Estructuras de Datos no Lineales

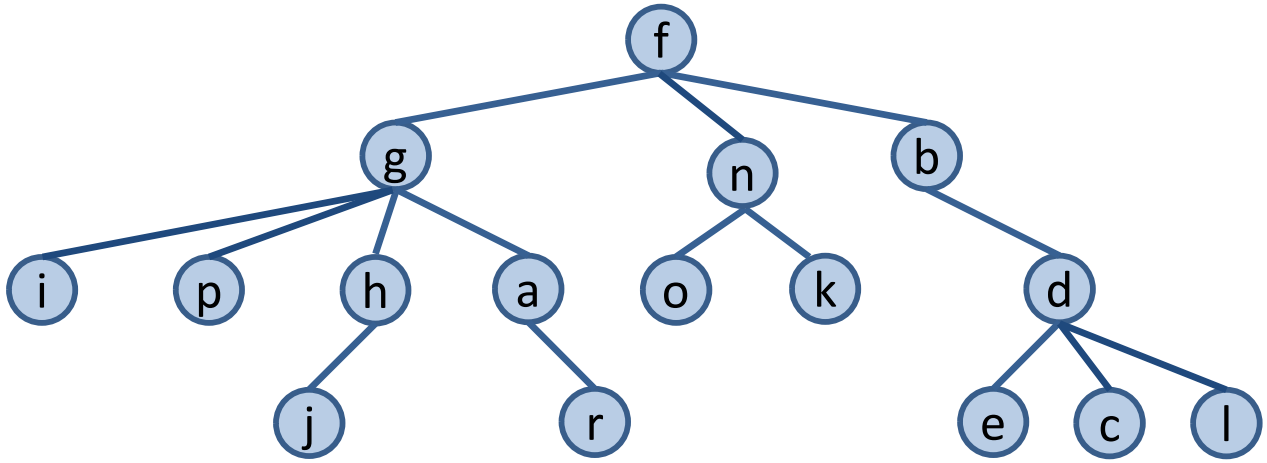
1.3. Árboles generales

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

Versión 2.1

Árboles generales

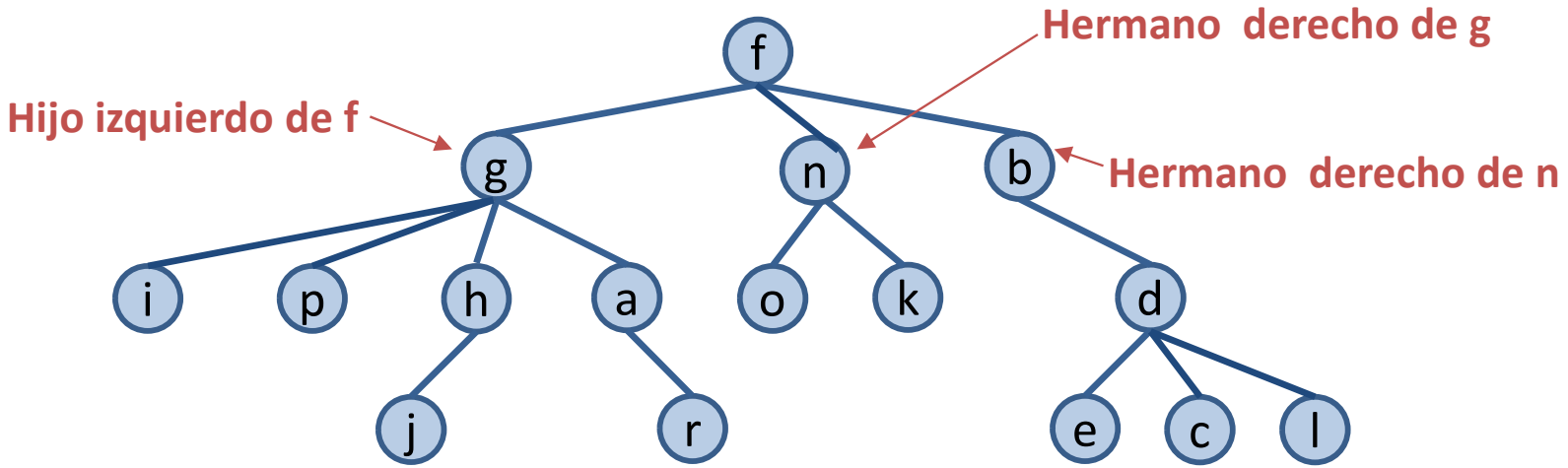
Un árbol cuyos nodos son de cualquier grado, es decir, pueden tener un número cualquiera de hijos, es un árbol general.



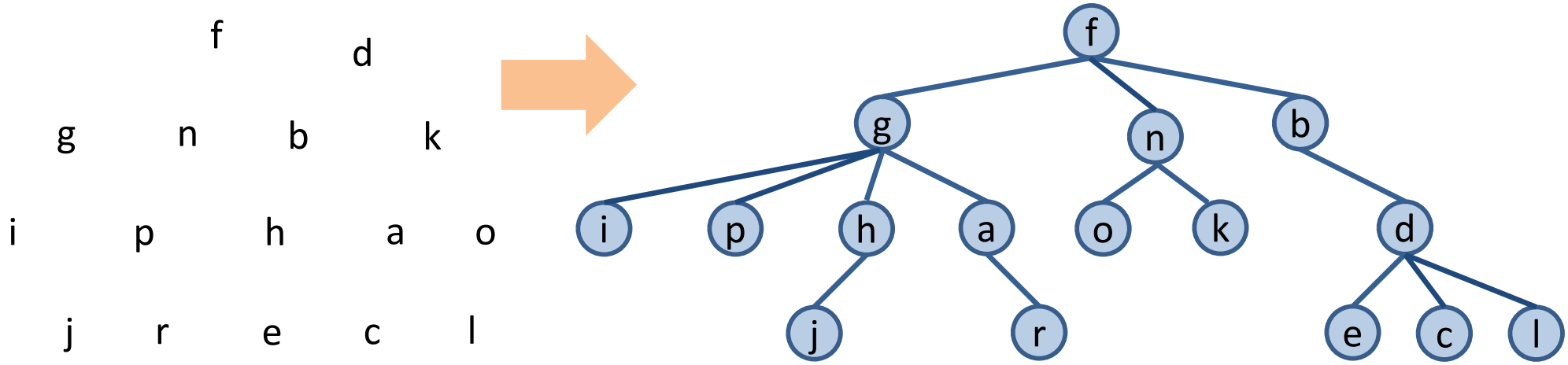
TAD Árbol General

Definición:

Un árbol general se define como un árbol cuyos nodos son de cualquier grado, es decir, pueden tener un número cualquiera de hijos. Los hijos de un nodo están ordenados de izquierda a derecha, de tal forma que el primer hijo de un nodo se llama *hijo izquierdo*, el segundo es el *hermano derecho* de éste, el tercero es el *hermano derecho* del segundo y así sucesivamente.



Construcción de un árbol general

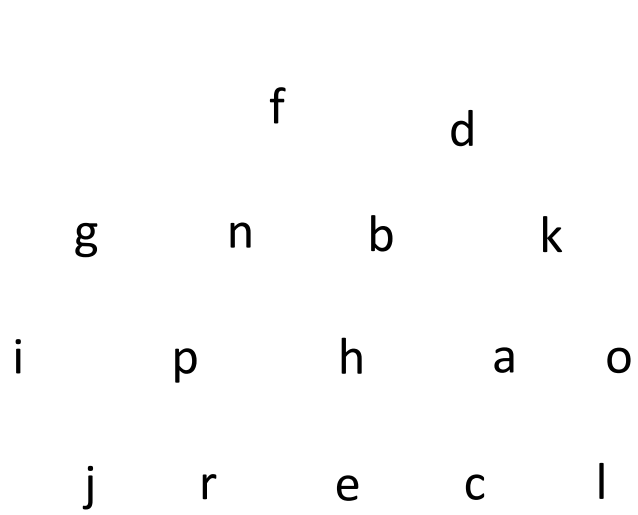


Construcción añadiendo los nodos uno a uno desde la raíz hacia las hojas:

Agen(); // Árbol vacío
void insertarRaiz(const T& e);
void insertarHijoIzqdo(nodo n, const T& e);
void insertarHermDrcho(nodo n, const T& e);

