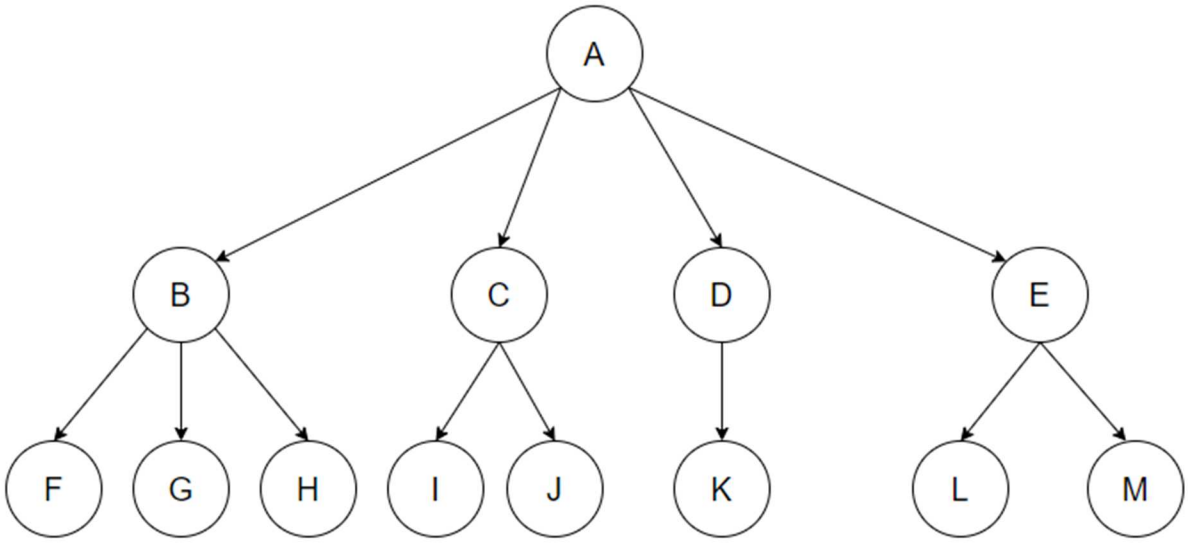
Objetivo: Presentar el tema de Árboles. Presentar una implementación de un TDA de Árbol Binario de Búsqueda.

Temas: Árboles

Árboles en programación

Desde el punto de vista matemático, un Árbol es un tipo especial de grafo donde hay exactamente un solo camino parara llegar a cualquiera de los nodos que lo componen, y existe una jerarquía entre los mismos.



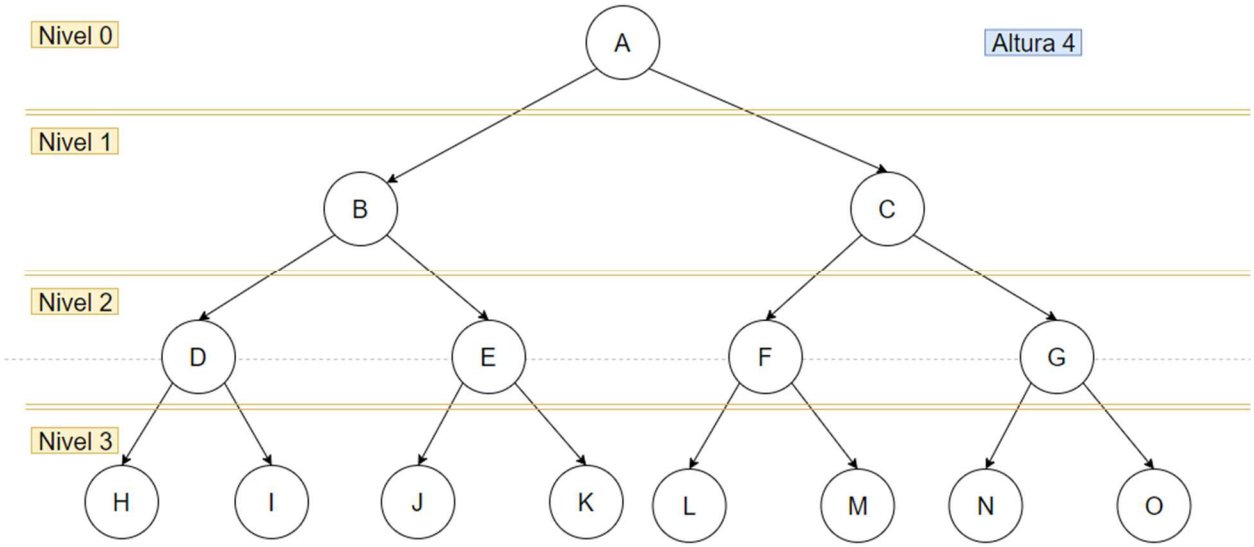
También hay un solo punto de entrada al árbol. A este nodo le llamaremos raíz, y en nuestro ejemplo es el nodo A. Es el único nodo en todo el árbol que no tiene padre.

