TEMA 5: ÁRBOLES

1. INTRODUCCIÓN

En la vida real nos encontramos con situaciones en las que la organización de los datos no es fácilmente representable mediante las estructuras de datos lineales (pilas, colas y listas). Estamos hablando de situaciones en las que los diferentes elementos involucrados no se encuentran al mismo nivel, sino organizados en una jerarquía de diferentes capas o niveles, perdiéndose por tanto el concepto de secuencialidad que aparece en todas las estructuras de datos lineales. Necesitamos por tanto una estructura de datos que nos permita representar la **jerarquía**, concepto que aparece habitualmente en el mundo real.

Por otra parte, existen una serie de problemas, que aunque son representables mediante estructuras de datos lineales, no son computables en un tiempo razonable en la práctica mediante el uso de dichas estructuras. Nos estamos refiriendo, por ejemplo, al problema de la **búsqueda** de un elemento en una colección de los mismos. Los algoritmos conocidos sobre estructuras de datos lineales son de orden n (tamaño o longitud de la lista). Es muy habitual, en aplicaciones informáticas del mundo real, tener que procesar colecciones de millones e incluso de miles de millones de elementos. Si el tiempo necesario para realizar una búsqueda fuera proporcional al tamaño de la misma, sería impracticable.

Existe una estructura de datos que nos permite enfrentarnos a los dos problemas que nos han surgido, a saber, potencia representativa (ser capaces de representar jerarquías) y eficiencia en las búsquedas (realizar búsquedas no ya en orden lineal, sino logarítmico): Esta estructura de datos recibe el nombre de **árbol**.

2. CONCEPTOS BÁSICOS

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

Una definición recursiva más formal es la siguiente:

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

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

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

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

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

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

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

Hoja: Nodo que no tiene descendientes propios.

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

Rama: Camino que termina en un nodo hoja.

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

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

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

3. 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.*

Especificación

Abin CrearABin ()

Postcondiciones: Crea y devuelve un árbol vacío.

void CrearRaizB (tElemento e, Abin A)

Precondiciones: *A* es el árbol vacío.

Postcondiciones: Crea el nodo raíz de *A* cuyo contenido es *e*.

void InsertarHijoIzqdoB (nodo n, tElemento e, Abin A)

Precondiciones: *n* es un nodo de *A* y no tiene hijo izquierdo.

Postcondiciones: Inserta el elemento *e* como hijo izquierdo del nodo *n* del árbol *A*.

void InsertarHijoDrchoB (nodo n, tElemento e, Abin A)

Precondiciones: *n* es un nodo de *A* y no tiene hijo derecho.

Postcondiciones: Inserta el elemento *e* como hijo derecho del nodo *n* del árbol *A*.

void EliminarHijoIzqdoB (nodo n, Abin A)

Precondiciones: *n* es un nodo de *A*. Existe *HijoIzqdo(n)* y es una hoja.

Postcondiciones: Destruye el hijo izquierdo del nodo *n* en el árbol *A*.

void EliminarHijoDrchoB (nodo n, Abin A)

Precondiciones: *n* es un nodo de *A*. Existe *HijoDrcho(n)* y es una hoja.

Postcondiciones: Destruye el hijo derecho del nodo *n* en el árbol *A*.

void EliminarRaizB (Abin A)

Precondiciones: *A* no está vacío y *Raiz(A)* es una hoja.

Postcondiciones: Destruye el nodo raíz del árbol *A*.

void DestruirAbin (Abin A)

Postcondiciones: Libera la memoria ocupada por el árbol *A*. Para usarlo otra vez se debe volver a crear con la operación *CrearAbin()*.

int ArbolVacioB (Abin A)

Postcondiciones: Devuelve 0 si A no es el árbol vacío y 1 en caso contrario.

tElemento RecuperarB (nodo n, Abin A)

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

Postcondiciones: Devuelve el elemento del nodo *n*.

void ModificarB (nodo n, tElemento e, Abin A)

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

Postcondiciones: Modifica el elemento del nodo *n* con *e*.

nodo RaízB (Abin A)

Postcondiciones: Devuelve el nodo raíz de *A*. Si *A* está vacío, devuelve *NODO_NULO*.

nodo PadreB (nodo n, Abin A)

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

Postcondiciones: Devuelve el padre del nodo *n*. Si *n* es el nodo raíz, devuelve *NODO_NULO*.

nodo HijoIzqdoB (nodo n, Abin A)

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

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

nodo HijoDrchoB (nodo n, Abin A)

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

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

3.1 Implementación vectorial de árboles binarios

Los nodos de un árbol podrían almacenarse en un vector de registros con dos campos, uno para el elemento del nodo y otro para almacenar el índice del vector en el que se encuentra el nodo padre. El nodo raíz puede distinguirse con el valor NODO_NULO como índice del nodo padre. Con esta representación, la operación para localizar el padre de un nodo dado es eficiente (O(1)) y fácil de implementar. Sin embargo, no es adecuada para las operaciones en las que haya que acceder a los hijos de un nodo n, ya que es necesario rastrear el vector para encontrar los nodos cuyo padre sea n; pero, además, no es posible distinguir entre el hijo izquierdo y el derecho. Podemos añadir dos campos más a los registros para almacenar los índices de los hijos y utilizar el valor NODO_NULO para indicar la inexistencia del hijo correspondiente. De esta forma, tanto el padre como los hijos de un nodo pueden encontrarse en un tiempo constante y es fácil moverse por el árbol hacia arriba y hacia abajo.

Con esta estructura de datos debemos tener en cuenta que al añadir un nuevo nodo al árbol es necesario utilizar una posición del vector que no esté ocupada ya por algún otro nodo del árbol. Por otra parte, al suprimir un nodo se libera una celda del vector, que puede ser ocupada en una posterior inserción. Esto quiere decir que de alguna forma hay que representar el estado de cada celda del vector. Se puede hacer basándonos en la propiedad de los árboles de que cada nodo, excepto la raíz, tiene un padre; es decir, si el padre de un nodo es NODO_NULO, la celda está vacía y en caso contrario está ocupada, pero si el valor del campo padre es el índice de la propia celda, entonces en ella se encuentra el nodo raíz del árbol. Ahora, para insertar un nuevo nodo hay que localizar una celda cuyo padre sea NODO_NULO y al eliminarlo hay que marcar la celda como libre colocando NODO_NULO en el campo padre. Las operaciones de inserción, por tanto, son poco eficientes puesto que hay que recorrer el vector.

Para mejorar el tiempo de inserción podemos adoptar la estrategia de tener siempre los nodos del árbol en las primeras celdas del vector y dejar todas las celdas libres al final del mismo. Sabiendo el número de nodos del árbol o la posición de la primera celda libre, no es necesario recorrer el vector para encontrar un hueco y la inserción de un nodo se puede hacer en tiempo constante. A cambio, la eliminación de un nodo consume un poco más de tiempo, ya que hay que cubrir el hueco moviendo el último nodo almacenado en el vector. Además, de esta forma, ya no tenemos que hacer distinción entre celdas ocupadas y libres, ya que éstas están al final del vector, por lo que el nodo raíz se puede distinguir del resto porque su padre es NODO_NULO.

```
/* AbinMat0.h                              */

#define NODO_NULO -1
typedef char tElemento  /* Por ejemplo */

#ifndef _ARBOL_BINARIO_
#define _ARBOL_BINARIO_
  typedef int nodo; /* índice de la matriz
                       entre 0 y maxNodos-1 */
  struct celda {
          tElemento elto;
          nodo padre, hizq, hder;
     };

  typedef struct {
                 struct celda *nodos;
                 int maxNodos;
                 int numNodos;
               } tipoArbol;
  typedef tipoArbol *Abin;

  Abin CrearAbin (int maxNodos);
  void CrearRaizB (tElemento e, Abin A);
  void InsertarHijoIzqdoB(nodo n, tElemento e, Abin A);
  void InsertarHijoDrchoB(nodo n, tElemento e, Abin A);
  void EliminarHijoIzqdoB (nodo n, Abin A);
  void EliminarHijoDrchoB (nodo n, Abin A);
  void EliminarRaizB (Abin A);
  void DestruirAbin (Abin A);
  int ArbolVacioB (Abin A);
  tElemento RecuperarB (nodo n, Abin A);
  void ModificarB (nodo n, tElemento e, Abin A);
  nodo RaizB (Abin A);
```

```
  nodo PadreB (nodo n, Abin A);
  nodo HijoIzqdoB (nodo n, Abin A);
  nodo HijoDrchoB (nodo n, Abin A);
#endif



/* AbinMat0.c                              */

#include <stdlib.h>
#include "error.h"
#include "abinmat0.h"

Abin CrearAbin (int maxNodos)
{
     Abin A;

     A = (Abin) malloc(sizeof(tipoArbol));
     if (!A)
          ERROR("CrearAbin(): No hay memoria");
     A->nodos = (struct celda *) calloc(maxNodos,
          sizeof(struct celda));
     if (!A->nodos)
          ERROR("CrearAbin(): No hay memoria");
     A->maxNodos = maxNodos;
     A->numNodos = 0;
     return A;
}

void CrearRaizB (tElemento e, Abin A)
{
     if (A->numNodos)
          ERROR("CrearRaizB(): Árbol no vacío");
```

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

void InsertarHijoIzqdoB (nodo n, tElemento e, Abin A)
{
     nodo hizqdo;

     if (n < 0 || n > A->numNodos - 1)
       ERROR("InsertarHijoIzqdoB(): Nodo inexistente");
     if (A->nodos[n].hizq != NODO_NULO)
       ERROR("InsertarHijoIzqdoB(): Ya existe
                hijo izquierdo");
     if (A->numNodos == A->maxNodos)
       ERROR("InsertarHijoIzqdoB(): Espacio
                insuficiente para hijo izqdo");
/* Añadir el nuevo nodo al final de la secuencia */
     hizqdo = A->numNodos;
     A->numNodos++;
     A->nodos[n].hizq = hizqdo;
     A->nodos[hizqdo].elto = e;
     A->nodos[hizqdo].padre = n;
     A->nodos[hizqdo].hizq = NODO_NULO;
     A->nodos[hizqdo].hder = NODO_NULO;
}
void InsertarHijoDrchoB (nodo n, tElemento e, Abin A)
{
     nodo hdrcho;

     if (n < 0 || n > A->numNodos - 1)
          ERROR(”...: Nodo inexistente");
```

```
     if (A->nodos[n].hder != NODO_NULO)
          ERROR(”...: Ya existe hijo derecho");
     if (A->numNodos == A->maxNodos)
          ERROR(”...: Espacio insuficiente ...");

/* Añadir el nuevo nodo al final de la secuencia */
     hdrcho = A->numNodos;
     A->numNodos++;
     A->nodos[n].hder = hdrcho;
     A->nodos[hdrcho].elto = e;
     A->nodos[hdrcho].padre = n;
     A->nodos[hdrcho].hizq = NODO_NULO;
     A->nodos[hdrcho].hder = NODO_NULO;
}
void EliminarHijoIzqdoB (nodo n, Abin A)
{
     nodo hizqdo;

     if (n < 0 || n > A->numNodos - 1)
          ERROR(”...: Nodo inexistente");
     hizqdo = A->nodos[n].hizq;
     if (hizqdo == NODO_NULO)
          ERROR(”... : No existe hijo izqdo.");
     if (A->nodos[hizqdo].hizq != NODO_NULO ||
           A->nodos[hizqdo].hder != NODO_NULO)
          ERROR(”...: Hijo izqdo no es hoja");

     if (hizqdo != A->numNodos-1) {
       A->nodos[hizqdo]=A->nodos[A->numNodos-1];
       if (A->nodos[A->nodos[hizqdo].padre].hizq
                == A->numNodos-1)
          /* El nodo movido es hijo izq. de su padre */
          A->nodos[A->nodos[hizqdo].padre].hizq=hizqdo;
```

```
       else
          /* El nodo movido es hijo der. de su padre */
         A->nodos[A->nodos[hizqdo].padre].hder=hizqdo;
       if (A->nodos[hizqdo].hizq != NODO_NULO)
          /* El nodo movido tiene hijo izq. */
         A->nodos[A->nodos[hizqdo].hizq].padre=hizqdo;
       if (A->nodos[hizqdo].hder != NODO_NULO)
          /* El nodo movido tiene hijo der. */
         A->nodos[A->nodos[hizqdo].hder].padre=hizqdo;
     }
     A->numNodos--;
     A->nodos[n].hizq = NODO_NULO;
}

void EliminarHijoDrchoB (nodo n, Abin A)
{
     nodo hdrcho;
     /* Comprobar precondiciones */
     ...
     if (hdrcho != A->numNodos-1) {
       A->nodos[hdrcho] = A->nodos[A->numNodos-1];
       if (A->nodos[A->nodos[hdrcho].padre].hizq
              == A->numNodos-1)
          A->nodos[A->nodos[hdrcho].padre].hizq=hdrcho;
       else
          A->nodos[A->nodos[hdrcho].padre].hder=hdrcho;
       if (A->nodos[hdrcho].hizq != NODO_NULO)
          A->nodos[A->nodos[hdrcho].hizq].padre=hdrcho;
       if (A->nodos[hdrcho].hder != NODO_NULO)
          A->nodos[A->nodos[hdrcho].hder].padre=hdrcho;
     }
     A->numNodos--;
     A->nodos[n].hder = NODO_NULO;
}
```

```
void EliminarRaizB (Abin A)
{
     if (!A->numNodos)
          ERROR("EliminarRaizB(): Árbol vacío");
     if (A->nodos[0].hizq != NODO_NULO ||
           A->nodos[0].hder != NODO_NULO)
           ERROR("EliminarRaizB(): La raíz no es
                  un nodo hoja");
     A->numNodos = 0;
}

void DestruirAbin (Abin A)
{
     free(A->nodos);
     free(A);
}

int ArbolVacioB (Abin A)
{
     return (A->numNodos == 0);
}

tElemento RecuperarB (nodo n, Abin A)
{
     if (n < 0 || n > A->numNodos - 1)
          ERROR("RecuperarB(): Nodo inexistente");
     return A->nodos[n].elto;
}

void ModificarB (nodo n, tElemento e, Abin A)
{
     if (n < 0 || n > A->numNodos - 1)
          ERROR("ModificarB(): Nodo inexistente");
```