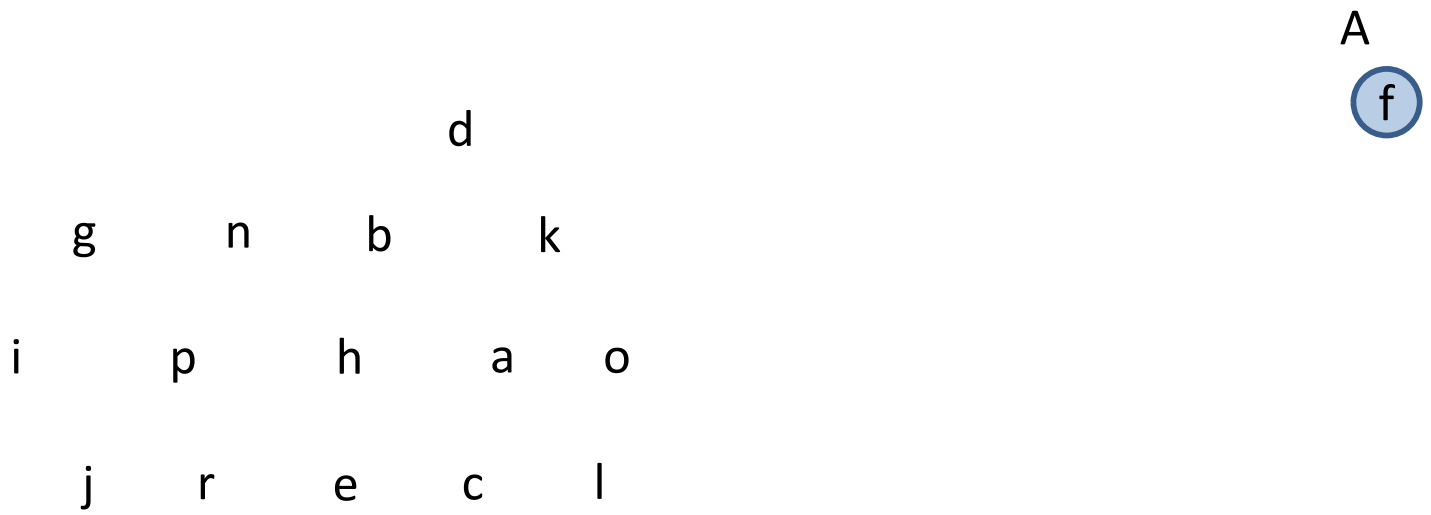
Construcción de un árbol general

Creación del árbol A como un contenedor vacío.



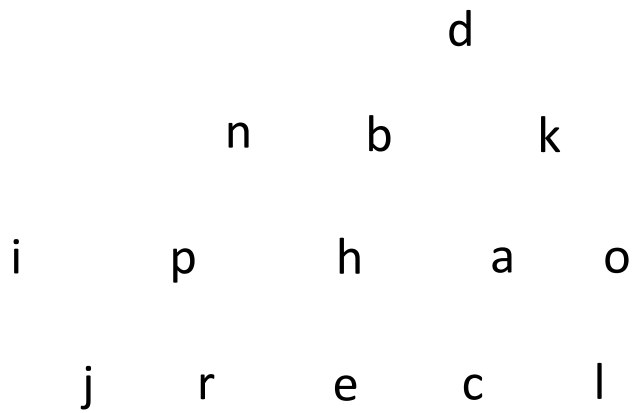
Agen A;

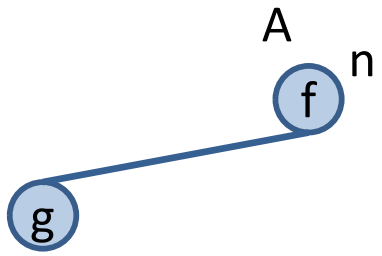
Construcción de un árbol general



A.insertarRaiz('f');

Construcción de un árbol general





A.insertarHijoIzqdo(n, ‘g’);

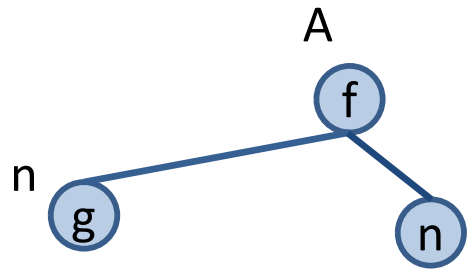
Construcción de un árbol general

d

b k

i p h a o

j r e c l



A.insertarHermDrcho(n, ‘n’);

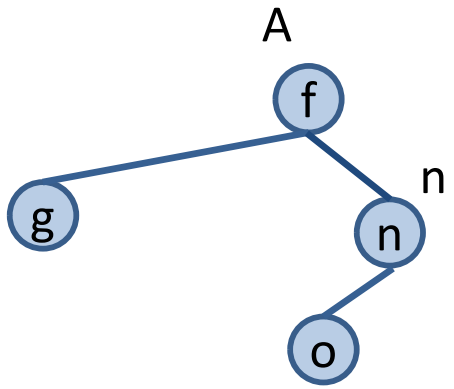
Construcción de un árbol general

d

b k

i p h a

j r e c l



A.insertarHijoIzqdo(n, ‘o’);

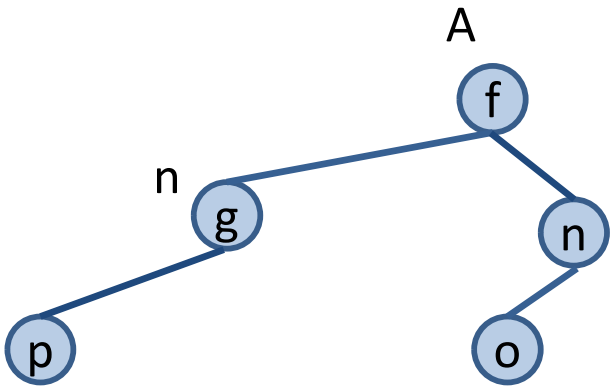
Construcción de un árbol general

d

b k

i h a

j r e c l



A.insertarHijoIzqdo(n, ‘p’);

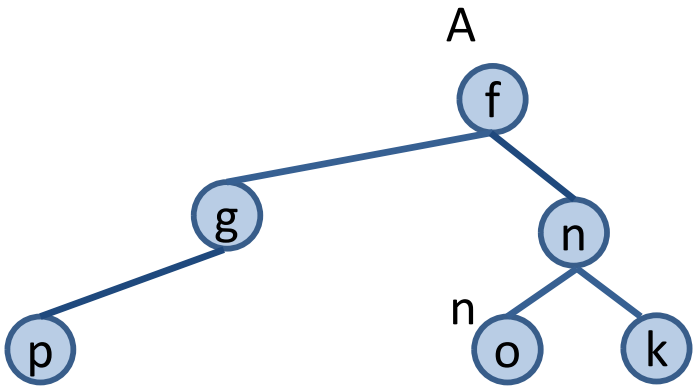
Construcción de un árbol general

d

b

i h a

j r e c l



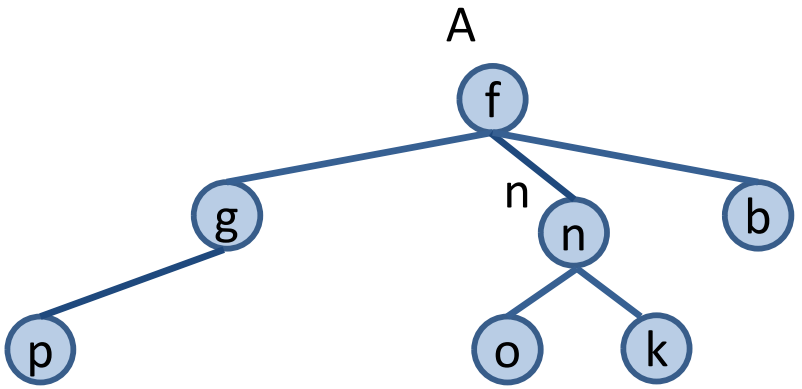
A.insertarHermDrcho(n, ‘k’);

