoDrcho(nodo n) const
{
   assert(valido(n));
   return valido(2*n+2) ? 2*n+2 : NODO_NULO;
}
```

```
template <typename T>
Abin<T>::Abin(const Abin& A) : Abin{A.maxNodos, A.ELTO_NULO}
{
   if (!A.vacio()) {
      for (nodo n = 0; n < maxNodos; ++n)
         nodos[n] = A.nodos[n];
      numNodos = A.numNodos;
   }
}

template <typename T>
inline Abin<T>& Abin<T>::operator =(const Abin& A)
{
   Abin B{A};
   std::swap(nodos, B.nodos);
   std::swap(maxNodos, B.maxNodos);
   std::swap(numNodos, B.numNodos);
   std::swap(ELTO_NULO, B.ELTO_NULO);
   return *this;
}
```

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

#endif // ABIN_VEC1_H
```