```
   A->nodos[n].elto = e; /* Se asume que el operador =
                            es compatible con tElemento.
                         Si no es así, hay que
                         implementar una función privada
                         para la realizar la copia. */
}
nodo RaizB (Abin A)
{
     if (A->numNodos)
          return 0;
     else
          return NODO_NULO;
}

nodo PadreB (nodo n, Abin A)
{
     if (n < 0 || n > A->numNodos - 1)
          ERROR("Padre(): Nodo inexistente");
     return A->nodos[n].padre;
}

nodo HijoIzqdoB (nodo n, Abin A)
{
     if (n < 0 || n > A->numNodos - 1)
          ERROR("HijoIzqdo(): Nodo inexistente");
     return A->nodos[n].hizq;
}

nodo HijoDrchoB (nodo n, Abin A)
{
     if (n < 0 || n > A->numNodos - 1)
          ERROR("HijoDrcho(): Nodo inexistente");
     return A->nodos[n].hder;
}
```

3.2 Implementación mediante una matriz de posiciones relativas

Es posible ocupar menos espacio para almacenar cada nodo aprovechando el hecho de que el árbol es binario. Podemos utilizar un vector de nodos en el que cada posición guarda solamente el elemento correspondiente y un nodo se almacena en una posición que depende del lugar que ocupa en el árbol. El nodo raíz se coloca en la primera posición del vector; el hijo izquierdo del nodo de la posición *i*, si existe, está en la posición *2i+1*; y el hijo derecho, si existe, está en la posición *2i+2*. Por tanto, el padre del nodo que está en la posición *i*, se encuentra en la posición *(i-1)/2*.

Con esta representación, el tamaño del vector para almacenar un árbol de altura *h* tiene que ser $2^{(h+1)}-1$. Es decir, dicho tamaño vendrá determinado por la longitud de la rama más larga, por lo que interesa que todas las ramas tengan la misma longitud, para que así se aproveche todo el espacio ocupado por el vector. Por tanto, esta estructura es adecuada para almacenar árboles binarios con todos sus niveles llenos, o en su defecto, árboles que tengan completos todos sus niveles excepto el último, pero en los que los nodos estén lo más a la izquierda posible (árboles completos).

Con esta representación es imposible determinar si una celda del vector está ocupada o no, o dicho de otra forma, no podemos saber si el valor almacenado en una posición del vector corresponde a un elemento del árbol. Esto implica que tenemos que marcar cada celda para distinguir si está libre u ocupada. Una posibilidad para no utilizar más espacio es utilizar un valor del tipo de los elementos almacenados en el árbol, que sea ilegal en la aplicación que se está desarrollando. Así, al crear un árbol, inicializamos todas las celdas del vector con este valor ilegal (representando el hecho de que el árbol está vacío). Por otra parte, al eliminar un nodo debemos sustituir el elemento con este valor ilegal.

```
/*            AbinMat1.h          */

#define NODO_NULO -1
typedef char tElemento;  /* Por ejemplo */
#ifndef _ARBOL_BINARIO_
#define _ARBOL_BINARIO_
  typedef int nodo; /* índice de la matriz
                       entre 0 y maxNodos-1 */
  typedef struct {
              tElemento *nodos;
              int maxNodos;
          } tipoArbol;
```

```
  typedef tipoArbol *Abin;
  Abin CrearAbin (int maxNodos);
  void CrearRaizB (tElemento e, Abin A);
  void InsertarHijoIzqdoB(nodo n, tElemento e, Abin A);
  void InsertarHijoDrchoB(nodo n, tElemento e, Abin A);
  void EliminarHijoIzqdoB (nodo n, Abin A);
  void EliminarHijoDrchoB (nodo n, Abin A);
  void EliminarRaizB (Abin A);
  void DestruirAbin (Abin A);
  int ArbolVacioB (Abin A);
  tElemento RecuperarB (nodo n, Abin A);
  void ModificarB (nodo n, tElemento e, Abin A);
  nodo RaizB (Abin A);
  nodo PadreB (nodo n, Abin A);
  nodo HijoIzqdoB (nodo n, Abin A);
  nodo HijoDrchoB (nodo n, Abin A);
#endif

/* AbinMat1.c                          */

#include <stdlib.h>
#include "error.h"
#include "abinmat1.h"

#define ELTO_NULO '\0'/* Un valor "ilegal" */

Abin CrearAbin (int maxNodos)
{
     Abin A;
     nodo i;

     A = (Abin) malloc(sizeof(tipoArbol));
     if (!A)
          ERROR("CrearAbin(): No hay memoria");
```

```
     A->nodos = (tElemento *) calloc(maxNodos,
                              sizeof(tElemento));
     if (!A->nodos)
          ERROR("CrearAbin(): No hay memoria");
     A->maxNodos = maxNodos;
     for (i = 0; i < maxNodos; i++)
          A->nodos[i] = ELTO_NULO;
     return A;
}

void CrearRaizB (tElemento e, Abin A)
{
     if (A->nodos[0] != ELTO_NULO)
          ERROR("CrearRaizB(): Arbol no vacío");

     A->nodos[0] = e;
}

void InsertarHijoIzqdoB (nodo n, tElemento e, Abin A)
{
     if (n < 0 || n > A->maxNodos - 1)
          ERROR(”...: Nodo inexistente");
     if (A->nodos[n] == ELTO_NULO)
          ERROR(”...: Nodo inexistente");
     if (A->maxNodos - 1 < 2 * n + 1)
          ERROR(”...: Espacio insuficiente "
               "para añadir hijo izquierdo");
     if (A->nodos[2*n+1] != ELTO_NULO)
          ERROR(”...: Ya existe hijo izquierdo");

     A->nodos[2*n+1] = e;
}
```

```
void InsertarHijoDrchoB (nodo n, tElemento e, Abin A)
{
     if (n < 0 || n > A->maxNodos - 1)
          ERROR(”...: Nodo inexistente");
     if (A->nodos[n] == ELTO_NULO)
          ERROR(”...: Nodo inexistente");
     if (A->maxNodos - 1 < 2 * n + 2)
          ERROR(”...: Espacio insuficiente "
                    "para añadir hijo derecho");
     if (A->nodos[2*n+2] != ELTO_NULO)
          ERROR(”...: Ya existe hijo derecho");

     A->nodos[2*n+2] = e;
}

void EliminarHijoIzqdoB (nodo n, Abin A)
{
     nodo hizqdo;
     if (n < 0 || n > A->maxNodos - 1)
          ERROR(”...: Nodo inexistente");
     if (A->nodos[n] == ELTO_NULO)
          ERROR(”...: Nodo inexistente");
     hizqdo = 2 * n + 1;
     if (hizqdo > A->maxNodos - 1)
          ERROR(”...: No existe hijo izquierdo");
     if (A->nodos[hizqdo] == ELTO_NULO)
          ERROR(”...: No existe hijo izquierdo");
     if (A->nodos[2*hizqdo+1] != ELTO_NULO ||
           A->nodos[2*hizqdo+2] != ELTO_NULO)
          ERROR(”...: Hijo izqdo no es hoja");
     A->nodos[hizqdo] = ELTO_NULO;
}
```

```
void EliminarHijoDrchoB (nodo n, Abin A)
{
     nodo hdrcho;
     if (n < 0 || n > A->maxNodos - 1)
       ERROR("EliminarHijoDrchoB(): Nodo inexistente");
     if (A->nodos[n] == ELTO_NULO)
       ERROR("EliminarHijoDrchoB(): Nodo inexistente");
     hdrcho = 2 * n + 2;
     if (hdrcho > A->maxNodos -1)
        ERROR("...: No existe hijo derecho");
     if (A->nodos[hdrcho] == ELTO_NULO)
        ERROR("...: No existe hijo derecho");
     if (A->nodos[2*hdrcho+1] != ELTO_NULO ||
          A->nodos[2*hdrcho+2] != ELTO_NULO)
        ERROR("...: El hijo derecho no es una hoja");
     A->nodos[hdrcho] = ELTO_NULO;
}

void EliminarRaizB (Abin A)
{
     if (A->nodos[0] == ELTO_NULO)
          ERROR("EliminarRaizB(): Árbol vacío");
     if ((A->maxNodos > 1 &&
        A->nodos[1] != ELTO_NULO) ||(A->maxNodos > 2 &&
           A->nodos[2] != ELTO_NULO))
           ERROR(”...: La raíz no es hoja");
     A->nodos[0] = ELTO_NULO;
}
void DestruirAbin (Abin A)
{
     free(A->nodos);
     free(A);
}
```

```
int ArbolVacioB (Abin A)
{
     return (A->nodos[0] == ELTO_NULO);
}

tElemento RecuperarB (nodo n, Abin A)
{
     if (n < 0 || n > A->maxNodos - 1)
          ERROR(”...: Nodo inexistente");
     if (A->nodos[n] == ELTO_NULO)
          ERROR(”...: Nodo inexistente");
     return A->nodos[n];
}

void ModificarB (nodo n, tElemento e, Abin A)
{
     if (n < 0 || n > A->maxNodos - 1)
          ERROR(”...: Nodo inexistente");
     if (A->nodos[n] == ELTO_NULO)
          ERROR(”...: Nodo inexistente");
     A->nodos[n] = e;
}

nodo RaizB (Abin A)
{
     if (A->nodos[0] != ELTO_NULO)
          return 0;
     else
          return NODO_NULO;
}
```

```
nodo PadreB (nodo n, Abin A)
{
     if (n < 0 || n > A->maxNodos - 1)
          ERROR("PadreB(): Nodo inexistente");
     if (A->nodos[n] == ELTO_NULO)
          ERROR("PadreB(): Nodo inexistente");
     if (n == 0)
          return NODO_NULO;
     else
          return (n - 1) / 2;
}

nodo HijoIzqdoB (nodo n, Abin A)
{
     nodo hizqdo;
     if (n < 0 || n > A->maxNodos - 1)
          ERROR("HijoIzqdoB(): Nodo inexistente");
     if (A->nodos[n] == ELTO_NULO)
          ERROR("HijoIzqdoB(): Nodo inexistente");
     hizqdo = 2 * n + 1;
     if (hizqdo > A->maxNodos - 1 ||
           A->nodos[hizqdo] == ELTO_NULO)
          return NODO_NULO;
     else
          return hizqdo;
}

nodo HijoDrchoB (nodo n, Abin A)
{
     nodo hdrcho;

     if (n < 0 || n > A->maxNodos - 1)
          ERROR("HijoDrchoB(): Nodo inexistente");
```

```
     if (A->nodos[n] == ELTO_NULO)
          ERROR("HijoDrchoB(): Nodo inexistente");
     hdrcho = 2 * n + 2;
     if (hdrcho > A->maxNodos - 1 ||
          A->nodos[hdrcho] == ELTO_NULO)
          return NODO_NULO;
     else
          return hdrcho;
}
```

3.3 Implementación de árboles binarios usando celdas enlazadas

Los nodos de un árbol podrían almacenarse utilizando celdas enlazadas en las que en cada nodo tendríamos un registro con dos campos, uno para el elemento del nodo y otro para almacenar un puntero al nodo que contiene a su padre. El nodo raíz puede distinguirse con el valor NODO_NULO como puntero a su propio padre (ya sabemos que el nodo raíz no tiene padre). Con esta representación, la operación para localizar el padre de un nodo dado es eficiente (O(1)) y fácil de implementar. Sin embargo, no es adecuada para las operaciones en las que haya que acceder a los hijos de un nodo n, ya que es necesario rastrear toda la estructura enlazada para encontrar los nodos cuyo padre sea n; pero, además, no es posible distinguir entre el hijo izquierdo y el derecho.

Otra opción podría ser que cada uno de los nodos fuese un registro con tres campos, uno para el elemento del nodo y otros dos en los que habría punteros que apuntasen a su hijo izquierdo e hijo derecho, respectivamente. Con esta representación la operación para localizar el hijo izquierdo o el hijo derecho de un nodo dado es eficiente (O(1)) y fácil de implementar. Sin embargo no es adecuada para las operaciones en las que sea necesario acceder al padre de un nodo n, ya que sería necesario recorrer toda la estructura enlazada para encontrar el padre del nodo en cuestión. La inexistencia del hijo izquierdo o derecho se resolvería utilizando el valor NODO_NULO.

La solución obvia a los problemas de las dos primeras representaciones propuestas consiste en combinarlas, es decir, en cada nodo habría un registro en el que, aparte del valor del elemento del nodo, se encuentran tres punteros, que apuntan a los nodos de su padre, hijo izquierdo e hijo derecho respectivamente. La inexistencia de algún hijo determinado o del padre se representará mediante el NODO_NULO. De esta forma, tanto el padre como los hijos de un nodo pueden encontrarse en un tiempo constante y es fácil moverse por el árbol hacia arriba y hacia abajo, aunque la estructura escogida es más costosa en espacio.