Construcción de un árbol general

d

i h a

j r e c l

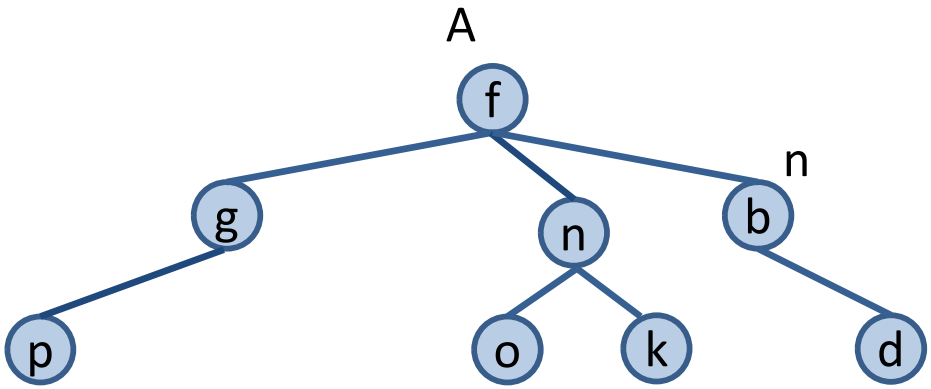


A.insertarHermDrcho(n, ‘b’);

Construcción de un árbol general

i h a

j r e c l

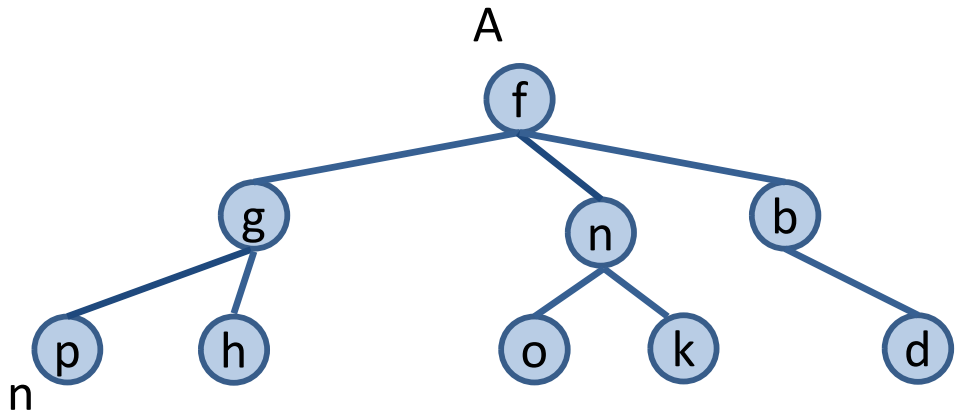


A.insertarHijoIzqdo(n, ‘d’);

Construcción de un árbol general

i a

j r e c l

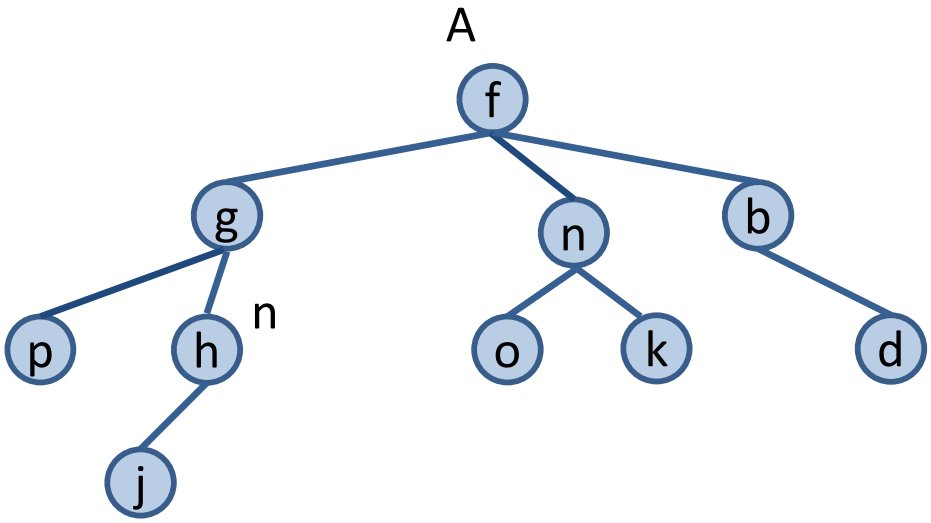


A.insertarHermDrcho(n, ‘h’);

Construcción de un árbol general

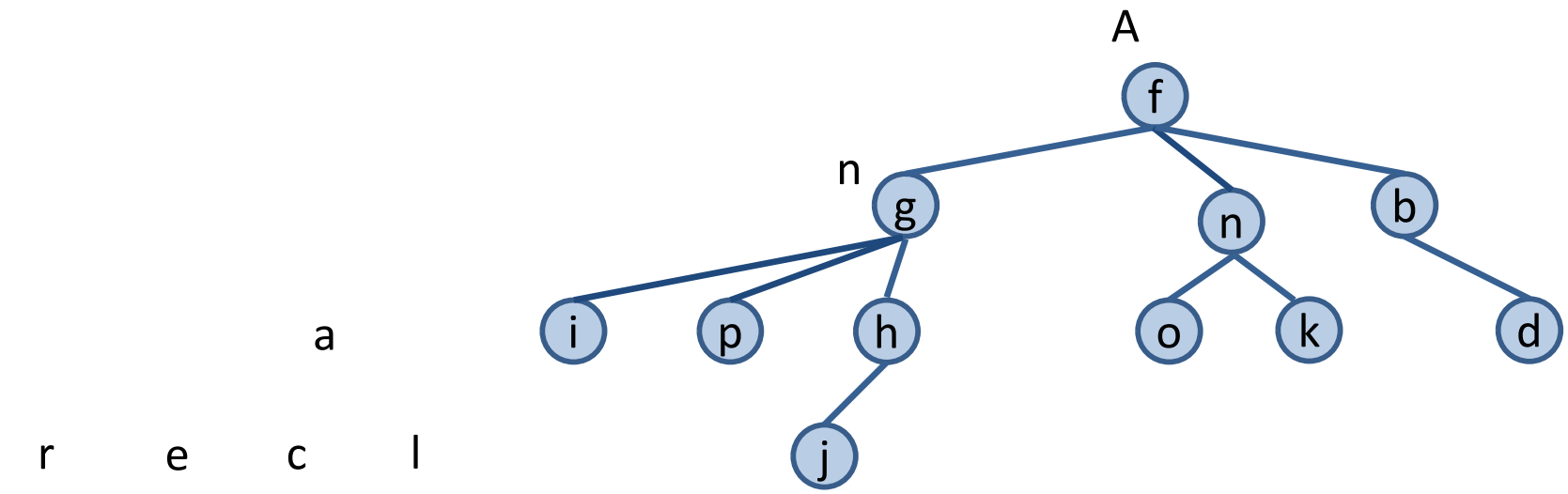
i a

r e c l



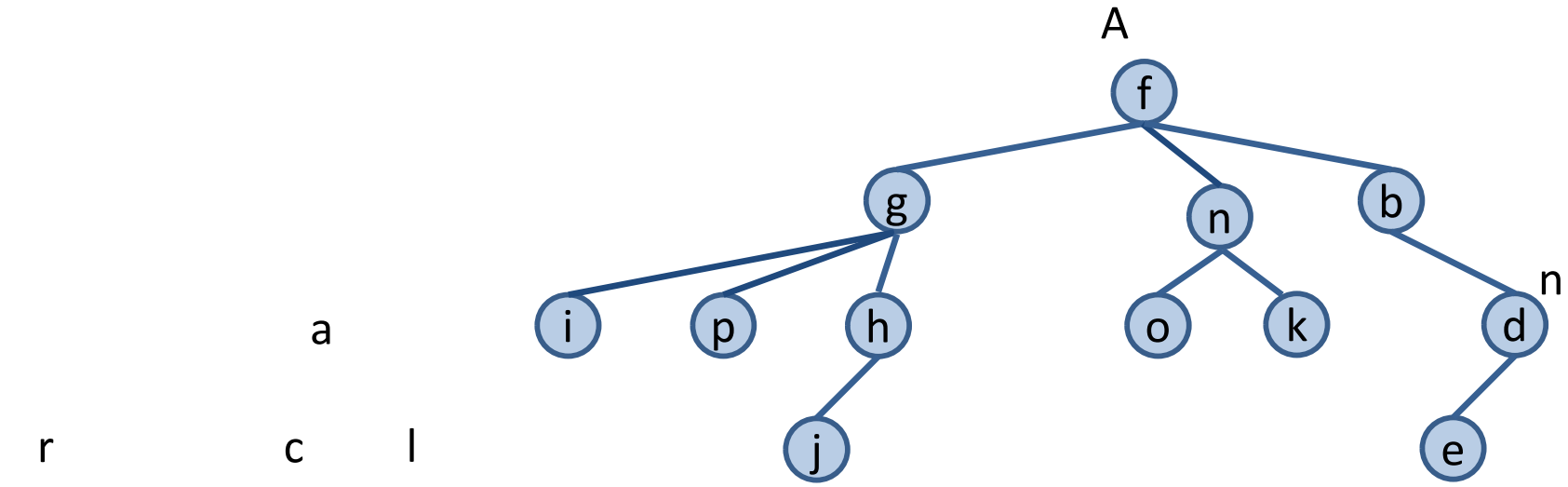
A.insertarHijoIzqdo(n, ‘j’);