Todos los nodos de deriven del nodo raíz serán sus hijos, y, a su vez, esos hijos serán árboles en sí mismos. Todos los nodos que tengan el mismo padre en común serán hermanos,

Cada nodo puede tener de 0 a N hijos en un árbol libre, pero nosotros nos concentraremos para este apunte en los árboles binarios.

Árboles binarios

Estos árboles solo admiten de 0 a 2 hijos por nodo. Y más aún, está bien diferenciado entre el nodo que está a la izquierda y el que está a la derecha.

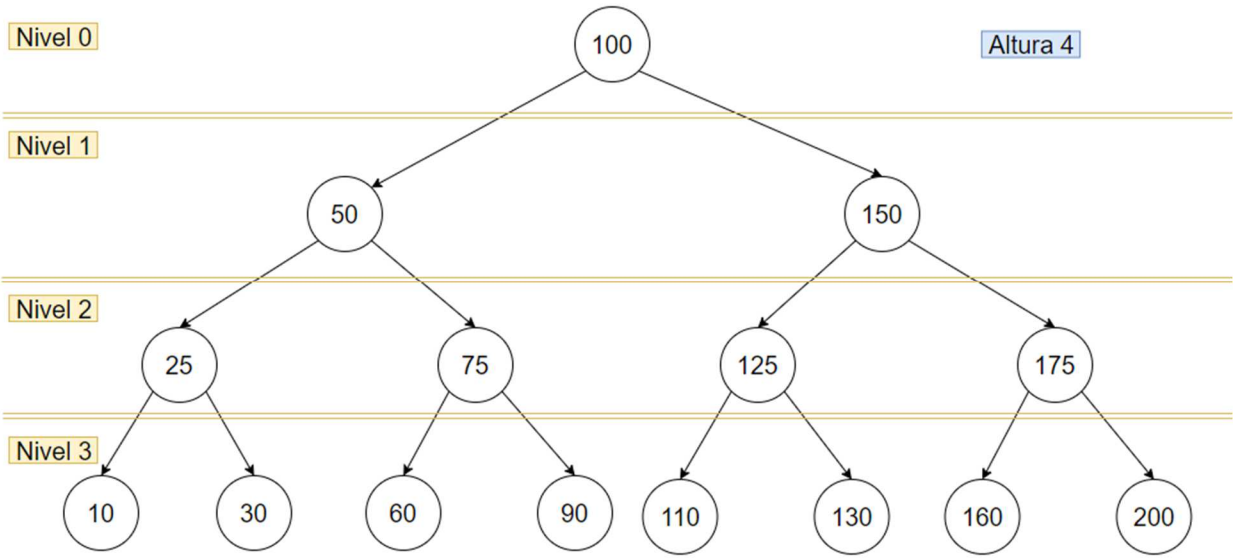


Como se ve en el gráfico, en el árbol se pueden identificar los niveles, que indican la cantidad de saltos que se tiene que dar para llegar a un nodo específico. La altura del árbol es la cantidad de niveles. Estos datos son de gran utilidad, porque nos indican que tan costoso puede ser encontrar un nodo en un árbol binario determinado.

De este tipo de árboles se desprende al menos 2 grandes categorías utilizadas en informática. Los árboles binarios de búsqueda y los árboles de expresión.