```
/* Abin.h                                   */

#define NODO_NULO NULL
typedef char tElemento; /* Por ejemplo */

#ifndef _ARBOL_BINARIO_
#define _ARBOL_BINARIO_
     typedef struct celda {
                 tElemento elto;
                 struct celda *padre, *hizq, *hder;
               } tipoNodo;
     typedef tipoNodo *nodo;
     typedef tipoNodo **Abin;

  Abin CrearAbin (void);
  void CrearRaizB (tElemento e, Abin A);
  void InsertarHijoIzqdoB(nodo n, tElemento e, Abin A);
  void InsertarHijoDrchoB(nodo n, tElemento e, Abin A);
  void EliminarHijoIzqdoB (nodo n, Abin A);
  void EliminarHijoDrchoB (nodo n, Abin A);
  void EliminarRaizB (Abin A);
  void DestruirAbin (Abin A);
  int ArbolVacioB (Abin A);
  tElemento RecuperarB (nodo n, Abin A);
  void ModificarB (nodo n, tElemento e, Abin A);
  nodo RaizB (Abin A);
  nodo PadreB (nodo n, Abin A);
  nodo HijoIzqdoB (nodo n, Abin A);
  nodo HijoDrchoB (nodo n, Abin A);
#endif
```

```
/* Abin.c                                   */

#include <stdlib.h>
#include "error.h"
#include "abin.h"

#define ARBOL_NULO NULL

static void DestruirNodos (nodo n)
{
     if (n != NODO_NULO) {
          DestruirNodos(n->hizq);
          DestruirNodos(n->hder);
          free(n)
     }
}
Abin CrearAbin ()
{
    Abin A;
     A= (Abin) malloc (sizeof (tipoNodo *));
     if (A== NULL)
          ERROR (“CrearAbin(): No hay memoria”);
     *A = NULL;
return A;
}

void CrearRaizB (tElemento e, Abin A)
{
     if (*A != NODO_NULO)
          ERROR("CrearRaizB(): Árbol no vacío");

     *A = (nodo) malloc(sizeof(tipoNodo));
     if (*A == NULL)
          ERROR("CrearRaizB(): No hay memoria");
```

```
     (*A)->elto = e;
     (*A)->padre = NODO_NULO;
     (*A)->hizq = NODO_NULO;
     (*A)->hder = NODO_NULO;
}

void InsertarHijoIzqdoB (nodo n, tElemento e, Abin A)
{
     if (n == NODO_NULO)
       ERROR("InsertarHijoIzqdoB(): Nodo inexistente");
     if (n->hizq != NODO_NULO)
       ERROR("InsertarHijoIzqdoB(): Ya existe
                hijo izquierdo");
     n->hizq = (nodo) malloc(sizeof(tipoNodo));
     if (n->hizq == NULL)
       ERROR("InsertarHijoIzqdoB(): No hay memoria");
     n->hizq->elto = e;
     n->hizq->padre = n;
     n->hizq->hizq = NODO_NULO;
     n->hizq->hder = NODO_NULO;
}

void InsertarHijoDrchoB (nodo n, tElemento e, Abin A)
{
     if (n == NODO_NULO)
       ERROR("InsertarHijoDrchoB(): Nodo inexistente");
     if (n->hder != NODO_NULO)
       ERROR("InsertarHijoDrchoB(): Ya existe
                  hijo derecho");
     n->hder = (nodo) malloc(sizeof(tipoNodo));
     if (n->hder == NULL)
       ERROR("InsertarHijoDrchoB(): No hay memoria");
```

```
     n->hder->elto = e;
     n->hder->padre = n;
     n->hder->hizq = NODO_NULO;
     n->hder->hder = NODO_NULO;
}

void EliminarHijoIzqdoB (nodo n, Abin A)
{
     if (n == NODO_NULO)
       ERROR("EliminarHijoIzqdoB(): Nodo inexistente");
     if (n->hizq == NODO_NULO)
       ERROR("EliminarHijoIzqdoB(): No existe
                 hijo izquierdo");
     if (n->hizq->hizq != NODO_NULO ||
           n->hizq->hder != NODO_NULO)
          ERROR("EliminarHijoIzqdoB(): El hijo
                izquierdo no es una hoja");

     free(n->hizq);
     n->hizq = NODO_NULO;
}

void EliminarHijoDrchoB (nodo n, Abin A)
{
     if (n == NODO_NULO)
          ERROR(”...: Nodo inexistente");
     if (n->hder == NODO_NULO)
          ERROR(”...: No existe hijo derecho");
     if (n->hder->hizq != NODO_NULO ||
           n->hder->hder != NODO_NULO)
           ERROR(”...: Hijo drcho no es una hoja");
     free(n->hder);
     n->hder = NODO_NULO;
}
```

```
void EliminarRaizB (Abin A)
{
     if (*A == NODO_NULO)
          ERROR("EliminarRaizB(): Árbol vacío");
     if ((*A)->hizq != NODO_NULO ||
           (*A)->hder != NODO_NULO)
        ERROR(”...: La raíz no es un nodo hoja");
     free(*A);
     *A = NODO_NULO;
}

void DestruirAbin (Abin A)
{
     DestruirNodos(RaizB(A));
     free(A);
}

int ArbolVacioB (Abin A)
{
     return (*A == ARBOL_NULO);
}

tElemento RecuperarB (nodo n, Abin A)
{
     if (n == NODO_NULO)
          ERROR("RecuperarB(): Nodo inexistente");
     return n->elto;
}

void ModificarB (nodo n, tElemento e, Abin A)
{
     if (n == NODO_NULO)
          ERROR(“ModificarB(): Nodo inexistente”);
```

```
     n->elto = e;
}

nodo RaizB (Abin A)
{
     return *A;
}

nodo PadreB (nodo n, Abin A)
{
     if (n == NODO_NULO)
          ERROR("PadreB(): Nodo inexistente");

     return n->padre;
}

nodo HijoIzqdoB (nodo n, Abin A)
{
     if (n == NODO_NULO)
          ERROR("HijoIzqdoB(): Nodo inexistente");

     return n->hizq;
}

nodo HijoDrchoB (nodo n, Abin A)
{
     if (n == NODO_NULO)
          ERROR("HijoDrchoB(): Nodo inexistente");
     return n->hder;
}
```

4. 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.

Especificación:

Arbol CrearArbol ()

Postcondiciones: Crea y devuelve un árbol vacío.

void CrearRaiz (tElemento e, Arbol A)

Precondiciones: *A* es el árbol vacío.

Postcondiciones: Crea el nodo raíz de *A* cuyo contenido es *e*.

void InsertarHijoIzqdo (nodo n, tElemento e, Arbol A)

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

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

void InsertarHermDrcho (nodo n, tElemento e, Arbol A)

Precondiciones: *n* es un nodo de *A y* no es el nodo raíz.

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

void EliminarHijoIzqdo (nodo n, Arbol A)

Precondiciones: *n* es un nodo de *A*. Existe *HijoIzqdo(n)* y es una hoja.

Postcondiciones: Destruye el hijo izquierdo del nodo *n* en el árbol *A*. El segundo hijo, si existe, se convierte en el nuevo hijo izquierdo.

void EliminarHermDrcho (nodo n, Arbol A)

Precondiciones: *n* es un nodo de *A*. Existe *HermDrcho(n)* y es una hoja.

Postcondiciones: Destruye el hermano derecho del nodo *n* en el árbol *A*. El siguiente hermano se convierte en el nuevo hermano derecho de *n*.

void EliminarRaiz (Arbol A)

Precondiciones: *A* no está vacío y *Raiz(A)* es una hoja.

Postcondiciones: Destruye el nodo raíz del árbol *A*.

void DestruirArbol (Arbol A)

Postcondiciones: Libera la memoria ocupada por el árbol *A*. Para usarlo otra vez se debe volver a crear con la operación *CrearArbol()*.

i*nt ArbolVacio (Arbol A)*

Postcondiciones: Devuelve 0 si A no es el árbol vacío y 1 en caso contrario.

tElemento Recuperar (nodo n, Arbol A)

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

Postcondiciones: Devuelve el elemento del nodo *n*.

void Modificar (nodo n, tElemento e, Arbol A)

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

Postcondiciones: Modifica el elemento del nodo *n* con *e*.

nodo Raíz (Arbol A)

Postcondiciones: Devuelve el nodo raíz de *A*. Si *A* está vacío, devuelve *NODO_NULO*.

nodo Padre (nodo n, Arbol A)

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

Postcondiciones: Devuelve el padre del nodo *n*. Si *n* es el nodo raíz, devuelve *NODO_NULO*.

nodo HijoIzqdo (nodo n, Arbol A)

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

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

nodo HermDrcho (nodo n, Arbol A)

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

Postcondiciones: Devuelve el nodo hermano derecho del nodo *n*. Si no existe, devuelve *NODO_NULO*.

4.1 Implementación de árboles generales mediante listas de hijos

Una estructura de datos apta para representar el TAD Árbol General consiste en un vector en el que se almacena, por un lado, el contenido de los nodos y, por otro, los índices de las celdas en las que se encuentran los hijos de cada nodo. Sin embargo, la mayor dificultad a la que nos enfrentamos radica en que el grado del árbol puede ser cualquiera y, por tanto, el número de hijos de cualquier nodo es indeterminado y además ilimitado. Esto hace que sea imposible crear un número fijo de campos para almacenar la posición de los hijos dentro del vector, como hemos hecho en el caso de los árboles binarios. La solución es crear para cada nodo un solo campo y guardar en él una lista con las posiciones de los hijos. Ya que el número de éstos es ilimitado, la mejor opción es utilizar una lista enlazada con cabecera.

Esta representación es bastante eficiente para recorrer un árbol de arriba abajo, es decir, de padres a hijos, pero plantea un grave problema de eficiencia en la operación padre, ya que esta operación implicaría ir recorriendo todas y cada una de las listas de hijos, hasta encontrar el padre deseado. Una solución obvia a este problema sería añadir un campo padre para cada una de las celdas del vector.

Por otra parte, en la inserción de nodos es necesario tener en cuenta el estado de las celdas del vector. Es decir, para añadir un nuevo nodo al árbol hay que localizar en el vector una celda libre. Esto se puede hacer marcando las celdas libres con el valor NODO_NULO en el campo padre, o bien, asegurándose de que en todo momento todas las celdas libres ocupan las últimas posiciones del vector,o sea, desde la última celda del vector ocupada por un nodo del árbol en adelante.

```
/* ArbLis.h --------------------------*/

#include "listnodo.h"
#define NODO_NULO -1
typedef char tElemento;  /* Por ejemplo */
#ifndef _ARBOL_
#define _ARBOL_
     typedef int nodo; /* índice de la matriz
                          entre 0 y maxNodos-1 */
     struct tNodo {
          tElemento elto;
          nodo padre;
          ListaNodos Hijos;
     };
```

```
     typedef struct {
                     struct tNodo *nodos;
                     int maxNodos;
                     int numNodos;
                  } tipoArbol;
     typedef tipoArbol *Arbol;

  Arbol CrearArbol (int maxNodos);
  void CrearRaiz (tElemento e, Arbol A);
  void InsertarHijoIzqdo(nodo n, tElemento e, Arbol A);
  void InsertarHermDrcho(nodo n, tElemento e, Arbol A);
  void EliminarHijoIzqdo (nodo n, Arbol A);
  void EliminarHermDrcho (nodo n, Arbol A);
  void EliminarRaiz (Arbol A);
  void DestruirArbol (Arbol A);
  int ArbolVacio (Arbol A);
  tElemento Recuperar (nodo n, Arbol A);
  void Modificar (nodo n, tElemento e, Arbol A);
  nodo Raiz (Arbol A);
  nodo Padre (nodo n, Arbol A);
  nodo HijoIzqdo (nodo n, Arbol A);
  nodo HermDrcho (nodo n, Arbol A);
#endif

/*-----------------------------------------*/
/* listnodo.h                              */
/*    Definición del TAD lista de nodos de   */
/*    un árbol. Lista enlazada con cabecera. */
/*-----------------------------------------*/
typedef int nodo; /* tipo de elementos de una lista */

#ifndef _LISTA_NODOS_
#define _LISTA_NODOS_
```

```
   typedef struct celda {
              nodo elemento;
              struct celda *sig;
            } tipoCelda;
   typedef tipoCelda *ListaNodos;
   typedef tipoCelda *posicion;

     ListaNodos CrearLista (void);
     void Insertar (nodo x, posicion p, ListaNodos L);
     void Eliminar (posicion p, ListaNodos L);
     nodo ListaRecuperar(posicion p, ListaNodos L);
     posicion Buscar (nodo x, ListaNodos L);
     posicion Siguiente(posicion p,ListaNodos L);
     posicion Anterior(posicion p,ListaNodos L);
     posicion Primera (ListaNodos L);
     posicion Fin (ListaNodos L);
     void DestruirLista (ListaNodos L);
#endif
```

Implementación de algunas operaciones del TAD Árbol

```
Arbol CrearArbol (int maxNodos)
{
     Arbol A;
     nodo i;
     A = (Arbol) malloc(sizeof(tipoArbol));
     if (!A)
          ERROR("CrearArbol(): No hay memoria");

     A->nodos = (struct tNodo *)
           calloc(maxNodos, sizeof(struct tNodo));
     if (!A->nodos)
          ERROR("CrearArbol(): No hay memoria");
```

```
     A->maxNodos = maxNodos;
     A->numNodos = 0;
     for (i = 0; i < maxNodos; i++)
          A->nodos[i].padre = NODO_NULO;
     return A;
}

void CrearRaiz (tElemento e, Arbol A)
{
     if ((A)->numNodos)
          ERROR("CrearRaiz(): Árbol no vacío");

     (A)->numNodos = 1;
     (A)->nodos[0].elto = e;
     (A)->nodos[0].Hijos = CrearLista();
}

void InsertarHijoIzqdo (nodo n, tElemento e, Arbol A)
{
     nodo hizqdo;

if (n < 0 || n > A->maxNodos - 1)
          ERROR(”...: Nodo inexistente");
     if (n != 0 && A->nodos[n].padre == NODO_NULO)
          ERROR(”...: Nodo inexistente");
     if (A->numNodos == A->maxNodos)
          ERROR(”...: Espacio insuficiente "
               "para añadir hijo izquierdo");
     /*Añadir el nuevo nodo en la primera posición
libre */
     for (hizqdo = 1;
            A->nodos[hizqdo].padre != NODO_NULO;
            hizqdo++);
```

```
     A->numNodos++;
     A->nodos[hizqdo].elto = e;
     A->nodos[hizqdo].padre = n;
     A->nodos[hizqdo].Hijos = CrearLista();
     /* Añadir el nuevo nodo a la lista de hijos
          del padre */
     Insertar(hizqdo,
               Primera(A->nodos[n].Hijos),
               A->nodos[n].Hijos);
}

void EliminarHermDrcho (nodo n, Arbol A)
{
     nodo hdrcho, pad;
     posicion p;

     if (n < 0 || n > A->maxNodos - 1)
          ERROR(”...: Nodo inexistente");

     if (n == 0)
          ERROR(”...: No existe hermano derecho");

     if (A->nodos[n].padre == NODO_NULO)
          ERROR(”...(): Nodo inexistente");

     pad = A->nodos[n].padre;
     p = Buscar(n, A->nodos[pad].Hijos);
     p = Siguiente(p, A->nodos[pad].Hijos);
     if (p == Fin(A->nodos[pad].Hijos))
          ERROR(”...: No existe hermano derecho");

     hdrcho = ListaRecuperar(p, A->nodos[pad].Hijos);
```

```
     if (Primera(A->nodos[hdrcho].Hijos) !=
           Fin(A->nodos[hdrcho].Hijos))
          ERROR(”...: Hijo derecho no es hoja");

     A->numNodos--;
     A->nodos[hdrcho].padre = NODO_NULO;
     DestruirLista(A->nodos[hdrcho].Hijos);
     Eliminar(p, A->nodos[pad].Hijos);
}

nodo HijoIzqdo (nodo n, Arbol A)
{
     if (n < 0 || n > A->maxNodos - 1)
          ERROR("HijoIzqdo(): Nodo inexistente");

     if (n != 0 &&
           A->nodos[n].padre == NODO_NULO)
          ERROR("HijoIzqdo(): Nodo inexistente");

     if (Primera(A->nodos[n].Hijos) ==
           Fin(A->nodos[n].Hijos))
          return NODO_NULO;
     else
          return ListaRecuperar(
                    Primera(A->nodos[n].Hijos),
                    A->nodos[n].Hijos);
}
```

Inconvenientes:

- Número limitado de nodos.
- Acceso al hermano derecho de un nodo poco eficiente. El tiempo es proporcional al número de hermanos de un nodo.

4.2 Implementación de árboles generales usando celdas enlazadas

En esta representación el árbol se implementa mediante un puntero a su nodo raíz. En todos y cada uno de los nodos se encuentra, por un lado, el contenido del mismo, y por otro un puntero a su nodo padre, y otro a una lista de hijos. No es posible acceder a cada uno de los hijos con un puntero independiente, ya que a priori no está limitado ni es conocido el número de hijos de un nodo.