Construcción de un árbol general



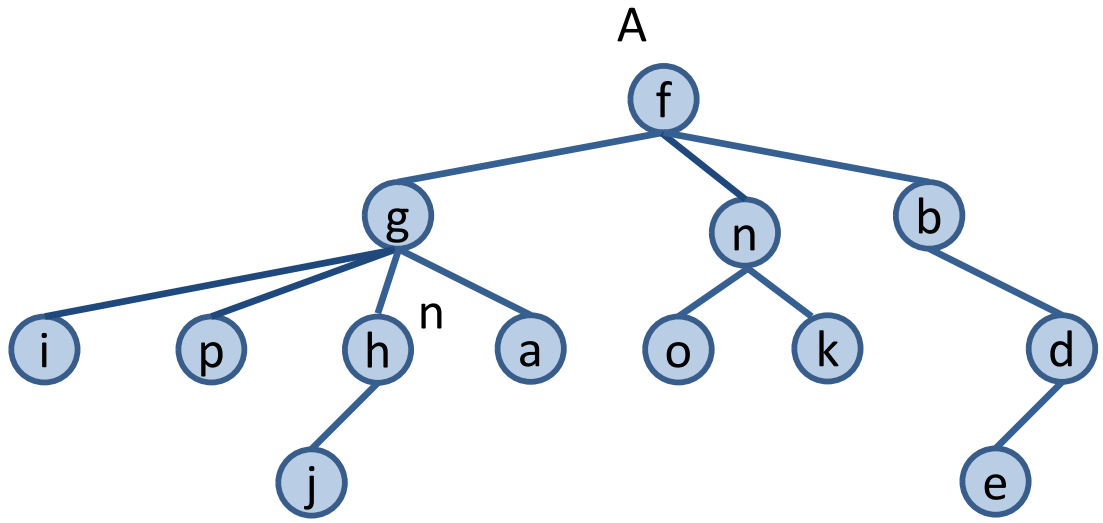
A.insertarHijoIzqdo(n, ‘i’);

Construcción de un árbol general



A.insertarHijoIzqdo(n, ‘e’);

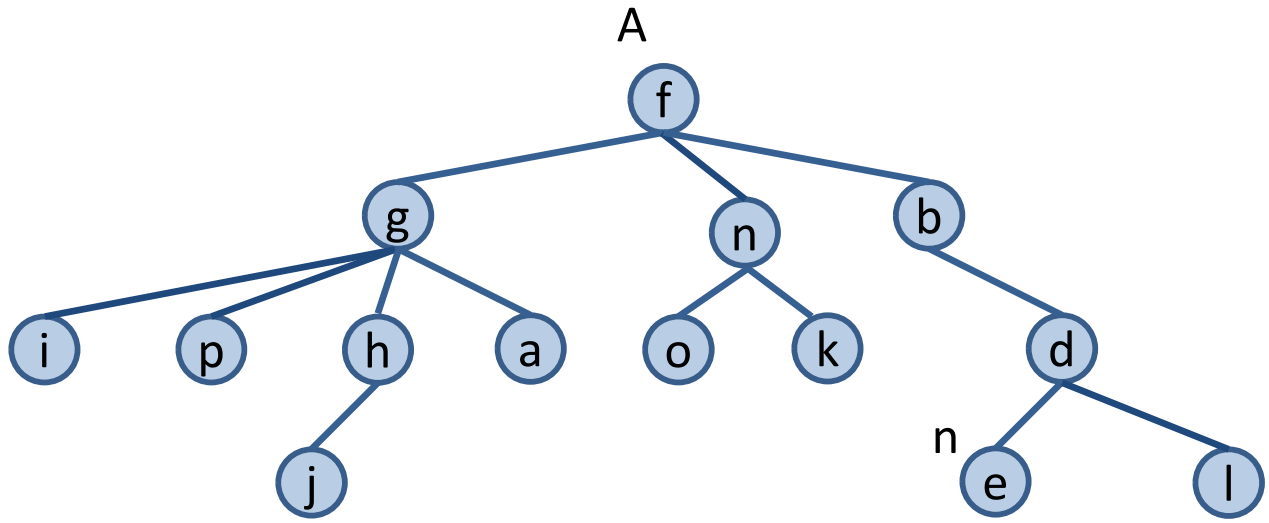
Construcción de un árbol general



r c l

A.insertarHermDrcho(n, ‘a’);