Árboles binarios de búsqueda

Son árboles binarios que están ordenados de manera tal que se cumple para todos los nodos del árbol que el valor del hijo izquierdo es menor que el de la raíz y el valor del hijo derecho es mayor. Son las estructuras más comúnmente utilizadas para índices en base de datos, porque, por su estructura (y como su nombre lo indica) la búsqueda es la operación que está más optimizada.

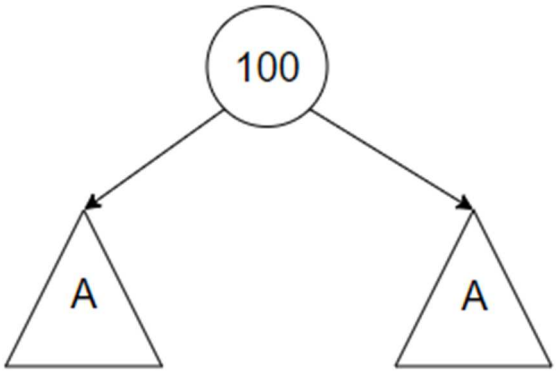


Entonces, para buscar en un árbol binario de búsqueda, simplemente nos vamos a la izquierda o a la derecha según si el dato que estamos buscando es menor o mayor que la raíz. Y es aquí donde toma importancia el concepto de la altura, porque es la que nos va a indicar el máximo de saltos que hacen falta para encontrar un nodo en el árbol.

Tipos de recorrida

Los árboles tienen distintas formas de ser recorrido según lo que queramos hacer. Lo primero que tenemos que tener en cuenta es que como los árboles son estructuras recursivas, para recorrerlos usar recursividad, va a ser lo que resulte más sencillo de programar.

- Recorrida en orden, Es la forma más común de mostrar un árbol binario de búsqueda, que naturalmente en este tipo de árboles, va a mostrar todos los nodos ordenados. Teniendo en consideración de que cada hijo de un nodo es un árbol en sí mismo, y suponiendo que queremos *mostrar* el árbol en pantalla, primero tenemos que *mostrar en orden* el árbol que desprende del hijo izquierdo, luego la raíz, y luego *mostrar en orden* el árbol que deprende del hijo derecho.



En pseudocódigo:

```
RecorrerEnOrden (arbol)
    si (arbol no es vacío)
    {
        RecorrerEnOrden(arbol->izq)
        MostrarRaiz (arbol)
        RecorrerEnOrden(arbol->der)
    }
```

- Recorrida en Preorden: Esta recorrida muestra primeros los nodos raíces y luego los subárboles izquierdo y derecho. Es de especial utilidad cuando necesitemos desarmar el árbol y volver a construir exactamente a como estaba en un principio.

En pseudocódigo:

```
RecorrerPreOrden (arbol)
```

```
si (arbol no es vacío)
{
    MostrarRaiz (arbol)
    RecorrerPreOrden(arbol->izq)
    RecorrerPreOrden(arbol->der)
}
```

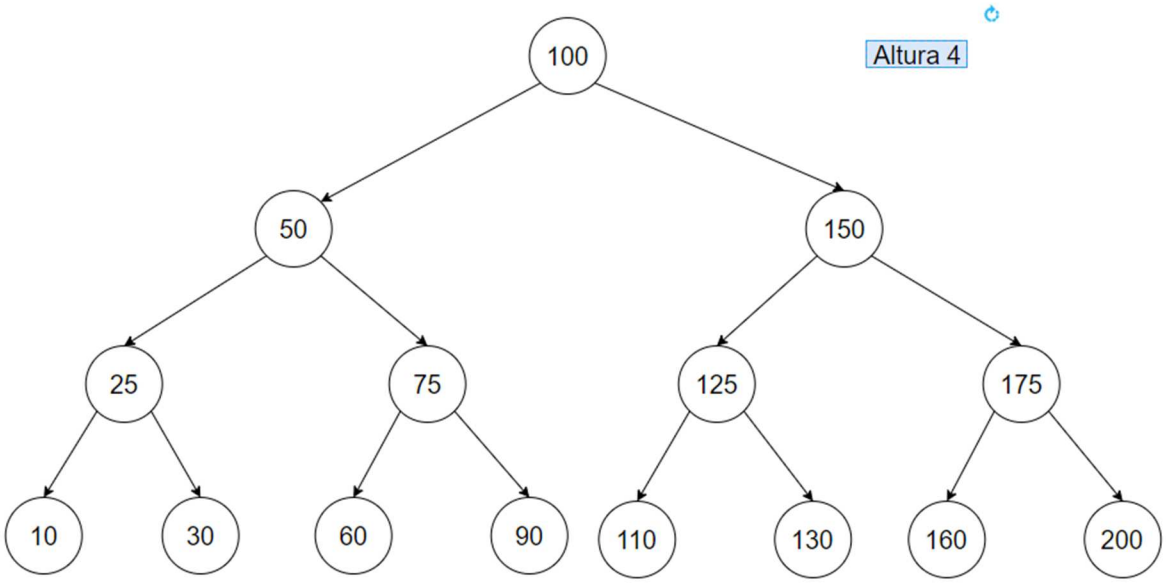
- Recorrida en Posorden (o Polaca Inversa): Esta recorrida accede primero a los subárboles izquierdo y derecho, y por último a la raíz. Es muy útil, también, para reconstruir árboles.
  En pseudocódigo:

```
RecorrerPosOrden (arbol)
    si (arbol no es vacío)
    {
        RecorrerPosOrden(arbol->izq)
        RecorrerPosOrden(arbol->der)
        MostrarRaiz (arbol)
    }
```