```
/* Agen.h                                   */
#define NODO_NULO NULL
typedef char tElemento; /* Por ejemplo */
#ifndef _ARBOL_
#define _ARBOL_
     typedef struct celda {
                 tElemento elto;
                 struct celda *padre, *hizq, *heder;
               } tipoNodo;
     typedef tipoNodo *nodo;
     typedef tipoNodo **Arbol;
  Arbol CrearArbol (void);
  void CrearRaiz (tElemento e, Arbol A);
  void InsertarHijoIzqdo(nodo n, tElemento e, Arbol A);
  void InsertarHermDrcho(nodo n, tElemento e, Arbol A);
  void EliminarHijoIzqdo (nodo n, Arbol A);
  void EliminarHermDrcho (nodo n, Arbol A);
  void EliminarRaiz (Arbol A);
  void DestruirArbol (Arbol A);
  int ArbolVacio (Arbol A);
  tElemento Recuperar (nodo n, Arbol A);
  void Modificar (nodo n, tElemento e, Arbol A);
  nodo Raiz (Arbol A);
  nodo Padre (nodo n, Arbol A);
  nodo HijoIzqdo (nodo n, Arbol A);
  nodo HermDrcho (nodo n, Arbol A);
#endif
```

```
/* Agen.c                                   */

#include <stdlib.h>
#include "error.h"
#include "agen.h"
#define ARBOL_NULO NULL

static void DestruirNodos (nodo n, Arbol A)
{
     nodo hermder;
     if (n->hizq != NODO_NULO) {
      /* Destruir hermanos del hijo izqdo */
     hermder = n->hizq->heder;
     while (hermder != NODO_NULO) {
        n->hizq->heder = hermder->heder;
        DestruirNodos(hermder, A);
        hermder = n->hizq->heder;
     }
     /* Destruir el hijo izqdo */
     DestruirNodos(n->hizq, A);
     }
     free(n);
}

Arbol CrearArbol ()
{
     Arbol A;
     A= (Arbol) malloc (sizeof(tipoNodo *));
     if (A==NULL)
          ERROR (“CrearArbol(): No hay memoria”);
     *A=NULL;
     return A;
}
```

```
void CrearRaiz (tElemento e, Arbol A)
{
     if (*A != NODO_NULO)
          ERROR("CrearRaiz(): Árbol no vacío");

     *A = (nodo) malloc(sizeof(tipoNodo));
     if (*A == NULL)
          ERROR("CrearRaiz(): No hay memoria");

     (*A)->elto = e;
     (*A)->padre = NODO_NULO;
     (*A)->hizq = NODO_NULO;
     (*A)->heder = NODO_NULO;
}

void InsertarHijoIzqdo (nodo n, tElemento e, Arbol A)
{
     nodo hizqant;
     if (n == NODO_NULO)
          ERROR(”...: Nodo inexistente");
     hizqant = n->hizq;
     n->hizq = (nodo) malloc(sizeof(tipoNodo));
     if (n->hizq == NULL)
          ERROR(”...: No hay memoria");
     n->hizq->elto = e;
     n->hizq->padre = n;
     n->hizq->hizq = NODO_NULO;
     /* hijo izqdo es el nuevo hermano drcho */
     n->hizq->heder = hizqant;
}
```

```
void InsertarHermDrcho (nodo n, tElemento e, Arbol A)
{
     nodo hedchoant;
     if (n == NODO_NULO)
          ERROR(”...: Nodo inexistente");
     if (*A == n)
          ERROR(”...: El nodo raíz no puede tener
hermano");
     hedchoant = n->heder;
     n->heder = (nodo) malloc(sizeof(tipoNodo));
     if (n->heder == NULL)
          ERROR(”...: No hay memoria");
     n->heder->elto = e;
     n->heder->padre = n->padre;
     n->heder->hizq = NODO_NULO;
     /* Inserta el nuevo hermano en la lista */
     n->heder->heder = hedchoant;
}

void EliminarHijoIzqdo (nodo n, Arbol A)
{
     nodo hizqdo;

     if (n == NODO_NULO)
          ERROR(”...: Nodo inexistente");
     if (n->hizq == NODO_NULO)
          ERROR(”...: No existe hijo izquierdo");
     if (n->hizq->hizq != NODO_NULO)
          ERROR(”...: Hijo izquierdo no es hoja");
     hizqdo = n->hizq;
     n->hizq = hizqdo->heder;
     free(hizqdo);
}
```

```
void EliminarHermDrcho (nodo n, Arbol A)
{
     nodo hermder;

     if (n == NODO_NULO)
          ERROR(”...: Nodo inexistente");
     if (n->heder == NODO_NULO)
          ERROR(”...: No existe hermano derecho");
     if (n->heder->hizq != NODO_NULO)
           ERROR(”...: Hermano drcho no es hoja");

     hermder = n->heder;
     n->heder = hermder->heder;
     free(hermder);
}

void EliminarRaiz (Arbol A)
{
     if (*A == NODO_NULO)
          ERROR("EliminarRaiz(): Árbol vacío");
     if ((*A)->hizq != NODO_NULO)
          ERROR(”...: La raíz no es una hoja");

     free(*A);
     *A = NODO_NULO;
}
void DestruirArbol (Arbol A)
{
     if (*A != ARBOL_NULO)
          DestruirNodos(RaizB(A), A);
     free(A);
}
```

5. RECORRIDOS DE ÁRBOLES BINARIOS

Existen diversos problemas cuya solución requiere procesar todos los nodos de un árbol. Para llevar a cabo este proceso hay que recorrer o visitar todos los nodos de una forma sistemática, es decir, siguiendo algún orden que impida olvidar algún nodo del árbol o visitar algún nodo en más de una ocasión. Para realizar este recorrido de forma ordenada, existen varios algoritmos que se pueden clasificar en dos categorías: recorridos en profundidad y recorrido en anchura o por niveles. Los primeros se clasifican a su vez en: recorrido en preorden, en inorden y en postorden. El nombre de cada algoritmo hace referencia al orden en que se visitan los nodos del árbol. Las definiciones recursivas de estos recorridos son las siguientes:

- **Preorden**: Primero se procesa la raíz y a continuación el subárbol izquierdo, en preorden, seguido del subárbol derecho también en preorden.

- **Inorden**: Se procesa en inorden el subárbol izquierdo, a continuación se procesa la raíz del árbol y por último, el subárbol derecho en inorden.

- **Postorden**: Se procesan los subárboles izquierdo y derecho en postorden, seguidos de la raíz del árbol.

- **Anchura**: Se procesan, de izquierda a derecha, todos los nodos de un nivel antes que los del siguiente, empezando por el primer nivel, o sea, por el nodo raíz.

Las funciones que se proponen a continuación incorporan como parámetro un puntero a una función para procesar cada nodo del árbol. De esta forma, es posible utilizar diferentes funciones para procesar los nodos de distinta manera, dependiendo de los requisitos del problema que se esté resolviendo. Así por ejemplo, podemos definir una función Imprimir( ) que escriba el contenido de un nodo en la pantalla y llamar a Preorden(Raíz(A), A, Imprimir) para escribir en preorden todos los nodos del árbol A; o bien, podemos definir una función Incrementar( ) que le sume 1 al elemento de un nodo y llamar a Preorden(Raíz(A), A, Incrementar) para incrementar los valores de todos los nodos de un árbol binario de enteros.

```
void PreordenAbin (nodo n, Abin A,
                   void (*Procesar)(nodo, Abin))

/* Recorrido en preorden del subárbol cuya
     raíz es el nodo n perteneciente al árbol
     binario A. Cada nodo visitado se procesa
     mediante la función Procesar() */
```

```
{
     if (n != NODO_NULO) {
          Procesar(n, A);
          PreordenAbin(HijoIzqdoB(n, A), A, Procesar);
          PreordenAbin(HijoDrchoB(n, A), A, Procesar);
     }
}

void InordenAbin (nodo n, Abin A,
                  void (*Procesar)(nodo, Abin))
{
     if (n != NODO_NULO) {
          InordenAbin(HijoIzqdoB(n, A), A, Procesar);
          Procesar(n, A);
          InordenAbin(HijoDrchoB(n, A), A, Procesar);
     }
}

void PostordenAbin (nodo n, Abin A,
                  void (*Procesar)(nodo, Abin))
{
     if (n != NODO_NULO) {
          PostordenAbin(HijoIzqdoB(n, A), A, Procesar);
          PostordenAbin(HijoDrchoB(n, A), A, Procesar);
          Procesar(n, A);
     }
}
```

6. RECORRIDOS DE ÁRBOLES GENERALES

Las definiciones de los recorridos en profundidad de árboles de cualquier grado son generalizaciones de las dadas para árboles binarios.

- **Preorden**: Se procesa la raíz y a continuación, de izquierda a derecha, cada uno de los subárboles en preorden.

- **Inorden**: Se procesa en inorden el subárbol izquierdo, a continuación el nodo raíz y por último, de izquierda a derecha, el resto de los subárboles en inorden.

- **Postorden**: Se procesan de izquierda a derecha todos los subárboles de la raíz en postorden y a continuación el nodo raíz.

La definición del recorrido en anchura o por niveles es la misma que para árboles binarios.

```
void Preorden (nodo n, Arbol A,
                  void (*Procesar)(nodo, Arbol))
{
     if (n != NODO_NULO) {
          Procesar(n, A);
          n = HijoIzqdo(n, A);
          while (n != NODO_NULO) {
               Preorden(n, A, Procesar);
               n = HermDrcho(n, A);
          }
     }
}

void Inorden (nodo n, Arbol A,
                  void (*Procesar)(nodo, Arbol))
{
     nodo hijo;
     if (n != NODO_NULO) {
          hijo = HijoIzqdo(n, A);
          if (hijo != NODO_NULO) {
             Inorden(hijo , A, Procesar);
             Procesar(n, A);
             while ((hijo = HermDrcho(hijo, A))
                         != NODO_NULO)
               Inorden(hijo, A, Procesar);
          }
```

```
          else
             Procesar(n, A);
     }
}

void Postorden (nodo n, Arbol A,
                void (*Procesar)(nodo, Arbol))
{
     nodo hijo;
     if (n != NODO_NULO) {
          hijo = HijoIzqdo(n, A);
          while (hijo != NODO_NULO) {
               Postorden(hijo, A, Procesar);
               hijo = HermDrcho(hijo, A);
          }
          Procesar(n, A);
     }
}

void RecNiveles (nodo n, Arbol A,
                 void (*Procesar)(nodo, Arbol))
/* Recorrido por niveles del subárbol cuya
raíz es el nodo n perteneciente al árbol A.
Cada nodo recorrido se procesa mediante la
función Procesar() */
{
     Cola C; /* Cola de nodos */
     nodo hizqdo, hdrcho;

     C = CrearCola();
     if (n != NODO_NULO) ColaPush(n, C);
     while (!ColaVacia(C)) {
          n = Frente(C);
```

```
          ColaPop(C);
          Procesar(n, A);
          hizqdo = HijoIzqdo(n, A);
          if (hizqdo != NODO_NULO) {
             ColaPush(hizqdo, C);
             hdrcho = HermDrcho(hizqdo, A);
             while (hdrcho != NODO_NULO) {
               ColaPush(hdrcho, C);
               hdrcho = HermDrcho(hdrcho, A);
             }
          }
     }
     DestruirCola(C);
}
```

Estas funciones se pueden escribir de forma no recursiva utilizando una pila. Por ejemplo, en la versión iterativa del recorrido en preorden se introduce en la pila cada nodo procesado, el cuál se sacará más tarde cuando se hayan procesado todos sus hijos. Es decir, la pila se utiliza para guardar los nodos de los cuales quedan subárboles por procesar.