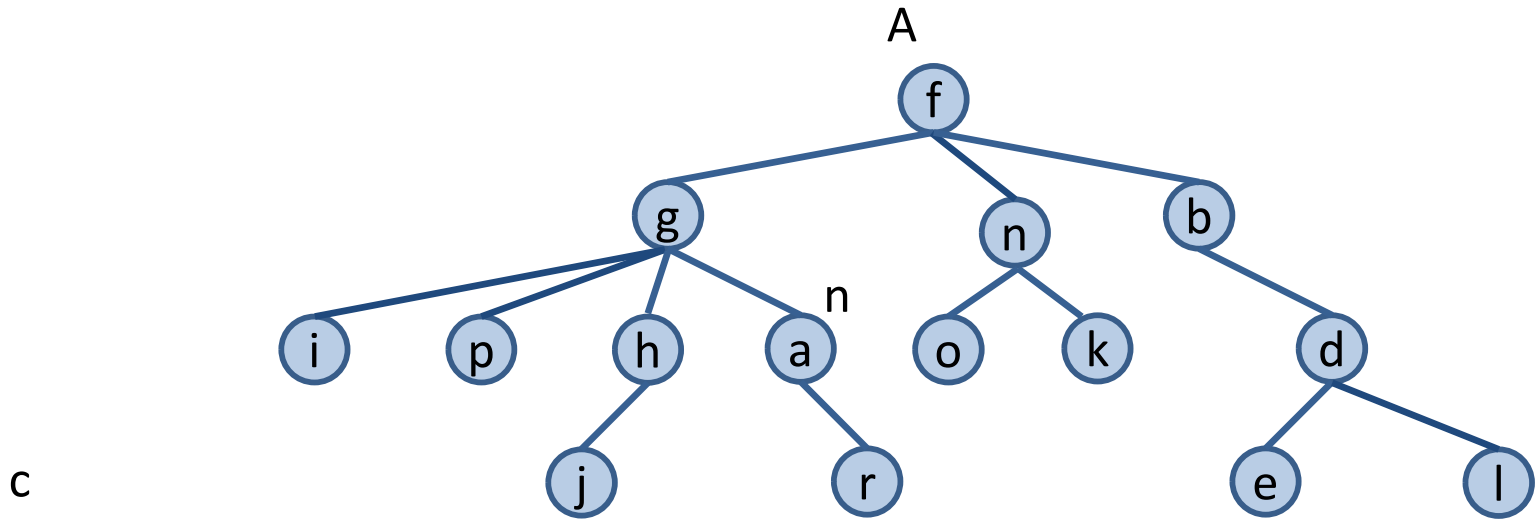
Construcción de un árbol general



r c

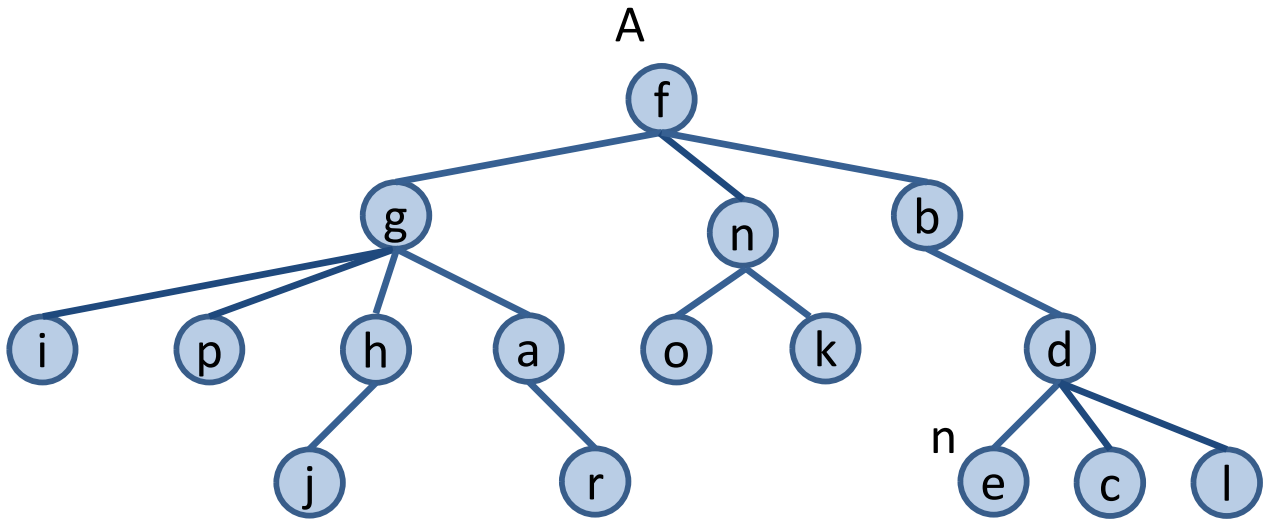
A.insertarHermDrcho(n, ‘l’);

Construcción de un árbol general



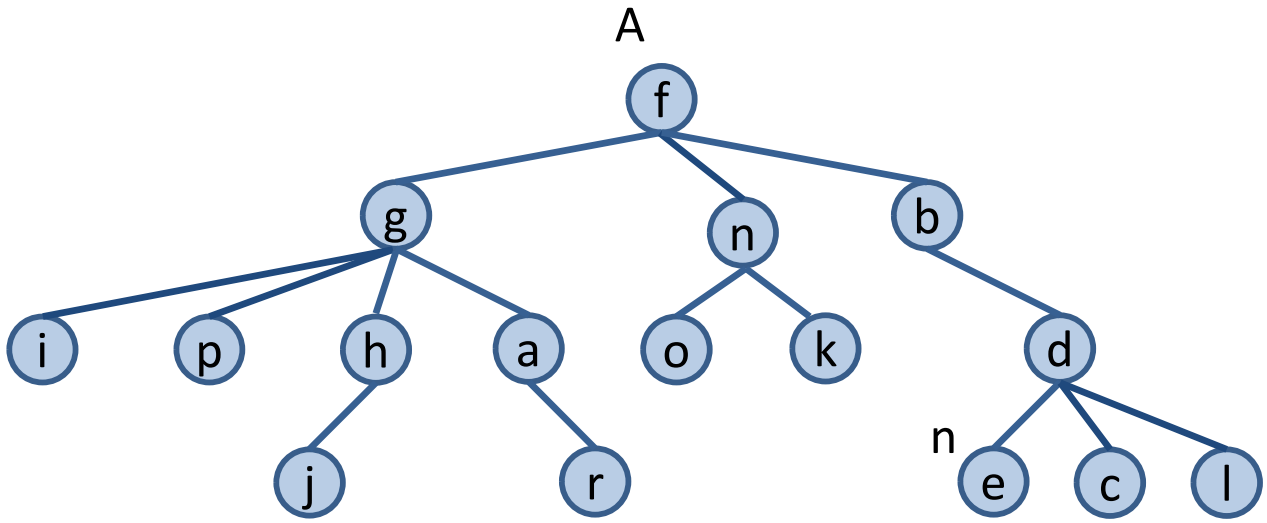
A.insertarHijoIzqdo(n, ‘r’);

Construcción de un árbol general



A.insertarHermDrcho(n, ‘c’);

Construcción de un árbol general



Especificación de operaciones:

Agen()

Post: Construye un árbol vacío.

void insertarRaiz (const T& e)

Pre: El árbol está vacío.

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

void insertarHijoIzqdo(nodo n, const T& e)

Pre: *n* es un nodo del árbol.

Post: Inserta el elemento *e* como hijo izquierdo del nodo *n*. Si ya existe hijo izquierdo, éste se convierte en el hermano derecho del nuevo nodo.

void insertarHermDrcho(nodo n, const T& e)

Pre: *n* es un nodo del árbol *y* no es el nodo raíz.

Post: Inserta el elemento *e* como hermano derecho del nodo *n* del árbol. Si ya existe hermano derecho, éste se convierte en el hermano derecho del nuevo nodo.

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*. El segundo hijo, si existe, se convierte en el nuevo hijo izquierdo de *n*.

void eliminarHermDrcho(nodo n)

Pre: *n* es un nodo del árbol. Existe *hermDrcho(n)* y es una hoja.
Post: Destruye el hermano derecho del nodo *n*. El siguiente hermano se convierte en el nuevo hermano derecho de *n*.

void eliminarRaiz()

Pre: El árbol no está vacío y *raiz()* es una hoja.
Post: Destruye el nodo raíz. El árbol queda vacío.

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 raiz() 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.

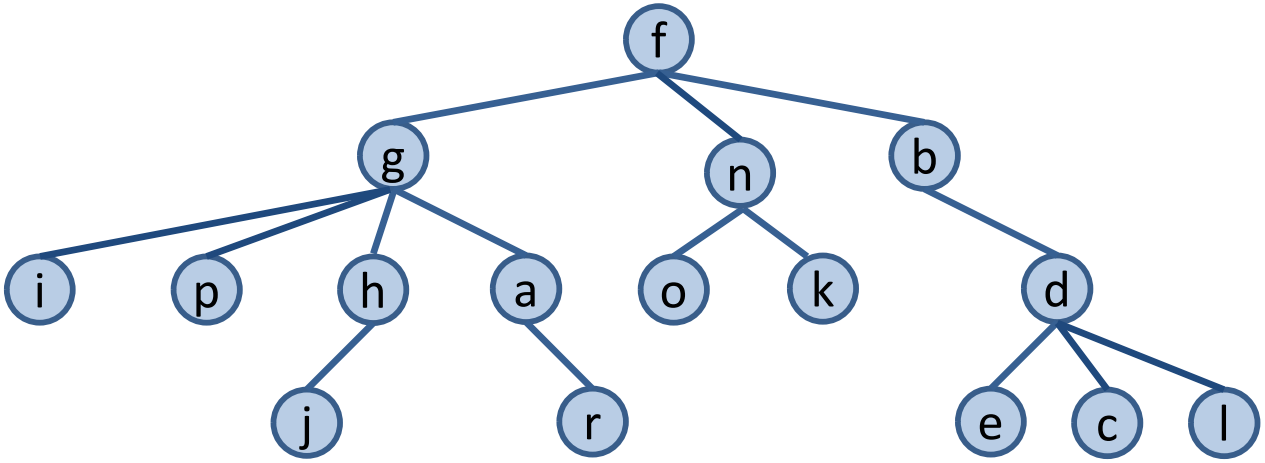
Post: Devuelve el hijo izquierdo del nodo *n*. Si no existe, devuelve *NODO_NULO*.

nodo hermDrcho(nodo n) const

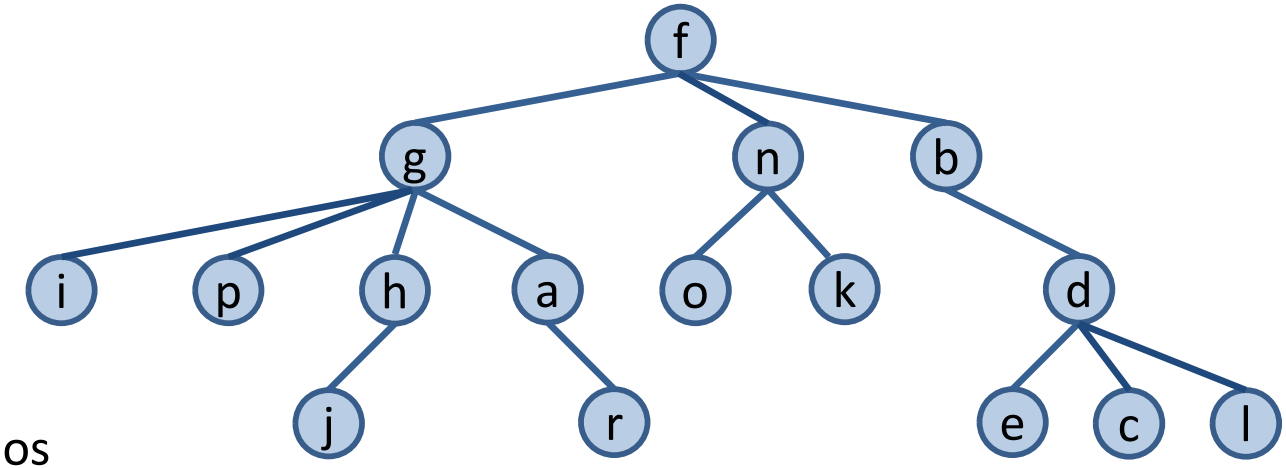
Pre: *n* es un nodo del árbol.