Balanceo de árboles

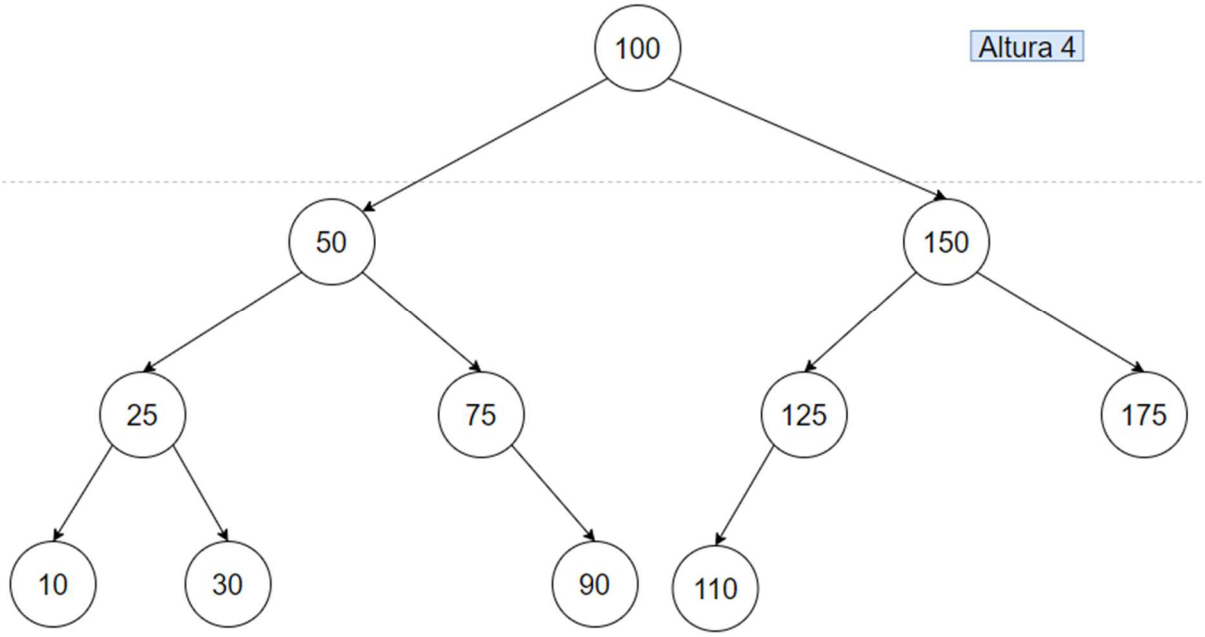
Para cumplir con el objetivo de ser eficientes en las búsquedas, los árboles binarios de búsqueda deben estar lo más balanceados posible. Pero ¿Qué significa que un árbol esté “lo más balanceado posible”? Significa que tiene aproximadamente la misma cantidad de nodos o la misma altura entre los subárboles que desprenden a la izquierda y a la derecha del mismo. Para definir con más exactitud este tema del balanceo, podemos decir que existen 3 tipos de árboles:

- Completos: Son los árboles que tienen exactamente la cantidad de nodos correspondiente su altura. Entonces un árbol de altura 2 debe tener 3 nodos, uno de altura 3, 7 nodos, altura 4, 15 nodos, etc. Para este tipo de árboles se da siempre que: cn = 2^h-1 donde cn es la cantidad de nodos y h es la altura.