```
void Preorden2 (nodo n, Arbol A,
                void (*Procesar)(nodo, Arbol))
{
     Pila P; /* Pila de nodos */

     P = CrearPila();
     do {
          if (n != NODO_NULO) {
             Procesar(n, A);
             Push(n, P);
             n = HijoIzqdo(n, A);
          }
          else if (!Vacia(P)) {
             n = HermDrcho(Tope(P), A);
             Pop(P);
          }
```

```
    } while (!Vacia(P));
    DestruirPila(P);
}
```

7. TAD ÁRBOL BINARIO DE BÚSQUEDA

Un **Árbol Binario de Búsqueda** es un modo de organizar conjuntos cuyos elementos están clasificados según un orden lineal. Esta organización es útil cuando se tiene un conjunto de elementos tan grande que no es práctico emplear un TAD lineal para realizar búsquedas de elementos en un tiempo razonable. Con un ABB es posible utilizar un algoritmo simple y eficiente basado en la propiedad que caracteriza a este tipo de árboles, mediante el cual se puede reducir drásticamente el tiempo de búsqueda de un elemento. La propiedad de estos árboles es que todos los elementos almacenados en el subárbol izquierdo de cualquier nodo *n*, son menores que el elemento almacenado en *n*, y todos los elementos almacenados en el subárbol derecho son mayores.

La propiedad de ABB hace que sea muy fácil diseñar un algoritmo para realizar la búsqueda. Para determinar si un elemento *e* está presente en el árbol, lo comparamos con el elemento situado en la raíz, *r*. Si coinciden, la búsqueda finaliza con éxito; si $e<r$, entonces es evidente que si está *e* en el árbol ha de estar en el subárbol izquierdo de la raíz; y si $e>r$ estará en el subárbol derecho. El algoritmo continuará de la misma forma en el subárbol correspondiente hasta encontrar *e* o hasta alcanzar un nodo hoja sin encontrarlo.

Puede probarse que siguiendo este algoritmo, la búsqueda en un ABB de *n* elementos requiere $O(log_2 n)$ operaciones en el caso medio (árbol completo) y en el peor caso (árbol degenerado en una lista) será necesario revisar los n elementos. Es por tanto interesante que al construir el árbol nos acerquemos, en lo posible, al árbol completo para obtener los mejores tiempos en la búsqueda. Esta característica se desarrollará posteriormente en el estudio de árboles AVL.

Definición:

Un **árbol binario de búsqueda** es un árbol binario en el que los nodos almacenan elementos de un conjunto (no existen elementos repetidos). La propiedad que define a estos árboles es que todos los elementos almacenados en el subárbol izquierdo de cualquier nodo *n* son menores que el elemento de *n*, y todos los elementos almacenados en el subárbol derecho de *n* son mayores que el elemento almacenado en el mismo.

Consideraremos que el tipo de los elementos consiste en un registro con un campo clave que distingue unos de otros y además existe un orden lineal definido sobre los valores de la clave.

Operaciones:

ABB CrearABB ()

Postcondiciones: Crea y devuelve un árbol vacío.

nodoABB ABBBuscar (tClave c, ABB A)

Precondiciones: *A* es un árbol binario de búsqueda.

Postcondiciones: Devuelve el nodo que contiene el elemento cuya clave es *c*. Si no existe devuelve *NODO_NULO*.

tElemento ABBRecuperar (nodoABB n)

Precondiciones: *n* es un nodo de un árbol binario de búsqueda.

Postcondiciones: Devuelve el elemento del nodo *n*.

*void ABBInsertar (tElemento e, ABB *A)*

Precondiciones: *e* no pertenece al árbol **A*.

Postcondiciones: Inserta el elemento *e* en el árbol **A.*

*void ABBSuprimir (tElemento e, ABB *A)*

Postcondiciones: Destruye el nodo que almacena el elemento *e* en el árbol **A*. Si no existe *e*, no se hace nada.

void DestruirABB (ABB A)

Postcondiciones: Libera la memoria ocupada por el árbol *A*. Para usarlo otra vez se debe volver a crear con la operación *CrearABB().*

7.1 Implementación de árboles binarios de búsqueda mediante celdas enlazadas

La representación para este tipo de árboles, caso particular de árboles binarios, es la misma que para los árboles binarios mediante celdas enlazadas con las siguientes excepciones:

- las operaciones propias de los ABB no requieren que los nodos tengan un puntero al padre (en las búsquedas el árbol se recorre en sentido descendente),
- para facilitar la implementación de las operaciones y su utilización en las aplicaciones, árbol y nodo son tipos equivalentes, por tanto, se define el tipo árbol como un puntero a la raíz.

```
/* ABinBus.h                              */

#define NODO_NULO NULL

#ifndef _tElemento_
#define _tElemento_
     typedef int tClave; /* un tipo con una
          relación de orden entre sus valores */
     typedef struct {
                     tClave clave;
                     /* resto de campos */
               } tElemento;
#endif

#ifndef _ABB_
#define _ABB_
     typedef struct celdaABB {
                 tElemento elto;
                 struct celdaABB *hizq,
                                 *hder;
               } tCeldaABB;
     typedef tCeldaABB *nodoABB;
     typedef tCeldaABB *ABB;

     ABB CrearABB (void);
     nodoABB ABBBuscar (tClave c, ABB A);
     tElemento ABBRecuperar (nodoABB n);
     void ABBInsertar (tElemento e, ABB *A);
     void ABBSuprimir (tElemento e, ABB *A);
     void DestruirABB (ABB A);
#endif
```

```
/* ABinBus.c                                */

#include <stdlib.h>
#include "error.h"
#include "ABinBus.h"

#define ABB_NULO NULL

ABB CrearABB (void)
{
     return ABB_NULO;
}

nodoABB ABBBuscar (tClave c, ABB A)
{
     if (A == ABB_NULO)
          return NODO_NULO;
     else if (A->elto.clave == c)
          return A;
     else if (A->elto.clave > c)
          return ABBBuscar(c, A->hizq);
     else
          return ABBBuscar(c, A->hder);
}

tElemento ABBRecuperar (nodoABB n)
{
     if (n == NODO_NULO)
       ERROR("ABBRecuperar(): No existe nodo");

     return n->elto;
}
```

```
void ABBInsertar (tElemento e, ABB *A)
{
  if (*A == ABB_NULO) {
      *A = (nodoABB) malloc(sizeof(tCeldaABB));
       if (*A == NULL)
           ERROR("ABBInsertar(): Sin memoria");
          (*A)->elto = e;
          (*A)->hizq = NODO_NULO;
          (*A)->hder = NODO_NULO;
     }
     else if (e.clave < (*A)->elto.clave)
          ABBInsertar(e, &(*A)->hizq);
     else if (e.clave > (*A)->elto.clave)
          ABBInsertar(e, &(*A)->hder);
     else
          ERROR("...: Ya existe el elemento");
}
static tElemento BorrarMin (ABB *A)
/* Devuelve y elimina el menor elemento del
 árbol binario de búsqueda apuntado por A. */
{
     tElemento e;
     nodoABB n;
     if ((*A)->hizq == NODO_NULO) {
          e = (*A)->elto;
          n = (*A)->hder;
          free (*A);
          *A = n;
          return e;
     }
     else
          return BorrarMin(&(*A)->hizq);
}
```

```
void ABBSuprimir (tElemento e, ABB *A)
{
     nodoABB n;

     if (*A != ABB_NULO)
          if (e.clave < (*A)->elto.clave)
             ABBSuprimir(e, &(*A)->hizq);
          else if (e.clave > (*A)->elto.clave)
             ABBSuprimir(e, &(*A)->hder);
          else /* Suprimir la raíz */
             if ((*A)->hizq == NODO_NULO &&
                (*A)->hder == NODO_NULO) {
                free(*A);
                *A = ABB_NULO;
             }
             else if ((*A)->hizq == NODO_NULO) {
                n = (*A)->hder;
                free(*A);
                *A = n;
             }
             else if ((*A)->hder == NODO_NULO) {
                n = (*A)->hizq;
                free(*A);
                *A = n;
             }
             else
               (*A)->elto = BorrarMin(&(*A)->hder);
}

void DestruirABB (ABB A)
{
     if (A != ABB_NULO) {
          DestruirABB(A->hizq);
          DestruirABB(A->hder);
```

```
            free(A);
        }
}
```

8. ÁRBOLES ABB EQUILIBRADOS (AVL)

Es fácil comprobar que el orden en el que se inserten los elementos en un ABB determina el grado de equilibrio del árbol, que en el peor caso puede llevarnos a un árbol degenerado en una lista y por lo tanto la búsqueda de un elemento necesitaría un tiempo O(n). Para obtener tiempo $O(\log_2 n)$ es necesario mantener en todo momento el árbol tan equilibrado como sea posible.

Un **árbol binario equilibrado** es un árbol binario en el cuál las alturas de los dos subárboles, para cada nodo, nunca difieren en más de una unidad.

A los árboles binarios de búsqueda equilibrados también se les llama AVL en honor a Adelson-Velskii y Landis que fueron los primeros en proponer y desarrollar este concepto.

El factor de equilibrio o balance de un nodo se define como la altura del subárbol derecho menos la altura del subárbol izquierdo correspondiente.

El factor de equilibrio de cada nodo en un árbol equilibrado será 1, -1 ó 0.

Para conseguir un árbol AVL con un número dado, N, de nodos, hay que distribuir equitativamente los nodos a la izquierda y a la derecha de un nodo dado, y recordar en todo momento que son árboles binarios de búsqueda y que se debe conservar para cada nodo la propiedad de los ABB.

En los AVL las operaciones para localizar una determinada clave, insertar un nodo y eliminar un nodo se pueden efectuar en $O(\log_2 n)$.

9. TAD ÁRBOL PARCIALMENTE ORDENADO

Un **árbol parcialmente ordenado** (montículo) es un árbol completamente lleno, con la posible excepción del nivel más bajo, el cuál se llena de izquierda a derecha (árbol completo).

Un árbol binario completo de altura h tiene entre 2^h y 2^{h+1} -1 nodos. Esto implica que la altura de un árbol binario completo es $\log_2 n$, por lo que las inserciones y eliminaciones son $O(\log_2 n)$.

Debido a esta particularidad de su estructura la mejor representación podría ser la representación de árboles binarios mediante un vector de posiciones relativas, de esta forma las operaciones necesarias para recorrer el árbol son muy sencillas y muy rápidas.

La propiedad que permite efectuar rápidamente las operaciones es la propiedad de orden del APO que consiste en que cualquier nodo debe ser menor que todos sus descendientes, o sea, en un APO, para todo nodo X, la clave en el padre de X es menor o igual que la clave en X (con excepción de la raíz, que no tiene padre).

Las operaciones básicas sobre APO son la inserción y la eliminación de elementos, teniéndonos que asegurar que siempre se mantenga la propiedad de orden para este tipo de árbol.

Una de las aplicaciones más usuales de los APO es la representación de colas con prioridad, donde la relación de orden bien definida por la prioridad de los elementos, de forma que cada nodo es más prioritario que sus hijos, y por tanto, en la raíz siempre se encuentra el elemento con mayor prioridad.

```
/* APO.h                                    */

typedef char tElemento;  /* Por ejemplo */
#ifndef _APO_
#define _APO_
  typedef int nodo; /* índice de la matriz
                       entre 0 y maxNodos-1 */
  typedef struct tArbol {
                     tElemento *nodos;
                     int maxNodos;
                     int ultimo;
             } tipoArbol;
  typedef tipoArbol *APO;

  APO CrearAPO (int maxNodos);
  void InsertarAPO (tElemento e, APO A);
  void SuprimirMinAPO (APO A);
  tElemento RecuperarMinAPO (APO A);
  void DestruirAPO (APO A);
  int APOVacio (APO A);
#endif
```

```
/* APO.c                                        */

#include <stdlib.h>
#include "error.h"
#include "APO.h"

APO CrearAPO (int maxNodos)
{
     APO A;
     A = (APO) malloc(sizeof(tipoArbol));
     if (A == NULL)
          ERROR("CrearAPO(): No hay memoria");
     A->nodos =(tElemento *) calloc(maxNodos,
                               sizeof(tElemento));
     if (A->nodos == NULL)
          ERROR("CrearAPO(): No hay memoria");
     A->maxNodos = maxNodos;
     A->ultimo = -1;
     return A;
}
void InsertarAPO (tElemento e, APO A)
{
   int p;
   if (A->maxNodos - 1 == A->ultimo)
      ERROR("InsertarAPO(): APO lleno");
   A->ultimo++;
   p = A->ultimo;
   while ((p > 0) && (A->nodos[(p-1)/2]>e)) {
      A->nodos[p] = A->nodos[(p-1)/2];
      p = (p-1)/2;
   }
   A->nodos[p] = e;
}
```

```
void SuprimirMinAPO (APO A)
{
   int p;
   int pMin, fin;
   tElemento ultimoElto;

   if (A->ultimo == -1)
      ERROR("SuprimirAPO(): APO vac¡o");

   ultimoElto = A->nodos[A->ultimo];
   A->ultimo--;
   if (A->ultimo >= 0) {
      p = 0;
      if (A->ultimo > 0) {
         fin = 0;
         while (p <= (A->ultimo-1)/2 && !fin) {
            if (2*p+1 == A->ultimo)
               pMin = 2*p+1;
            else if (A->nodos[2*p+1] < A->nodos[2*p+2])
               pMin = 2*p+1;
            else
               pMin = 2*p+2;
            if (A->nodos[pMin] < ultimoElto) {
               A->nodos[p] = A->nodos[pMin];
               p = pMin;
            }
            else
               fin = 1;
         }
      }
      A->nodos[p] = ultimoElto;
   }
}
```