Post: Devuelve el hermano derecho del nodo *n*. Si no existe, devuelve N*ODO_NULO*.

Implementación vectorial de árboles generales mediante listas de hijos

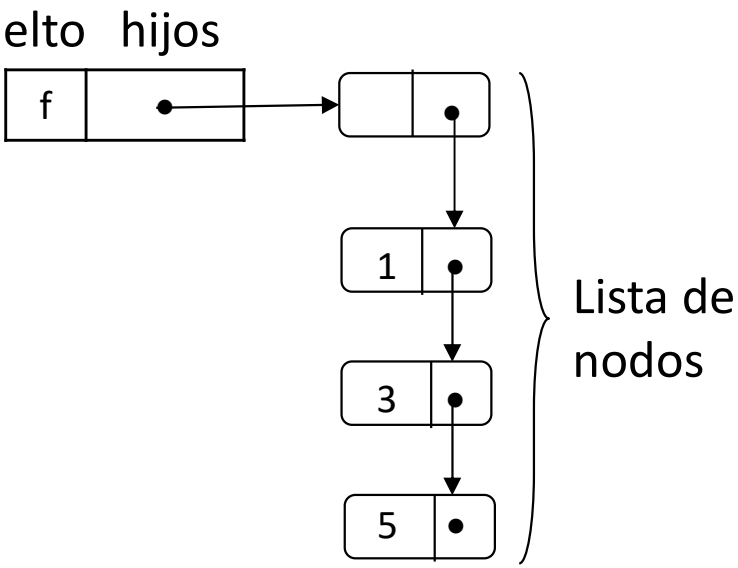


Implementación vectorial de árboles generales mediante listas de hijos

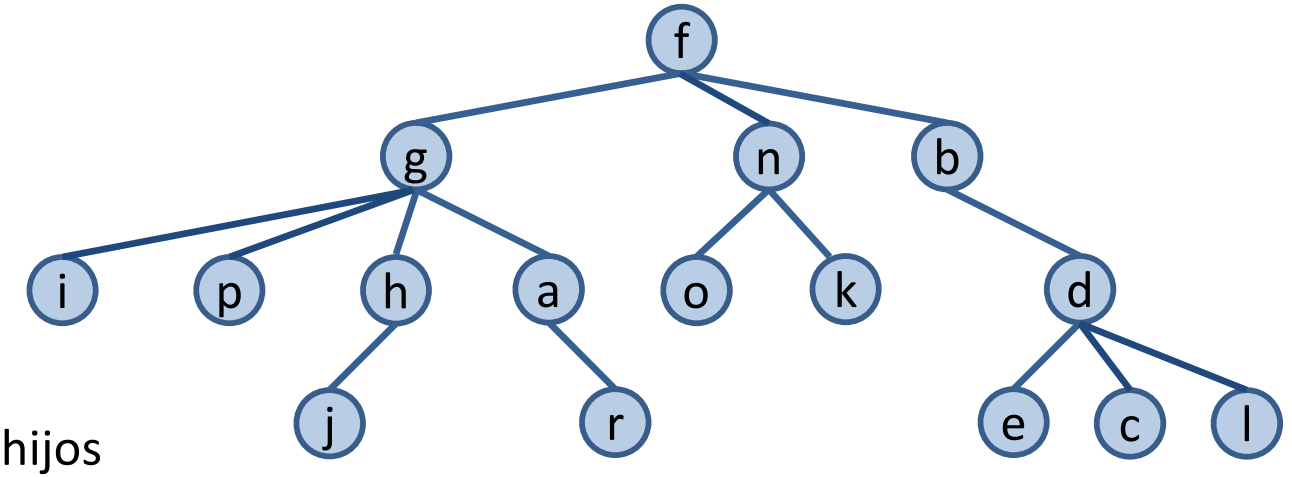


	elto	hijos
0	f	(1,3,5)
1	g	(7,14,2,8)
2	h	(9)
3	n	(4,10)
4	o	( )
5	b	(6)
6	d	(11,12,13)
..		
99	—	—

nodos	
numNodos	16
maxNodos	100



Implementación vectorial de árboles generales mediante listas de hijos

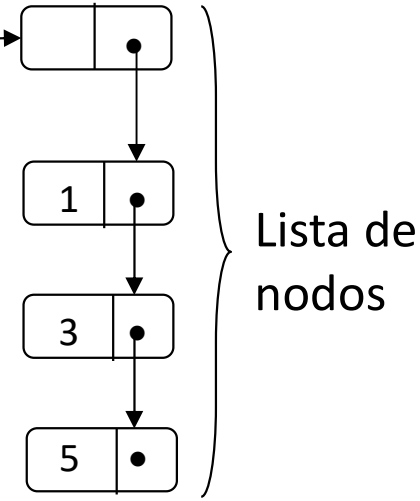


	elto	padre	hijos
0	f	*	(1,3,5)
1	g	0	(7,14,2,8)
2	h	1	(9)
3	n	0	(4,10)
4	o	3	( )
5	b	0	(6)
6	d	5	(11,12,13)
..			
99	—	*	—

nodos	
numNodos	16
maxNodos	100

elto padre hijos

f	*	



```
#ifndef AGEN_LIS_H
#define AGEN_LIS_H
#include <cassert>
#include "listaenla.h"

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

   explicit Agen(size_t maxNodos);     // Ctor., requiere ctor. T()
   void insertarRaiz(const T& e);
   void insertarHijoIzqdo(nodo n, const T& e);
   void insertarHermDrcho(nodo n, const T& e);
   void eliminarHijoIzqdo(nodo n);
   void eliminarHermDrcho(nodo n);
   void eliminarRaiz();
   const T& elemento(nodo n) const;        // Lectura en Agen const
   T& elemento(nodo n);       // Lectura/escritura en Agen no-const
```

```
    nodo raiz() const;
    nodo padre(nodo n) const;
    nodo hijoIzqdo(nodo n) const;
    nodo hermDrcho(nodo n) const;
    Agen(const Agen<T>& A);                    // Ctor. de copia
    Agen<T>& operator =(const Agen<T>& A); // Asignación de árboles
    ~Agen();                                   // Destructor
private:
    struct celda {
        T elto;
        nodo padre;
        Lista<nodo> hijos;
    };
    celda *nodos;       // Vector de nodos
    size_t maxNodos;    // Tamaño del vector
    size_t numNodos;    // Número de nodos del árbol
};

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

```
template <typename T>
inline Agen<T>::Agen(size_t maxNodos) :
   nodos(new celda[maxNodos]),  // Se crean las listas de
                                // hijos vacías.
   maxNodos(maxNodos),
   numNodos(0)
{
   // Marcar todas las celdas como libres.
   for (nodo i = 0; i <= maxNodos-1; i++)
      nodos[i].padre = NODO_NULO;
}

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

   numNodos = 1;
   nodos[0].elto = e;
   // La lista de hijos está vacía.
}
```

```
template <typename T>
void Agen<T>::insertarHijoIzqdo(nodo n, const T& e)
{
   nodo hizqdo;

   assert(numNodos > 0);                    // Árbol no vacío.
   assert(n >= 0 && n <= maxNodos-1);       // n es una celda del vector
   assert(n == 0 || nodos[n].padre != NODO_NULO); // que está ocupada.
   assert(numNodos < maxNodos);    // Árbol no lleno.