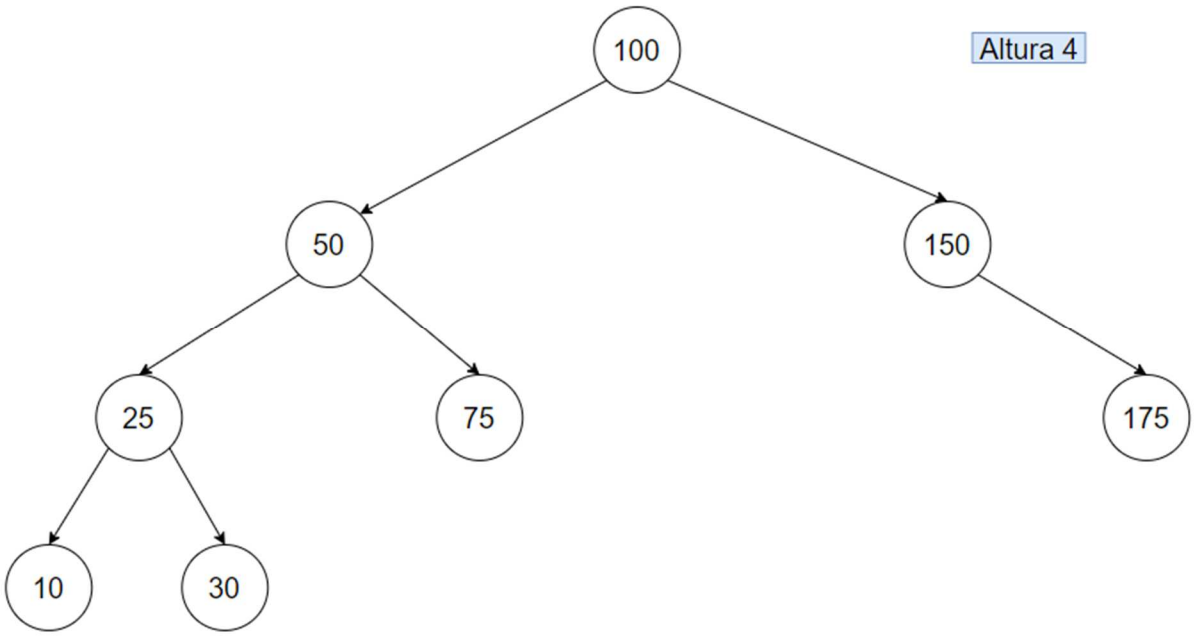
  Estos tipos de árboles tienen la ventaja que son los más eficientes en las búsquedas para la cantidad de nodos que tienen. Pero, la principal desventaja es que sí o sí tienen que tener la cantidad de nodos correspondiente a su altura.
- Balanceados: Son árboles casi completos. Deben estar completos hasta un nivel anterior al último para cumplir con la condición de balanceo. Son los árboles que

permiten cualquier cantidad de nodos más eficientes en términos de búsqueda.



- Balanceados AVL: Si bien los árboles balanceados son los más eficientes en términos de búsqueda, obtener un árbol balanceado a partir de uno que no lo está, tiene una complejidad de cómputo muy alta. Entonces los matemáticos rusos **A**delson-**V**elskii y **L**andis descubrieron un tipo de árbol mucho más fácil de obtener que conserva ciertas propiedades en la eficiencia de la búsqueda. Los árboles AVL cumplen con la condición de que, para todos los nodos del árbol, se cumple que la diferencia entre las alturas de los subárboles izquierdo y derecho nunca es

mayor a 1.



Revisando el código

Vamos a revisar la implementación de un TDA de árbol.