```
tElemento RecuperarMinAPO (APO A)
{
   return A->nodos[0];
}

void DestruirAPO (APO A)
{
     free(A->nodos);
     free(A);
}

int APOVacio (APO A)
{
     return (A->ultimo == -1);
}
```

10. ÁRBOLES B

Es relativamente frecuente en la vida real, encontrarse colecciones de elementos formados por muchos millones, e incluso miles de millones de registros. Pensemos, por ejemplo, en los datos que deben procesar compañías como el BSCH, Telefónica, o incluso, la mismísima Hacienda. No hablamos simplemente de clientes, cada elemento podría ser, por ejemplo, una llamada telefónica, imaginemos cuántas llamadas se hacen en un año en España, y pensemos en USA o Japón, por ejemplo.

Si la organización elegida para almacenar este conjunto de datos es un ABB, el tiempo de búsqueda es $O(\log_2 N)$, pero en cada uno de los pasos hay que acceder a un dato almacenado en memoria externa y el tiempo de acceso a esta memoria es del orden de millones de veces más lento que en memoria principal. Esto da lugar a tiempos de búsqueda inaceptables.

Para reducir este número de accesos se utiliza una generalización de ABB que nos permite almacenar en cada nodo varios elementos. Para minimizar el tiempo en las búsquedas, lo ideal es que cada nodo ocupe un bloque de memoria secundaria. De esta forma, en cada acceso leemos la mayor cantidad de información útil para determinar por qué camino continuar la búsqueda, lo cuál se hace se hace finalmente en memoria principal utilizando para ello algún algoritmo de búsqueda rápida entre los elementos de un nodo.

La adaptación correcta de la idea de árboles múltiples (generalización de ABB para árboles generales), consiste en pensar en los nodos como bloques físicos. Un nodo interior contiene punteros a m hijos y también contiene m-1 claves que separan los descendientes del hijo. Los nodos hoja son bloques también, y contienen los registros del archivo principal.

Si se emplea un ABB de n nodos para representar un archivo almacenado en forma externa, pueden requerirse en promedio $\log_2 N$ accesos a bloques para recuperar un registro del archivo. En cambio, si se utiliza un árbol de búsqueda general (m-ario) para representar el archivo, requerirá, en promedio sólo $\log_m N$ accesos a bloques para recuperar un registro.

No se puede hacer *m* arbitrariamente grande, pues si m es demasiado grande, el tamaño de un nodo podrá exceder el tamaño de un bloque y será necesario más de un acceso a memoria secundaria para leer un nodo. Además es más lento leer y procesar un bloque más grande, así que hay que buscar el valor óptimo de m.

Este tipo de árboles recibe el nombre de **árboles B**, y tienen las siguientes características:

1. Un nodo que no es hoja con *K* claves contiene *K+1* hijos.

2. Cada nodo tiene un número máximo de *m* hijos, donde *m* es el orden del árbol.

3. Cada nodo, excepto la raíz, contiene por lo menos $|m/2|-1$ claves y, por supuesto, no más de $m-1$ claves.

4. La raíz tiene como mínimo una clave y dos hijos (a menos que sea una hoja).

5. Todas las hojas aparecen en el mismo nivel.

10.1 Inserción en un árbol B.

Básicamente la inserción en un árbol B no debe alterar ninguna de las características que definen el mismo. Dado que se trata de la inserción de una clave, la característica que nos preocupa es el máximo de claves que puede tener un nodo (aparece en la característica número 3). Puede darse el caso de que en la hoja en la que debería realizarse la inserción ya hubiese *m-1* claves, por lo que resulta imposible insertar una más sin superar el máximo. Este problema se soluciona mediante el procedimiento de división y promoción, como veremos en el siguiente ejemplo.

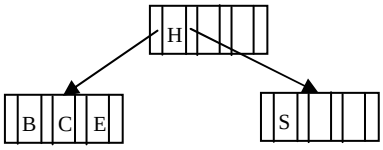
En este ejemplo vamos a utilizar un árbol B de orden 4, lo cuál implica que en cada nodo vamos a tener como máximo 3 claves y 4 hijos.

Vamos a insertar la siguiente secuencia de claves en el árbol:

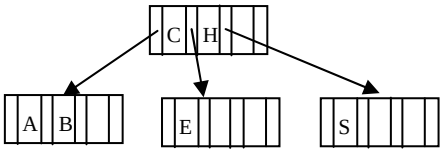
C, E, H, S, B, A, P, D, J, L, K, I, G, M, T, N, Y

En primer lugar insertamos C, E y H en el nodo inicial

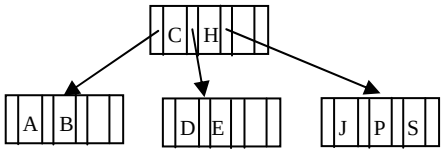
Al insertar S se provoca la primera división en el nodo y la promoción de H. Luego podemos insertar B en el nodo más a la izquierda sin que ello provoque ninguna reestructuración



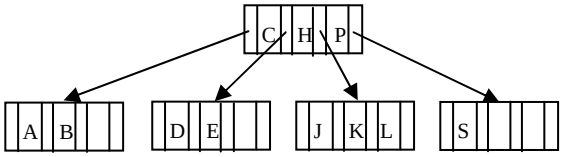
La inserción de A provoca otra nueva división y como consecuencia la promoción de C



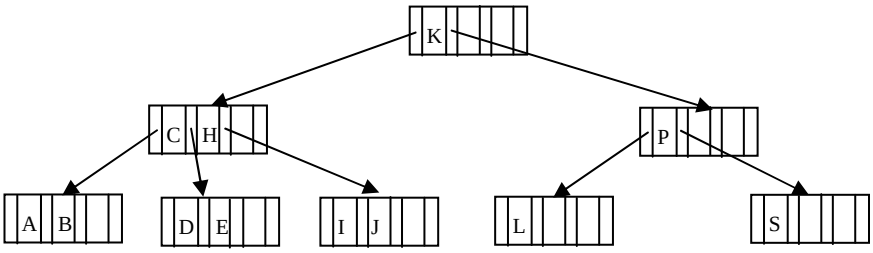
En los nodos que ya existen insertamos sin problema las claves P, D y J



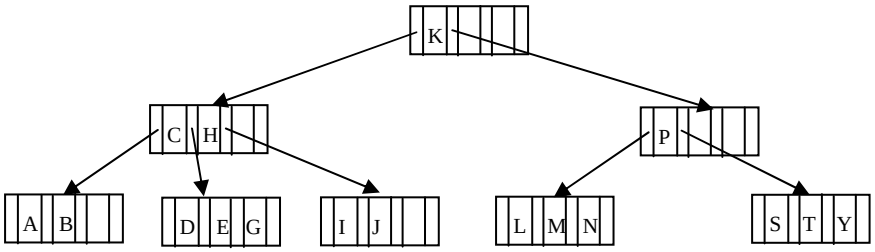
Insertando L, provocamos otra nueva división y la promoción de P. Seguidamente, en los nodos que tenemos podemos insertar K



La inserción de la clave I provoca una división en un nodo hoja, que a vez va a provocar una división en la raíz, quedando el árbol de la siguiente forma:



Finalmente en los nodos existentes podemos insertar las claves que nos quedan, G, M, T, N e Y, sin necesidad de hacer más divisiones, resultando el siguiente árbol B:



10.2 Eliminación en un árbol B

El algoritmo de eliminación de claves en un árbol B sería el siguiente:

1. Si la clave que se va a eliminar no está en una hoja, intercambiarla con su sucesor más cercano según el orden lineal de las claves, el cuál se encontrará siempre en una hoja.

2. Eliminar la clave.

3. Si el nodo del que se ha suprimido la clave contiene por lo menos el número mínimo de claves, se termina.

4. Si este nodo contiene una menos de las claves mínimas, comprobar los hermanos izquierdo y derecho.

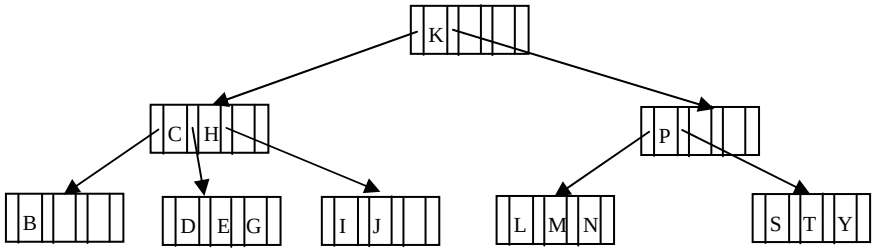
    a. Si existe un hermano que tenga más del número mínimo de claves, redistribuir las claves con este hermano.

    b. Si no existe un hermano que tenga más del número mínimo de claves, se concatenan los dos nodos junto con la clave del nodo padre que separa a ambos.

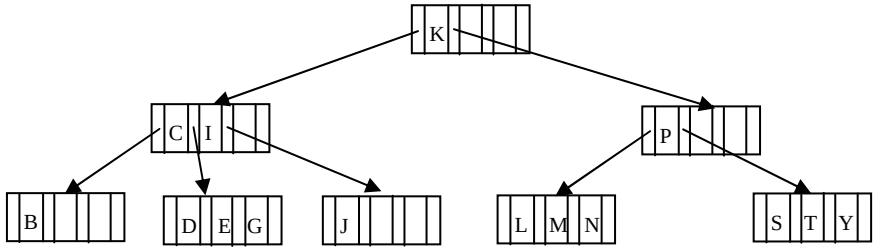
5. Si se han concatenado dos nodos, volver a aplicar al nodo padre desde el paso 3, excepto cuando el padre sea el nodo raíz, en cuyo caso, el árbol queda con un nivel menos.

A continuación mostramos un ejemplo en el cuál quitamos las claves A, H, B y J al árbol obtenido anteriormente.

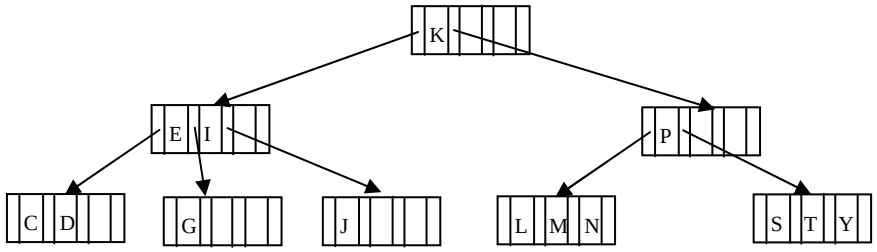
Eliminar la clave A es el caso más simple, ya que está en una hoja y su eliminación no implica nada más.



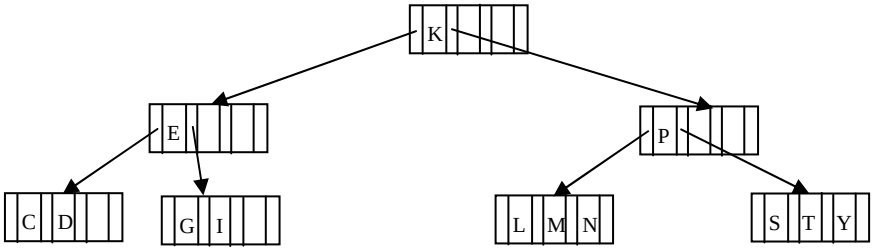
Para eliminar H, hay que intercambiarla con su sucesor (I), puesto que no está en un nodo hoja. A continuación eliminamos de la hoja y volvemos a estar en el caso anterior.



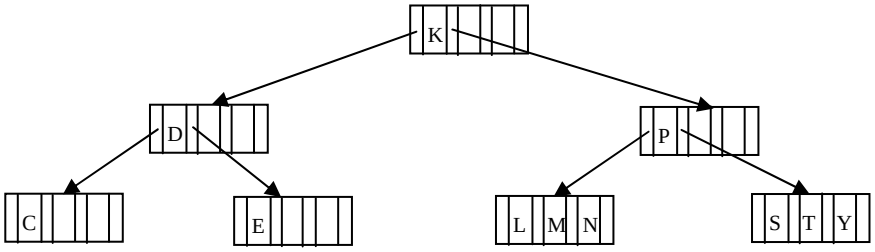
Si eliminamos B se nos queda un nodo vacio, es decir, con una clave menos del mínimo permitido, por lo tanto, debemos redistribuir las claves de su hermano derecho.



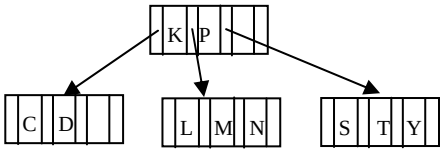
Al eliminar J, el nodo queda vacio, y no es posible redistribuir las claves de su hermano izquierdo porque no tiene más del mínimo, de modo que estamos obligados a concatenar los nodos.



Eliminando las claves G e I se nos queda el siguiente árbol



Si eliminamos E se nos queda el nodo vacio, y no podemos redistribuir con su hermano izquierdo, por tanto hay que concatenar. Al concatenar C y D, se nos queda vacio el nodo padre, otra vez no podemos redistribuir las claves y volvemos a concatenar quedándonos con un nivel menos en el árbol.



TEMA 6: GRAFOS

1. INTRODUCCIÓN

Un ***grafo*** $G = (V, A)$ consta de un conjunto de ***vértices*** o ***nodos***, V, y un conjunto de ***aristas*** o ***arcos*** $A \subseteq (V \times V)$ que define una relación binaria en V. Cada arista es, por tanto, un par de vértices $(v, w) \in A$.

Si cada arista $(v, w) \in A$ es un par ordenado, es decir, si (v, w) y (w, v) no son equivalentes, entonces el grafo es ***dirigido*** y la arista (v, w) se representa como una flecha de v a w. El vértice v se dice que es *incidente* sobre el vértice w y w es *adyacente* a v.

Si, por el contrario, cada arista es un par no ordenado de vértices y por tanto $(v, w) = (w, v)$, entonces el grafo es ***no dirigido*** y la arista (v, w) se representa como un segmento entre v y w. En este caso, se dice que v y w son *adyacentes* y la arista (v, w) es incidente sobre v y w.

Una arista puede tener un valor asociado, llamado ***peso***, que representa un tiempo, una distancia, un coste, etc. Un grafo cuyas aristas tienen pesos asociados recibe el nombre de ***grafo ponderado***.

2. CONCEPTOS BÁSICOS

Grado: El grado de un vértice en un grafo no dirigido es el número de arcos del vértice. Si el grafo es dirigido, se distingue entre *grado de entrada* (número de arcos incidentes en el vértice) y *grado de salida* (número de arcos adyacentes al vértice).

Camino: Una sucesión de vértices de un grafo $n_1, n_2, ..., n_k$, tal que (n_i, n_{i+1}) es una arista para $1 \leq i \leq k$. La *longitud* de un camino es el número de arcos que comprende, en este caso k-1. Si el grafo es ponderado la longitud de un camino se calcula como la suma de los pesos de las aristas que lo constituyen.

Camino simple: Un camino cuyos arcos son todos distintos. Si además todos los vértices son distintos, se llama *camino elemental*.

Ciclo: Es un camino en el que coinciden los vértices inicial y final. Si el camino es simple, el ciclo es *simple* y si el camino es elemental, entonces el ciclo se llama *elemental*. Se permiten arcos de un vértice a sí mismo; si un grafo contiene arcos de la forma (v, v), lo cual no es frecuente, estos son ciclos de longitud 1; de lo contrario y como caso especial, un vértice v por sí mismo denota un camino de longitud 0

Grafo conexo: Grafo no dirigido en el que hay al menos un camino entre cualquier par de vértices.

Grafo fuertemente conexo: Grafo dirigido en el que hay al menos un camino entre cualquier par de vértices. Si un grafo dirigido no es fuertemente conexo, pero el grafo no dirigido subyacente (sin dirección en los arcos) es conexo, entonces es *débilmente conexo*.

Grafo completo: Aquel en el cual existe una arista entre cualquier par de vértices (en ambos sentidos si el grafo es dirigido).

Subgrafo: Dado un grafo $G = (V, A)$, diremos que $G' = (V', A')$, donde $V' \subseteq V$ y $A' \subseteq A$, es un subgrafo de G si A' contiene sólo las aristas de A que unen dos vértices de V'.

Un **componente conexo** de un grafo no dirigido G es un subgrafo conexo maximal, es decir, un subgrafo conexo que no es subgrafo de ningún otro subgrafo conexo de G. Análogamente se define *componente fuertemente conexo* de un grafo dirigido.

3. REPRESENTACIONES DE GRAFOS

3.1 Matriz de adyacencia

Dado un grafo $G = (V, A)$ con n vértices, se define la matriz de adyacencia asociada a G como una matriz $M_{n \times n}$ donde

$$M_{i,j} = 1 \text{ si } (i, j) \in A \quad \text{y} \quad M_{i,j} = 0 \text{ si } (i, j) \notin A$$

Si G es un grafo no dirigido, M es una matriz simétrica ya que $(i, j) = (j, i)$ para cualesquiera vértices i, j.