   // Añadir el nuevo nodo en la primera celda libre.
   for (hizqdo = 1; nodos[hizqdo].padre != NODO_NULO; hizqdo++);
   nodos[hizqdo].elto  = e;
   nodos[hizqdo].padre = n;
   // Insertar el nuevo nodo al inicio de la lista de hijos de n.
   Lista<nodo>& Lh = nodos[n].hijos; // Lista de hijos.
   Lh.insertar(hizqdo, Lh.primera());
   ++numNodos;
}
```

```
template <typename T>
void Agen<T>::insertarHermDrcho(nodo n, const T& e)
{
   nodo hedrcho;

   assert(n >= 0 && n <= maxNodos-1);   // n es un nodo válido.
   assert(nodos[n].padre != NODO_NULO); // n existe y no es la raíz.
   assert(numNodos < maxNodos);         // Árbol no lleno.

   // Añadir el nuevo nodo en la primera celda libre.
   for (hedrcho = 1; nodos[hedrcho].padre != NODO_NULO; hedrcho++);
   nodos[hedrcho].elto  = e;
   nodos[hedrcho].padre = nodos[n].padre;
   // Insertar el nuevo nodo en la lista de hijos del padre
   // en la posición siguiente a la de n.
   Lista<nodo>& Lhp = nodos[nodos[n].padre].hijos; // Lista de hijos
                                                   // del padre.
   Lista<nodo>::posicion p = Lhp.primera();
   while (n != Lhp.elemento(p)) p = Lhp.siguiente(p);
   Lhp.insertar(hedrcho, Lhp.siguiente(p));
   ++numNodos;
}
```

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

   assert(numNodos > 0);                    // Árbol no vacío.
   assert(n >= 0 && n <= maxNodos-1);       // n es una celda del vector
   assert(n == 0 || nodos[n].padre != NODO_NULO); // que está ocupada.
   Lista<nodo>& Lh = nodos[n].hijos; // Lista de hijos.
   assert(Lh.primera() != Lh.fin()); // Lista no vacía, n tiene hijos.
   hizqdo = Lh.elemento(Lh.primera());
   assert(nodos[hizqdo].hijos.primera() ==  // Lista vacía, hijo izq.
          nodos[hizqdo].hijos.fin());       // de n es una hoja.

   // Eliminar hijo izqdo. de n.
   nodos[hizqdo].padre = NODO_NULO; // Marcar celda libre.
   Lh.eliminar(lh.primera());       // Eliminar primer nodo de la
                                    // lista de hijos de n.
   --numNodos;
}
```

Text

```
template <typename T>
void Agen<T>::eliminarHermDrcho(nodo n)
{
   nodo hdrcho;
   Lista<nodo>::posicion p;

   assert(n >= 0 && n <= maxNodos-1);   // n es un nodo válido.
   assert(nodos[n].padre != NODO_NULO); // n existe y no es la raíz.
   // Buscar hermano drcho. de n en la lista de hijos del padre.
   Lista<nodo>& Lhp = nodos[nodos[n].padre].hijos; // Lista de hijos
                                                   // del padre.
   Lista<nodo>::posicion p = Lhp.primera();
   while (n != Lhp.elemento(p)) p = Lhp.siguiente(p);
   p = Lhp.siguiente(p);
   assert(p != Lhp.fin());   // n tiene hermano drcho.
   hdrcho = Lhp.elemento(p);
   assert(nodos[hdrcho].hijos.primera() ==  // Lista vacía, hermano
          nodos[hdrcho].hijos.fin());       // drcho. de n es hoja.

   // Eliminar hermano drcho. de n.
   nodos[hdrcho].padre = NODO_NULO; // Marcar celda libre.
   Lhp.eliminar(p);                 // Eliminar hermano de la lista
                                    // de hijos del padre.
   --numNodos;
}
```

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

template <typename T>
inline const T& Agen<T>::elemento(nodo n) const
{
   assert(numNodos > 0);                    // Árbol no vacío.
   assert(n >= 0 && n <= maxNodos-1);       // n es una celda del vector
   assert(n == 0 || nodos[n].padre != NODO_NULO); // que está ocupada
   return nodos[n].elto;
}

template <typename T>
inline T& Agen<T>::elemento(nodo n)
{
   assert(numNodos > 0);                    // Árbol no vacío.
   assert(n >= 0 && n <= maxNodos-1);       // n es una celda del vector
   assert(n == 0 || nodos[n].padre != NODO_NULO); // que está ocupada
   return nodos[n].elto;
}
```

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

template <typename T>
inline typename Agen<T>::nodo Agen<T>::padre(nodo n) const
{
   assert(numNodos > 0);                    // Árbol no vacío.
   assert(n >= 0 && n <= maxNodos-1);       // n es una celda del vector
   assert(n == 0 || nodos[n].padre != NODO_NULO); // que está ocupada

   return nodos[n].padre;
}
```

```
template <typename T>
inline typename Agen<T>::nodo Agen<T>::hijoIzqdo(nodo n) const
{
   assert(numNodos > 0);                    // Árbol no vacío.
   assert(n >= 0 && n <= maxNodos-1);     // n es una celda del vector
   assert(n == 0 || nodos[n].padre != NODO_NULO); // que está ocupada.

   Lista<nodo>& Lh = nodos[n].hijos;
   if (Lh.primera() != Lh.fin())   // Lista no vacía.
      return Lh.elemento(Lh.primera());
   else
      return NODO_NULO;
}
```

```
template <typename T>
inline typename Agen<T>::nodo Agen<T>::hermDrcho(nodo n) const
{
   Lista<nodo>::posicion p;

   assert(numNodos > 0);                    // Árbol no vacío.
   assert(n >= 0 && n <= maxNodos-1);       // n es una celda del vector
   assert(n == 0 || nodos[n].padre != NODO_NULO); // que está ocupada.

   if (n == 0)    // n es la raíz.
      return NODO_NULO;
   else
   {
      Lista<nodo>& Lhp = nodos[nodos[n].padre].hijos; // Lista de hijos
                                                      // del padre.
      Lista<nodo>::posicion p = Lhp.primera();
      while (n != Lhp.elemento(p)) p = Lhp.siguiente(p);
      p = Lhp.siguiente(p);
      return p != Lhp.fin() ?   // n tiene hermano drcho.
         Lhp.elemento(p) : NODO_NULO;
   }
}
```

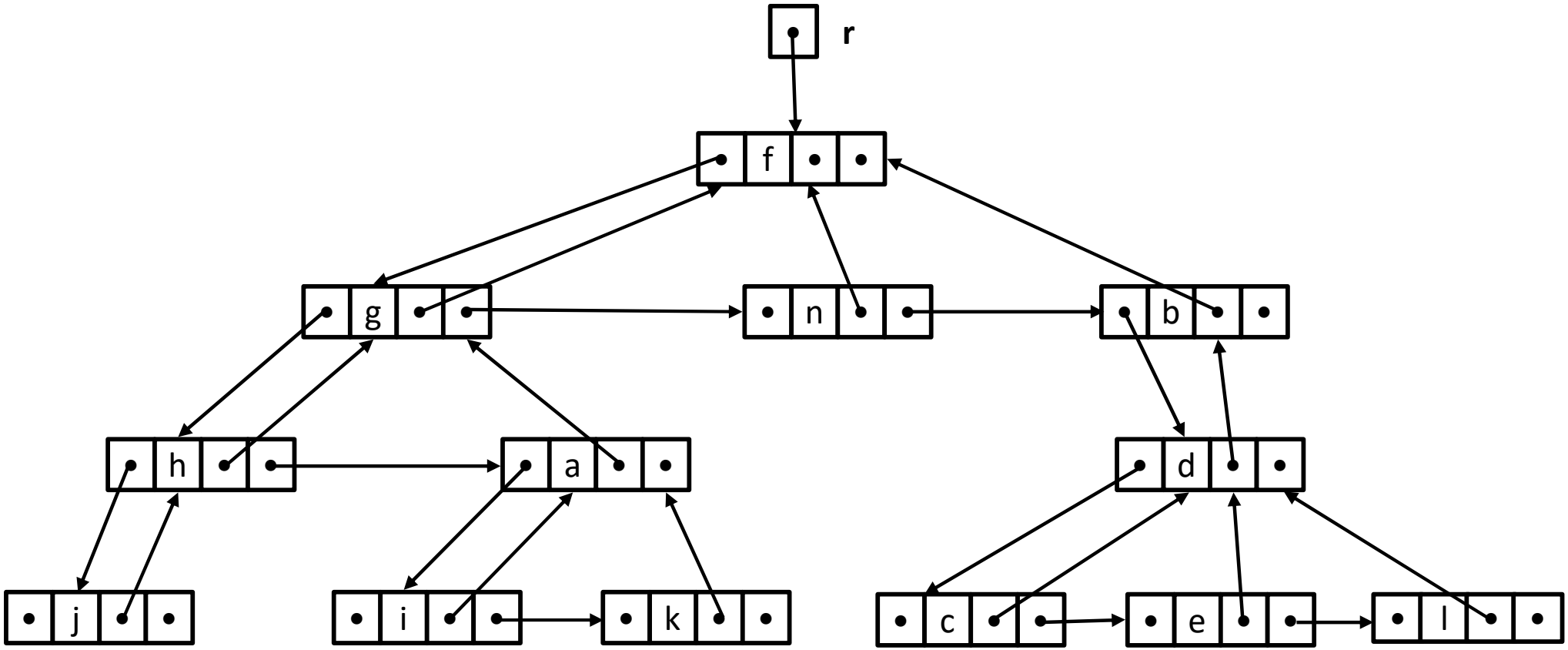
```
template <typename T>
Agen<T>::Agen(const Agen<T>& A) :
   nodos(new celda[A.maxNodos]),
   maxNodos(A.maxNodos), numNodos(A.numNodos)
{
   for (nodo n = 0; n <= maxNodos-1; n++) // Copiar el vector.
      nodos[n] = A.nodos[n];
}

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