Primero, los nodos deben contener una referencia a la información y almacenar el tamaño de la misma. Y luego hacer referencia también a los hijos izquierdo y derecho. Por último, nuestro árbol va a ser el puntero a la raíz del mismo,

```
19
20  typedef struct sNodoArbol
21  {
22      void *info;
23      unsigned tamInfo;
24      struct sNodoArbol *izq,
25                        *der;
26  }tNodoArbol;
27
28  typedef tNodoArbol *tArbolBinBusq;
```

Un árbol vacío, es aquel en que el puntero al nodo raíz es nulo.

Insertar en el árbol

En orden, obviamente. Al ser un árbol binario de búsqueda, la única operación de inserción directa es la inserción en orden. Que puede ser lograda de manera iterativa:

```
43  #define reservarMemoriaNodo( X , Y , Z , W ) (                          \
44                ( ( X ) = (typeof( X ))malloc( Y ) ) == NULL ||                    \
45                ( ( Z ) = malloc( W ) ) == NULL   ?                        \
46                      free( X ), 0 : 1  )
47
48  int insertarArbolBinBusq(tArbolBinBusq *p, const void *d, unsigned tam,
49                           int (*cmp)(const void *, const void *))
50  {
51      tNodoArbol *nue;
52      int         rc;
53
54      while(*p)
55      {
56          if((rc = cmp(d, (*p)->info)) < 0)
57              p = &(*p)->izq;
58          else if (rc > 0)
59              p = &(*p)->der;
60          else
61              return CLA_DUP;
62      }
63      if(!reservarMemoriaNodo(nue, sizeof(tNodoArbol), nue->info, tam))
64          return SIN_MEM;
65      nue->tamInfo = tam;
66      memcpy(nue->info, d, tam);
67      nue->der = nue->izq = NULL;
68      *p = nue;
69      return TODO_BIEN;
70  }
```

Las inserciones en el árbol, siempre se dan en algún puntero vacío. Siempre va a haber un lugar para insertar un nuevo valor entre 2 valores. Esto hace especialmente sencilla la operación de inserción puesto que nos tenemos que mover al lugar exacto donde va a ser insertado el nodo, sin tener que recorrer el árbol entero, y siempre vamos a insertar sobre un puntero vacío.

Por lo mencionado anteriormente recorremos hasta encontrarnos con un “Árbol” vacío (ln. 54), y si la información que queremos insertar es menor que la que encontramos en el árbol (ln 56), nos movemos a la izquierda (ln 57), si es mayor, a la derecha, y si ya existe, salimos indicando que no se pudo insertar por clave duplicada.

Luego vienen las operaciones de crear el nodo, y cargarlo. En este caso se hizo uso de una macro para hacer la reserva de memoria, pero las operaciones son difieren nada de lo que hicimos anteriormente para Lista, pila o cola. Se reserva memoria para el nodo nuevo (el operador typeof devuelve el tipo de dato de una variable), y se evalúa si se pudo reservar la memoria (ln 44), y se hace lo mismo con la información (ln 45). Si el primero falla, la evaluación da verdadera y, por la condición OR, el segundo nunca se ejecuta. Si alguno de los 2 fallan, se devuelve falso (0) como resultado y se libera la memoria que fue reservada en el primer malloc (si fue esta la reserva de memoria que falló, el puntero pasado a free va a ser nulo, y free no va a hacer nada)(ln 46). Si ambos son correctos, se devuelve verdadero indicando que la operación fue exitosa (ln 46).

En definitiva, esto nos deja con una expresión donde podemos mandar a reservar la memoria, y evaluar si la operación fue exitosa o no (ln 63).

Se carga la información (ln 65) con su tamaño correspondiente (ln 66), y se asignan los punteros a izquierda y derecha a nulos indicando que este nuevo nodo no tiene hijos (ln 67).

Por último, se “engancha” el nuevo nodo al árbol asignándolo al puntero que encontramos como el lugar correcto para este nuevo nodo.