```
#define N 100 /* Número máximo de vértices */
typedef enum {FALSE, TRUE} boolean;
typedef int vertice; /* un valor entre 0 y
                            numVert-1 */
typedef struct {
                boolean Ady[N][N];
                int numVert;
            } tGrafo;
typedef tGrafo *Grafo;
```

3.2 Matriz de costes

Dado un grafo $G = (V, A)$ con n vértices, se define la matriz de costes asociada a G como una matriz $C_{n \times n}$ donde

$$C_{i,j} = p \text{ si } (i, j) \in A, \text{ siendo } p = \text{peso asociado a } (i, j)$$

$C_{i,j} = peso\_ilegal$ si $(i, j) \notin A$, (*peso_ilegal* es un valor no válido como peso de un arco).

```
#define N 100 /* Número máximo de vértices */
typedef int vertice; /* un valor entre 0 y
                          numVert-1 */
typedef unsigned tCoste; /* valor asociado a un arco */
typedef struct {
                tCoste Costes[N][N];
                int numVert;
            } tGrafo;
typedef tGrafo *Grafo;
```

3.3 Listas de adyacencia

La idea es asociar a cada vértice i del grafo una lista que almacene todos los vértices adyacentes a i.

3.3.1 Vector de listas de adyacencia:

```
typedef int vertice; /* índice del vector
                        entre 0 y numVert-1 */
typedef struct {
               ListaAdy *adyacentes; /* vector de
listas */
               int maxVert;
               int numVert;
            } tGrafo;
typedef tGrafo *Grafo;
```

`tElemento` en el TAD ListaAdy se define como sigue:

a) *Grafos no ponderados*:

```
typedef vertice tElemento;
```

b) *Grafos ponderados*:

```
typedef struct {
        vertice vert;
        tCoste coste;
      } tElemento;
```

3.3.2 Lista de listas de adyacencia:

```
typedef ListaVert Grafo;
```

tElemento en el TAD ListaVert se define como sigue:

```
typedef struct {
       vertice vert;
       ListaAdy adyacentes;
      } tElemento;
```

Ventajas e inconvenientes:

- Las matrices de adyacencia y costes son muy eficientes para comprobar si existe una arista entre un vértice y otro.
- Pueden desaprovechar gran cantidad de memoria si el grafo no es completo.
- Tiene limitación para el número máximo de vértices, con lo cual, cuando el número real de vértices es inferior al máximo, se puede desaprovechar una cantidad considerable de memoria.
- La representación mediante listas de adyacencia aprovecha mejor el espacio de memoria, pues sólo se representan los arcos existentes en el grafo.
- Cuando se utiliza una lista de listas es posible añadir y suprimir vértices.
- Las listas de adyacencia son poco eficientes para determinar si existe una arista entre dos vértices del grafo.

4. RECORRIDOS DE GRAFOS

En el caso de los recorridos en un grafo surge un problema nuevo que no había aparecido en ninguna estructura anterior. Básicamente, dado que no existen reglas definidas en la conexión entre nodos de un grafo (no existe secuencialidad, como en el caso de las listas, ni existen jerarquías, como en el caso de los árboles), nada nos impide, por tanto, meternos en un ciclo. No existen reglas para evitar que esto ocurra, por lo que la solución para evitar recorrer en más de una ocasión el mismo nodo, pasa por marcarlo de alguna forma como visitado. La idea no es nueva, ya surge en la literatura infantil con el cuento de Pulgarcito. Por suerte en la memoria del ordenador no hay pájaros, como en el cuento, que se coman nuestras "migas de pan" (marcas).

```
typedef enum {NO_VISITADO,VISITADO} visitas;


void Profundidad (Grafo G)
{
     visitas *marcas;
     vertice i;
     marcas = calloc(G->numVert, sizeof(visitas));
     if (marcas == NULL)
          ERROR("...: No hay memoria");
     for (i = 0; i < G->numVert; i++)
          if (marcas[i] == NO_VISITADO)
               Profun(i, G, marcas);
     free(marcas);
}


void Profun(vertice v,Grafo G,visitas *marcas)
{
     vertice w;
     marcas[v] = VISITADO;
     printf("%d ", v); /* Procesar v */
     for (w = 0; w < G->numVert; w++)
          if (G->Ady[v][w] == TRUE &&
                    marcas[w] == NO_VISITADO)
               Profun(w, G, marcas);
}
```

```
void Profundidad2 (Grafo G)
{
     visitas *marcas;
     Pila P; /* Pila de vértices */
     vertice i, v, w;

     P = CrearPila();
     marcas = calloc(G->numVert, sizeof(visitas));
     if (marcas == NULL)
        ERROR("Profundidad2(): No hay memoria");
     for (i = 0; i < G->numVert; i++)
          if (marcas[i] == NO_VISITADO) {
             Push(i, P);
             do {
               v = Tope(P); Pop(P);
               if (marcas[v] == NO_VISITADO) {
                  /* Marcar y procesar v */
                  marcas[v] = VISITADO;
                  printf("%d ", v);
                  /* Meter en la pila los
                     adyacentes no visitados */
                 for (w = 0; w < G->numVert; w++)
                    if (G->Ady[v][w] == TRUE &&
                       marcas[w] == NO_VISITADO)
                         Push(w, P);
               }
             } while (!Vacia(P));
          } /* for if */
     free(marcas);
     DestruirPila(P);
}
```

```
void Anchura (Grafo G)
{
     visitas *marcas;
     Cola C; /* Cola de vértices */
     vertice i, v, w;

     C = CrearCola();
     marcas = calloc(G->numVert, sizeof(visitas));
     if (marcas == NULL)
          ERROR("Anchura(): No hay memoria");
     for (i = 0; i < G->numVert; i++)
          if (marcas[i] == NO_VISITADO) {
             ColaPush(i, C);
             do {
               v = Frente(C); ColaPop(C);
               if (marcas[v] == NO_VISITADO) {
                  /* Marcar y procesar v */
                  marcas[v] = VISITADO;
                  printf("%2d ", v);
                  /* Meter en la cola los
                        adyacentes no visitados */
                  for (w = 0; w < G->numVert; w++)
                    if (G->Ady[v][w] == TRUE &&
                          marcas[w] == NO_VISITADO)
                            ColaPush(w, C);
               }
             } while (!ColaVacia(C));
          } /* for if */
     free(marcas);
     DestruirCola(C);
}
```

5. ALGORITMOS DEL CAMINO MÁS CORTO

Un problema muy común en las aplicaciones de grafos consiste en determinar el coste o longitud del camino más corto entre dos vértices de un grafo. En realidad, este problema es tan difícil como determinar el coste de los caminos de coste mínimo desde un vértice origen a todos los demás vértices del grafo y después siempre podemos seleccionar el valor correspondiente al vértice destino que nos interese, o bien, si lo preferimos podemos terminar el algoritmo en el momento en que se encuentre el coste del camino que nos interesa. Por tanto, el siguiente algoritmo que recibe el nombre de su autor Dijkstra, resuelve el problema más general de encontrar el coste mínimo de los caminos desde un vértice origen hasta todos los vértices de un grafo ponderado

```
void Dijkstra (vertice origen, Grafo G,
                        tCoste **D, vertice **P)
/* Calcula los caminos de coste mínimo entre
     origen y todos los vértices del grafo G.
     Salida:
          - Un vector *D de tamaño G->numVert con
          estos costes mínimos.
          - Un vector *P de tamaño G->numVert tal
          que (*P)[i] es el último vértice del
          camino de origen a i.                */
{
     boolean *S;
     int i;
     vertice v, w;
     tCoste CosteMin, Owv;

     S = calloc(G->numVert, sizeof(boolean));
     if (S == NULL)
          ERROR("Dijkstra(): No hay memoria");

     S[origen] = TRUE; /* Incluir origen en S */
     *D = calloc(G->numVert, sizeof(tCoste));
     if (*D == NULL)
          ERROR("Dijkstra(): No hay memoria");
```

```
    *P = calloc(G->numVert, sizeof(vertice));
    if (*P == NULL)
        ERROR("Dijkstra(): No hay memoria");

    /* Inicializar *D y *P */
    for (v = 0; v < G->numVert; v++) {
        (*D)[v] = G->Costes[origen][v];
        (*P)[v] = origen;
    }
    for (i = 0; i < G->numVert-1; i++) {
        /* Localizar vértice w no incluido en S
           con coste mínimo desde origen */
        CosteMin = INFINITO;
        for (v = 0; v < G->numVert; v++)
            if (!S[v] && (*D)[v] < CosteMin) {
                CosteMin = (*D)[v];
                w = v;
            }
        S[w] = TRUE; /* Incluir w en S */
        /* Recalcular coste hasta cada v
           no incluido en S a través de w. */
        for (v = 0; v < G->numVert; v++) {
           Owv = Suma((*D)[w], G->Costes[w][v]);
           if (!S[v] && (*D)[v] > Owv) {
             (*D)[v] = Owv;
             (*P)[v] = w;
           }
        }
    }
    free(S);
}
```

```
void Caminoi (vertice orig, vertice i, vertice *P)
/* Reconstruye el camino de orig a i a partir
     de un vector P obtenido mediante la función
     Dijkstra(). */
{
     if (P[i] != orig) {
          Caminoi(orig, P[i], P);
          printf("%2d ", P[i]);
     }
}
```

En ciertos casos es necesario determinar el coste de los caminos de coste mínimo entre cualquier par de vértices del grafo. Este es el problema de los caminos de coste mínimo entre todos los pares. Se puede resolver utilizando el algoritmo de Dijkstra con cada vértice del grafo, pero existe un método más directo mediante el algoritmo de Floyd.

```
void Floyd (Grafo G, tCoste A[][N], vertice P[][N])

/*Devuelve una matriz de costes mínimos A de tamaño NxN
y una matriz de vértices P de tamaño NxN, tal que
P[i][j] es el vértice por el que pasa el camino de
coste mínimo de i a j, o bien es -1 si este camino es
directo*/

{
     vertice i, j, k;
     tCoste ikj;

     for (i = 0; i < G->numVert; i++)
          for (j = 0; j < G->numVert; j++) {
               A[i][j] = G->Costes[i][j];
               P[i][j] = -1;
          }
```

```
for (i = 0; i < G->numVert; i++)
          A[i][i] = 0;
     for (k = 0; k < G->numVert; k++)
          for (i = 0; i < G->numVert; i++)
             for (j = 0; j < G->numVert; j++) {
               ikj = Suma(A[i][k], A[k][j]);
               if (A[i][j] > ikj) {
                  A[i][j] = ikj;
                  P[i][j] = k;
               }
             }
}

void Camino (vertice i, vertice j,
                         vertice P[][N])
/* Reconstruye el camino de i a j a partir
     de una matriz P obtenida mediante la
     función Floyd(). */

{
     vertice k;

     k = P[i][j];
     if (k != -1) {
          Camino(i, k, P);
          printf("%2d ", k);
          Camino(k, j, P);
     }
}
```

Para otros problemas es suficiente conocer si existe un camino entre cualquier par de vértices. Esto se puede conseguir con una pequeña modificación del algoritmo de Floyd, dando lugar al algoritmo de Warshall.

```
void Warshall (Grafo G, boolean A[][N])
/* Determina si hay un camino entre cada par
     de vértices del grafo G. Devuelve una
     matriz booleana A de tamaño NxN, tal que
     A[i][j] == TRUE si existe al menos un
     camino entre el vértice i y el vértice j,
     y A[i][j] == FALSE si no existe ningún
     camino entre los vértices i y j. */
{
     vertice i, j, k;

     /* Inicializar A con la matriz de
          adyacencia de G */
     for (i = 0; i < G->numVert; i++)
          for (j = 0; j < G->numVert; j++)
             A[i][j] = G->Ady[i][j];
     /* Comprobar camino entre cada par de
     vértices i, j a través de cada vértice k */
     for (k = 0; k < G->numVert; k++)
          for (i = 0; i < G->numVert; i++)
             for (j = 0; j < G->numVert; j++)
               if (!A[i][j])
                  A[i][j] = A[i][k] && A[k][j];
}
```

6. ÁRBOLES GENERADORES DE COSTE MÍNIMO

Un problema característico de grafos se plantea en el diseño de redes de comunicación, donde los vértices representan nodos de la red y las aristas, las líneas de comunicación entre los mismos. El peso asociado a cada arista representa el coste de establecer esa línea de la red.

La cuestión es seleccionar el conjunto de líneas que permitan la comunicación entre todos los nodos de la red, tal que el costo total de la red diseñada sea mínimo.

La solución de este problema se puede obtener hallando un árbol generador de coste mínimo para el grafo que comprenda todas las líneas posibles de comunicación de la red.

Dado un grafo no dirigido y conexo G = (V, A), se define un **árbol generador de G** como un árbol que conecta todos los vértices de V; su coste es la suma de los costes de las aristas del árbol. Un árbol es un grafo conexo acíclico.

Existen dos algoritmos muy conocidos para construir un árbol de extensión de coste mínimo a partir de un grafo ponderado. Estos se deben a Prim y Kruskall.

6.1. Algoritmo de Prim

```
void Prim (Grafo G, arista **T)
/* Devuelve en un vector *T el conjunto de aristas que
forman un árbol generador de coste mínimo de un grafo
conexo G. */
{
     boolean *U;
     vertice j, k;
     int i;
     arista a;
     tCoste CosteMin;

     *T = calloc(G->numVert-1, sizeof(arista));
     if (*T == NULL)
          ERROR("Prim(): No hay memoria");

     U = calloc(G->numVert, sizeof(boolean));
     if (U == NULL)
          ERROR("Prim(): No hay memoria");

     U[0] = TRUE;
     for (i = 0; i < G->numVert-1; i++) {
          /* Buscar una arista a=(u, v) de coste
             mínimo, tal que u está ya en el
             conjunto U y v no está en U. */
          CosteMin = INFINITO;
```