```
template <typename T>
inline Agen<T>::~Agen()
{
   delete[] nodos; // También destruye las listas de hijos.
}

#endif // AGEN_LIS_H
```

Implementación de un árbol general usando celdas enlazadas



```
#ifndef AGEN_H
#define AGEN_H
#include <cassert>

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

   Agen();                                    // Constructor
   void insertarRaiz(const T& e);
   void insertarHijoIzqdo(nodo n, const T& e);
   void insertarHermDrcho(nodo n, const T& e);
   void eliminarHijoIzqdo(nodo n);
   void eliminarHermDrcho(nodo n);
   void eliminarRaiz();
   const T& elemento(nodo n) const;          // Lectura en Agen const
   T& elemento(nodo n);        // Lectura/escritura en Agen no-const
   nodo raiz() const;
   nodo padre(nodo n) const;
   nodo hijoIzqdo(nodo n) const;
   nodo hermDrcho(nodo n) const;
```

```
   Agen(const Agen<T>& a);                    // Ctor. de copia
   Agen<T>& operator =(const Agen<T>& a); // Asignación de árboles
   ~Agen();                                   // Destructor
private:
   struct celda {
      T elto;
      nodo padre, hizq, heder;
      celda(const T& e, nodo p = NODO_NULO): elto(e),
         padre(p), hizq(NODO_NULO), heder(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 Agen<T>::nodo Agen<T>::NODO_NULO(nullptr);
```

```
/*--------------------------------------------------------*/
/* Métodos públicos                                       */
/*--------------------------------------------------------*/

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

template <typename T>
inline void Agen<T>::insertarRaiz(const T& e)
{
   assert(r == NODO_NULO);   // Árbol vacío.

   r = new celda(e);
}
```

```
template <typename T>
inline void Agen<T>::insertarHijoIzqdo(nodo n, const T& e)
{
   assert(n != NODO_NULO);

   nodo hizqdo = n->hizq;    // Hijo izqdo. actual.
   n->hizq = new celda(e, n);
   n->hizq->heder = hizqdo; // El actual hijo izqdo. se convierte en
                            // hermano drcho. del nuevo hijo.
}

template <typename T>
inline void Agen<T>::insertarHermDrcho(nodo n, const T& e)
{
   assert(n != NODO_NULO);
   assert(n != r);    // n no es la raíz.

   nodo hedrcho = n->heder;    // Hermano drcho. actual
   n->heder = new celda(e, n->padre);
   n->heder->heder = hedrcho; // El actual hermano drcho. se convierte
                              // en hermano drcho. del nuevo.
}
```

```
template <typename T>
inline void Agen<T>::eliminarHijoIzqdo(nodo n)
{
   nodo hizqdo;

   assert(n != NODO_NULO);
   hizqdo = n->hizq;    Si no hay segundo hijo, ese hizq es nodo nulo
   assert(hizqdo != NODO_NULO);          // Existe hijo izqdo.
   assert(hizqdo->hizq == NODO_NULO); // Hijo izqdo. es hoja.

   // El hermano drcho. pasa a ser el nuevo hijo izqdo.
   n->hizq = hizqdo->heder;
   delete hizqdo;
}
```

```
template <typename T>
inline void Agen<T>::eliminarHermDrcho(nodo n)
{
   nodo hedrcho;

   assert(n != NODO_NULO);
   hedrcho = n->heder;
   assert(hedrcho != NODO_NULO);    // Existe hermano drcho.
   assert(hedrcho->hizq == NODO_NULO); // Hermano drcho. es hoja.

   // El hermano del hermano se convierte en el
   // nuevo hermano drcho. de n.
   n->heder = hedrcho->heder;
   delete hedrcho;
}
```

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

   delete(r);Liberamos la memoria
   r = NODO_NULO; Indicamos que el nodo está vacío
}
```

```
template <typename T>
inline const T& Agen<T>::elemento(nodo n) const
{
   assert(n != NODO_NULO);

   return n->elto;
}

template <typename T>
inline T& Agen<T>::elemento(nodo n)
{
   assert(n != NODO_NULO);

   return n->elto;
}
```

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

template <typename T> inline
typename Agen<T>::nodo Agen<T>::padre(nodo n) const
{
   assert(n != NODO_NULO);

   return n->padre;
}
```

```
template <typename T> inline
typename Agen<T>::nodo Agen<T>::hijoIzqdo(nodo n) const
{
   assert(n != NODO_NULO);

   return n->hizq;
}

template <typename T> inline
typename Agen<T>::nodo Agen<T>::hermDrcho(nodo n) const
{
   assert(n != NODO_NULO);

   return n->heder;
}
```

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

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

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

```
/*-------------------------------------------------------*/
/* Métodos privados                                      */
/*-------------------------------------------------------*/
// Destruye un nodo y todos sus descendientes
template <typename T>
void Agen<T>::destruirNodos(nodo& n)
{
   if (n != NODO_NULO)
   {
      if (n->hizq != NODO_NULO)
      {  // Destruir hermanos del hijo izqdo.
         nodo hedrcho = n->hizq->heder;
         while (hedrcho != NODO_NULO)
         {
            n->hizq->heder = hedrcho->heder;
            destruirNodos(hedrcho);
            hedrcho = n->hizq->heder;
         }
         destruirNodos(n->hizq); // Destruir el hijo izqdo.
      }
      delete(n);
      n = NODO_NULO;
   }
}
```

Suponemos que n es una hoja, ya que al igual que en los árboles binarios para destruir un nodo éste debe de ser una hoja.

Si eres una hoja te elimino y si no, vamos a por tus hijos.
Cuando eliminamos a los hijos, eliminamos el primogénito (n)

El orden: Destruir hijos y luego el padre (n)

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

   if (n != NODO_NULO) {
      m = new celda(n->elto);          // Copiar n.
      if (n->hizq != NODO_NULO) {      // n tiene descendientes.
         m->hizq = copiar(n->hizq); // Copiar primer subárbol.
         m->hizq->padre = m;
         // Copiar el resto de subárboles.
         nodo hijo = m->hizq;              // Último subárbol copiado.
         nodo hedrcho = n->hizq->heder; // Siguiente subárbol a copiar
         while (hedrcho != NODO_NULO) {
            hijo = hijo->heder = copiar(hedrcho);
            hijo->padre = m;
            hedrcho = hedrcho->heder;
         }
      }
   }
   return m;
}

#endif // AGEN_H
```

El padre del hijo izquierdo de n es el propio nodo n

Los árboles servían para poder establecer jerarquías y en algunos casos poder realizar búsquedas en un Orden menor a O(n), aprox en O(log n).

Cosa que en las listas no podemos hacer.