La operación de búsqueda para la inserción puede hacerse de manera recursiva, aunque para este caso no es la más eficiente.

```
73  int insertarRecArbolBinBusq(tArbolBinBusq *p, const void *d, unsigned tam,
74                              int (*cmp)(const void *, const void *))
75  {
76      tNodoArbol *nue;
77      int         rc;
78
79      if(*p)
80      {
81          if((rc = cmp(d, (*p)->info)) < 0)
82              return insertarRecArbolBinBusq(&(*p)->izq, d, tam, cmp);
83          if (rc > 0)
84              return insertarRecArbolBinBusq(&(*p)->der, d, tam, cmp);
85          return CLA_DUP;
86      }
87      if(!reservarMemoriaNodo(nue, sizeof(tNodoArbol), nue->info, tam))
88          return SIN_MEM;
89      nue->tamInfo = tam;
90      memcpy(nue->info, d, tam);
91      nue->der = NULL;
92      nue->izq = NULL;
93      *p = nue;
94      return TODO_BIEN;
95  }
```

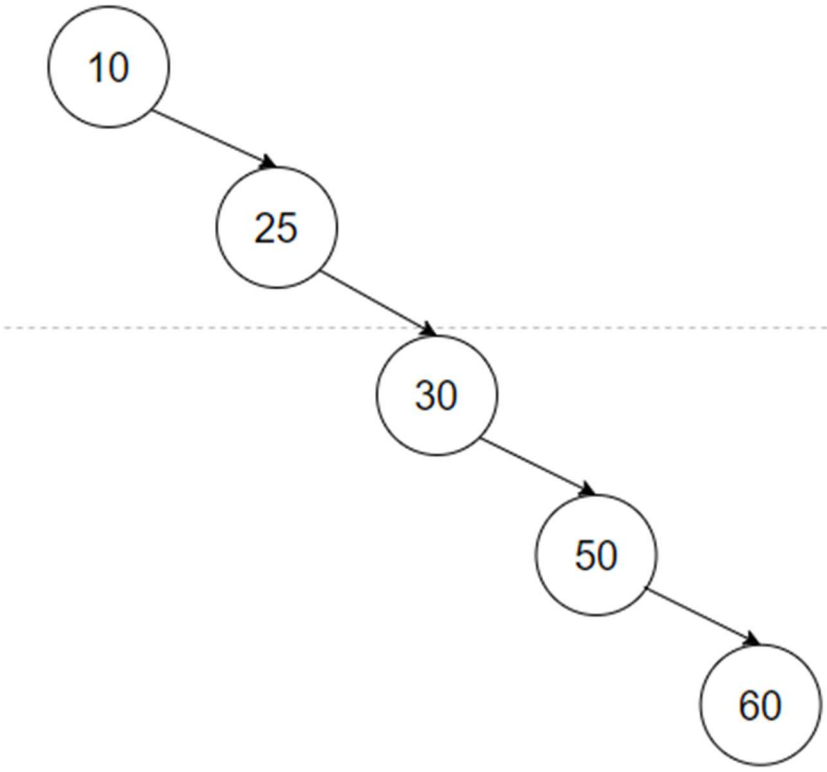
Teniendo en cuenta que esta función tiene 2 partes principales, la de búsqueda y la de inserción del nodo, nos damos cuenta de que, si se ingresó a la sección de búsqueda de la función, no se va a ingresar a la sección de inserción en ese llamado. Esto se da porque dentro del if de búsqueda, todos los caminos de código tienen un return.

Entonces, si el dato a insertar es menor al del nodo del árbol, “insertaremos” el árbol que está a la izquierda (ln 82) y devolveremos el resultado de esa operación. Si es mayor “insertaremos” el dato en el árbol derecho, y si son iguales, no insertaremos y saldremos con un error de clave duplicada.

Cargar el árbol desde un set de datos ordenado

El problema con cargar un set de datos ordenado en un árbol binario de búsqueda es que si leemos los datos secuencialmente y los vamos ingresando en el árbol

terminaríamos con un árbol completamente desbalanceado.



Entonces, para solucionar este problema debemos ir accediendo al elemento del medio del set de datos y repetir la operación para las mitades izquierda y derecha que nos quedaron al partir los datos en 2. El nodo raíz será el elemento del medio, la mitad izquierda, o sea los elementos menores al del medio, serán insertados, naturalmente, a la izquierda del nodo raíz y los mayores a la derecha.