```
            for (j = 0; j < G->numVert; j++)
              for (k = 0; k < G->numVert; k++)
                if (U[j] && !U[k])
                  if (G->Costes[j][k] <= CosteMin){
                      CosteMin = G->Costes[j][k];
                      a.orig = j;
                      a.dest = k;
                  }
             /* Incluir a en *T y v en U */
             (*T)[i] = a;
             U[a.dest] = TRUE;
      }
}
```

6.2. Algoritmo de Kruskall

TAD *Partición*

Una ***partición*** de un conjunto C de elementos de un cierto tipo es un conjunto de subconjuntos disjuntos cuya unión es el conjunto total C.

Partiendo de esta definición, nuestro objetivo es crear un TAD general que nos permita trabajar con particiones de cualquier conjunto finito C. En lugar de crear directamente este TAD general, crearemos un TAD para representar particiones solamente del conjunto de los números enteros en el intervalo [0, $n-1$] (donde n es el número de elementos de C). Este segundo TAD lo podremos utilizar para representar particiones de cualquier conjunto C de n elementos, simplemente definiendo una aplicación entre los elementos de C y el rango de enteros [0, $n-1$], tal que cada elemento se aplique en un único número y cada número corresponda a un solo elemento. Esta aplicación estará implementada mediante dos funciones externas al TAD cuyas especificaciones son las siguientes:

int IndiceElto (tElemento x);
Pre: $x \in C$.
Post: Devuelve el índice del elemento x en el rango [0, $n-1$].

tElemento NombreElto (int i);
Pre: $0 \leq i \leq n-1$
Post: Devuelve el elemento de C cuyo índice es i.

Una *relación de equivalencia* sobre los elementos de un conjunto C define una partición de C y, viceversa, cualquier partición de C define una relación de equivalencia sobre sus elementos, de tal forma que cada miembro de la partición es una clase de equivalencia. Así pues, para cada subconjunto o clase podemos elegir cualquier elemento como representante canónico de todos sus miembros.

A continuación se da la especificación del TAD *Partición* teniendo en cuenta las consideraciones anteriores.

Definición:

Una partición del conjunto de enteros $C = \{0, 1,\ldots, n-1\}$ es un conjunto de subconjuntos disjuntos cuya unión es el conjunto total C.

Operaciones:

Particion CrearParticion (int n);

Post: Construye y devuelve una partición del intervalo de enteros $[0, n-1]$ colocando un solo elemento en cada subconjunto.

void Union (int a, int b, Particion P);

Pre: La partición P está inicializada y $0 \leq a, b \leq n-1$ (a y b son los representantes de sus clases).

Post: Une el subconjunto del elemento a y el del elemento b en uno de los dos subconjuntos arbitrariamente. La partición P queda con un miembro menos.

int Encontrar (int x, Particion P);

Pre: La partición P está inicializada y $0 \leq x \leq n-1$.

Post: Devuelve el representante del subconjunto al que pertenece el elemento x.

void DestruirParticion (Particion P);

Post: Destruye la partición P, liberando el espacio ocupado en memoria.

Para la implementación del TAD *Partición* analizaremos diferentes estructuras de datos alternativas.

1. Vector de pertenencia

La estructura de datos más sencilla que se puede utilizar para representar una partición P del conjunto $C = \{0, 1,\ldots, n-1\}$ es un vector de enteros de tamaño n, tal que en la posición i-ésima se almacena el representante de la clase a la que pertenece i. Obviamente, la operación *Encontrar()* es *O(1)*, mientras que *CrearParticion()* y *Union()* son *O(n)*. La eficiencia de la operación constructora no es posible mejorarla, ya que debe crear n subconjuntos unitarios, sin embargo sí que podemos modificar la estructura de datos para hacer la unión más eficiente.

2. Listas de elementos

El punto débil de la unión es que hay que recorrer todo el vector en busca de todos los elementos de la clase de *b* para asignarles el representante de la clase de *a*, o viceversa. Una posibilidad para evitar este recorrido es enlazar todos los miembros de una clase en una lista cuyo principio sea el representante de la clase, añadiendo un campo a cada celda del vector para almacenar el siguiente elemento de la lista (utilizamos −1 para indicar el final de una lista). Ahora, en vez de recorrer el vector completo, basta recorrer la lista de los elementos de una clase para asignarles el representante de la otra y enlazar ambas listas en una sola.

Sería deseable, además, recorrer siempre la lista más corta, pero para eso necesitamos conocer el tamaño de ambas listas, así que podemos añadir otro campo más a cada celda para guardar el tamaño de la clase a la que pertenece el elemento. Pero entonces, hay que recorrer las dos listas para actualizar este valor con la suma de los tamaños de las clases que se unen. Por lo tanto, para evitar el recorrido de ambas listas conviene guardar la longitud de cada lista solamente en el elemento representante (los valores almacenados para los demás elementos son irrelevantes).

Con esta estructura de datos conseguimos reducir el tiempo de ejecución de la unión de dos conjuntos. Si unimos dos listas de elementos en una de ellas, entonces el tiempo será proporcional al número de elementos de la lista recorrida. Sin tener en cuenta la longitud, puede ocurrir que la lista recorrida sea la más larga. El caso peor se dará cuando se combine una clase unitaria con otra en la que estén el resto de los elementos del conjunto total, el tiempo será casi el mismo que cuando se utilice simplemente un vector de pertenencia. En todos los demás casos, la ganancia de tiempo será algo mayor, pero a costa de emplear un campo más por cada elemento para almacenar las listas.

Por otra parte, si consideramos la longitud de la listas a unir, la peor situación siempre se dará cuando ambas tengan la misma longitud. Entonces el caso extremo se presentará cuando la partición conste de dos subconjuntos con la mitad de los elementos cada uno. En tal caso habrá que recorrer cualquiera de las dos listas, pero se tardará la mitad de tiempo que en recorrer el vector entero (la mejora es mayor que antes). Además, en este caso, se necesita el mismo tiempo que sin considerar la longitud de las listas, pero en todos los demás casos el tiempo de ejecución nunca será mayor, porque siempre se recorre la lista más corta.

3. Bosque de árboles

La causa por la que la operación *Unión()* no se ejecuta en un tiempo constante es que para cada elemento se almacena el representante de su clase y esto obliga a modificar el representante de todos los elementos de uno de los cojuntos unidos. Podemos cambiar la estructura de datos para conseguir que la unión sea *O(1)*, pero a costa de empeorar el tiempo de ejecución de la operación *Encontrar()*, ya que no existe una estructura de datos que permita ejecutar simultáneamente estas dos operaciones en un tiempo constante.

La idea consiste en formar un árbol con todos los elementos de una clase y elegir la raíz como representante[1]. Así pues, una partición se representa como un bosque de árboles, cada uno de los cuales es un subconjunto de la partición. Un árbol se puede representar simplemente enlazando cada nodo con su padre y, por tanto, podemos almacenarlo en un vector de enteros en el que la posición *i*-ésima guarda el padre del elemento *i* o −1, si *i* es la raíz. De esta forma, la unión de conjuntos se convierte en la fusión de dos árboles, para lo cual no es necesario ningún recorrido, basta enlazar las dos raíces haciendo que una sea hija de la otra, con lo cual la operación *Unión()* claramente es *O(1)*. Sin embargo, determinar la clase a la que pertenece un elemento implica ascender por el árbol hasta la raíz y devolver ésta, por lo que el tiempo de la operación *Encontrar()* será proporcional a la profundidad del nodo que representa al elemento. Como la estrategia de unión descrita puede producir un árbol de máxima profundidad, es decir, con todos los nodos en una rama (esta situación se presenta cuando repetidas veces se unen dos conjuntos, uno de ellos unitario y éste siempre se convierte en raíz), entonces *Encontrar()* será *O(n)* en el peor caso.

```
/*--------------------------------------*/
/* particion.h                          */
/*--------------------------------------*/

#ifndef _PARTICION_
#define _PARTICION_
  typedef struct {
             int *padre;
             int nEltos; /* Tamaño cjto */
           } tipoParticion;
  typedef tipoParticion *Particion;

  Particion CrearParticion (int n);
  void Union(int a, int b, Particion P);
  int Encontrar (int x, Particion P);
  void DestruirParticion (Particion P);
#endif
```

[1] Obsérvese que esta estructura es muy parecida a la anterior, la cual podemos ver como un bosque de árboles degenerados en listas con raíz en el primer elemento.

```
/*--------------------------------------*/
/* particion1.c                         */
/*--------------------------------------*/

#include <stdlib.h>
#include "error.h"
#include "particion.h"

Particion CrearParticion (int n)
{
   Particion P;
   int i;

   P = (Particion) malloc(sizeof(tipoParticion));
   if (P == NULL)
      ERROR("CrearParticion: No hay memoria");

   P->padre = (int *) malloc(n * sizeof(int));
   if (P->padre == NULL)
      ERROR("CrearParticion: No hay memoria");

   for (i = 0; i < n; i++)
      P->padre[i] = -1;
   P->nEltos = n;
   return P;
}

void Union (int a, int b, Particion P)
{
   P->padre[b] = a;
}
```

```
int Encontrar (int x, Particion P)
{
   while (P->padre[x] != -1)
      x = P->padre[x];
   return x;
}

void DestruirParticion (Particion P)
{
   free(P->padre);
   free(P);
}
```

Conseguir reducir el tiempo de la búsqueda requiere modificar el procedimiento de unión, procurando que la altura de los árboles se mantenga lo más pequeña posible en todo momento. Existen dos enfoques para lograrlo: a) *unión por tamaño*: el árbol con menos nodos se convierte en subárbol del que tiene mayor número de nodos; b) *unión por altura*: el árbol menos alto se convierte en un subárbol del otro. Estos dos algoritmos realizan la unión de dos conjuntos en un tiempo *O(1)* y permiten localizar al representante de una clase en un tiempo *O(log n)* en el peor caso. Veámoslo para la unión por tamaño. Inicialmente cada nodo está a profundidad 0 y como un nodo sólo puede descender de nivel cuando su clase se une a otra mayor o igual, entonces su profundidad nunca podrá ser mayor que *log n* (número máximo de veces que se puede unir la clase de un elemento *x* con otra de igual tamaño hasta obtener una partición con un único conjunto). Se puede hacer un análisis similar para la unión por altura y llegar a la misma conclusión. Por tanto, en el peor de los casos la operación *Encontrar()* es *O(log n)*.

Para implementar esta estrategia es necesario guardar el tamaño o la altura de cada árbol. Podemos hacerlo almacenando en la raíz de cada árbol el valor correspondiente en negativo, así no es necesario utilizar espacio adicional y podemos seguir identificando los nodos raíces en la operación *Encontrar()*. No obstante, en la unión por altura almacenaremos el opuesto de la altura menos 1, para distinguir un árbol de altura 0 de un nodo cuyo padre es el nodo 0.

La implementación del TAD *Partición* con unión por tamaño o altura es bastante aceptable para la mayoría de las aplicaciones, pero aún podemos ganar eficiencia modificando también la operación *Encontrar()* para reducir la altura del árbol a la vez que se asciende hasta la raíz. El fundamento de esta idea es que la próxima vez que haya que determinar la clase a la que pertenece un elemento, algunos nodos del árbol estarán a menor profundidad y por tanto el tiempo requerido para alcanzar la raíz desde ellos será menor. Dicho de otra forma, el objetivo tras varias búsquedas es acercarnos lo máximo posible a la situación ideal

en la que todos los árboles tienen altura 1 y por tanto las posteriores operaciones de búsqueda tardarán un tiempo casi constante. Esta técnica, que es independiente del método de unión que se utilice, se conoce como *compresión de caminos* y se implementa haciendo que todos los nodos por los que se pasa durante una búsqueda se transformen en hijos de la raíz del árbol al que pertenecen.

```
/*-------------------------------------*/
/* particion.c                         */
/*-------------------------------------*/

/* TAD Paticion mediante unión por altura
   y búsqueda con compresión de caminos */

#include <stdlib.h>
#include "error.h"
#include "particion.h"

Particion CrearParticion (int n)
{
   Particion P;
   int i;

   P = (Particion) malloc(sizeof(tipoParticion));

   if (P == NULL)
      ERROR("CrearParticion: No hay memoria");

   P->padre = (int *) malloc(n * sizeof(int));
   if (P->padre == NULL)
      ERROR("CrearParticion: No hay memoria");

   /* Inicialmente hay n árboles de altura 0
      representada con -1 */
   for (i = 0; i < n; i++)
      P->padre[i] = -1;
```

```
   P->nEltos = n;
   return P;
}

void Union (int a, int b, Particion P)
{
   if (P->padre[b] < P->padre[a])
      P->padre[a] = b;
   else {
      if (P->padre[a] = P->padre[b])
         P->padre[a]--; /* el árbol resultante tiene un
                           nivel más */
      P->padre[b] = a;
   }
}

int Encontrar (int x, Particion P)
{
   int raiz, y;

   raiz = x;
   while (P->padre[raiz] > -1)
      raiz = P->padre[raiz];
   /* los nodos del camino de x a raiz
      se hacen hijos de raiz */
   while (P->padre[x] > -1) {
      y = P->padre[x];
      P->padre[x] = raiz;
      x = y;
   }
   return raiz;
}
```

```
void DestruirParticion (Particion P)
{
   free(P->padre);
   free(P);
}
```

Implementación del algoritmo de Kruskall

Previamente definiremos la estructura `struct aris` empleada en el algoritmo:

```
     struct aris {
               tCoste c;
               arista a;
     }
```

```
void Kruskall (Grafo G, arista **T)
/* Devuelve en un vector *T el conjunto de
     aristas que forman un árbol generador de
     coste mínimo de un grafo conexo G. */
{
     int a, i, j;
     Particion P;  /* Partición del conjunto de
                      vértices de G */
     vertice u, v;
     arista e;
     struct aris *H; /* Vector de aristas de G
                        ordenadas por costes */

     *T = calloc(G->numVert-1, sizeof(arista));
     if (*T == NULL)
          ERROR("Kruskall(): No hay memoria");

     H = calloc(G->numVert * G->numVert,
                   sizeof(struct aris));
     if (H == NULL)
          ERROR("Kruskall(): No hay memoria");
```

```
    P = CrearParticion(G->numVert);
    a = 0;
    for (u = 0; u < G->numVert; u++) {
        for (v = u+1; v < G->numVert; v++)
            if (G->Costes[u][v] != INFINITO) {
                H[a].a.orig = u;
                H[a].a.dest = v;
                H[a].c = G->Costes[u][v];
                a++;
            }
    }
    qsort((void *) H, a, sizeof(struct aris),
                                        CompAris);
    i = 0; a = 0;
    while (i < G->numVert-1) {
        e = H[a].a;
        u = Encontrar(e.orig, P);
        v = Encontrar(e.dest, P);
        if (u != v) { /* Los extremos de e pertenecen
                         a clases distintas */
            Union(u, v, P);
            /* Incluir e en *T */
            (*T)[i] = e;
            i++;
        }
        a++;
    } /* while */
    DestruirParticion(P);
    free(H);
}
```

```
int CompAris (const void *a, const void *b)
{
     if (((struct aris *) a)->c <
           ((struct aris *) b)->c)
          return -1;
     else if (((struct aris *) a)->c ==
                ((struct aris *) b)->c)
          return 0;
     else
          return 1;
}
```