UNLaM
Dto. Ingeniería e Investigaciones Tecnológicas

```
251 int cargarDesdeDatosOrdenadosRec(tArbolBinBusq * p, void * ds,
252                                  unsigned (*leer)(void **, void *, unsigned, void *params),
253                                  int li, int ls, void * params)
254 {
255     int m = (li+ls)/2,
256         r;
257     if (li>ls)
258         return TODO_BIEN;
259
260     (*p) = (tNodoArbol*) malloc(sizeof(tNodoArbol));
261     if (!*p || !((*p)->tamInfo = leer(&(*p)->info, ds, m, params)))
262     {
263         free(*p);
264         return SIN_MEM;
265     }
266
267     (*p)->izq = (*p)->der = NULL;
268
269     if((r=cargarDesdeDatosOrdenadosRec(&(*p)->izq, ds, leer, li, m-1, params))!=TODO_BIEN)
270         return r;
271     return cargarDesdeDatosOrdenadosRec(&(*p)->der, ds, leer, m+1, ls, params);
272 }
```

Teniendo lo anterior en cuenta, vamos a recibir en nuestra función el set de datos (que puede ser un vector o un archivo o alguna otra estructura que se nos ocurra), un límite inferior que nos va a indicar donde comienza el set de datos, y un límite superior que indica donde termina el set de datos.

Primero verificamos que no hayamos llegado al caso de que no haya datos a insertar, sería nuestra condición de corte de recursividad. Cuando lleguemos a una situación en la que el límite superior es menor que el inferior, quiere decir que esa parte del set ya no contiene más datos y podemos decir que en esa rama del árbol no se van a ingresar más valores. Por ende, salimos con éxito (ln 257).

Luego calculamos el elemento del medio a partir de los li y ls (ln 255) usando la operación matemática de la media.

Creamos un nuevo nodo (ln 260) y leemos el dato a partir de la función que nos enviaron por parámetro (ln 261). En ese momento evaluamos si la operación fue exitosa o no, y si no salimos con un error. La función leer se encarga de acceder a la información en el set de datos, reservar la memoria para almacenarla en el nodo, copiarla al espacio reservado, y además, devuelve el tamaño de la información para ser asignado al tamaño almacenado en el nodo.

UNLaM
Dto. Ingeniería e Investigaciones Tecnológicas

Por último, enviamos a insertar los elementos menores del set de datos a la izquierda del árbol y los mayores a la derecha (ln 269 y 271). En caso de error en la primera llamada, salimos retornando el error. Los elementos menores son los que van a estar comprendidos entre el límite inferior del set de datos y la media menos uno, y los mayores entre la media más uno y el límite superior.

Después contaremos con las funciones específicas para cada set de datos. En la Biblioteca incluimos una opción para archivos binarios:

```
286 int cargarArchivoBinOrdenadoArbolBinBusq(tArbolBinBusq * p, const char * path,
287                                          unsigned tamInfo)
288 {
289     int cantReg,
290         r;
291     FILE * pf;
292     if(*p)
293         return SIN_INICIALIZAR;
294     if(!(pf= fopen(path, "rb")))
295         return ERROR_ARCH;
296     fseek(pf, 0L, SEEK_END);
297     cantReg = ftell(pf)/tamInfo;
298     r = cargarDesdeDatosOrdenadosRec(p, pf, leerDesdeArchivoBin, 0, cantReg-1, &tamInfo);
299     fclose(pf);
300     return r;
301 }
```

Con su operación de lectura:

```
240 unsigned leerDesdeArchivoBin(void ** d, void * pf, unsigned pos, void * params)
241 {
242     unsigned tam = *((int*)params);
243     *d = malloc(tam);
244     if(!*d)
245         return 0;
246     fseek((FILE*)pf, pos*tam, SEEK_SET);
247     return fread(*d, tam, 1, (FILE*)pf);
248 }
249
```

Y otra función que sirve como punto de acceso para cargar el árbol desde el set de datos ordenados que el usuario defina. Con parámetros un poco más amigables.

```
304 int cargarDesdeDatosOrdenadosArbolBinBusq(tArbolBinBusq * p, void * ds, unsigned cantReg,
305                                          unsigned (*leer)(void **, void *, unsigned, void *params),
306                                          void * params)
307 {
308     if(*p || !ds)
309         return 0;
310     return cargarDesdeDatosOrdenadosRec(p, ds, leer, 0, cantReg-1, params);
311 }
```

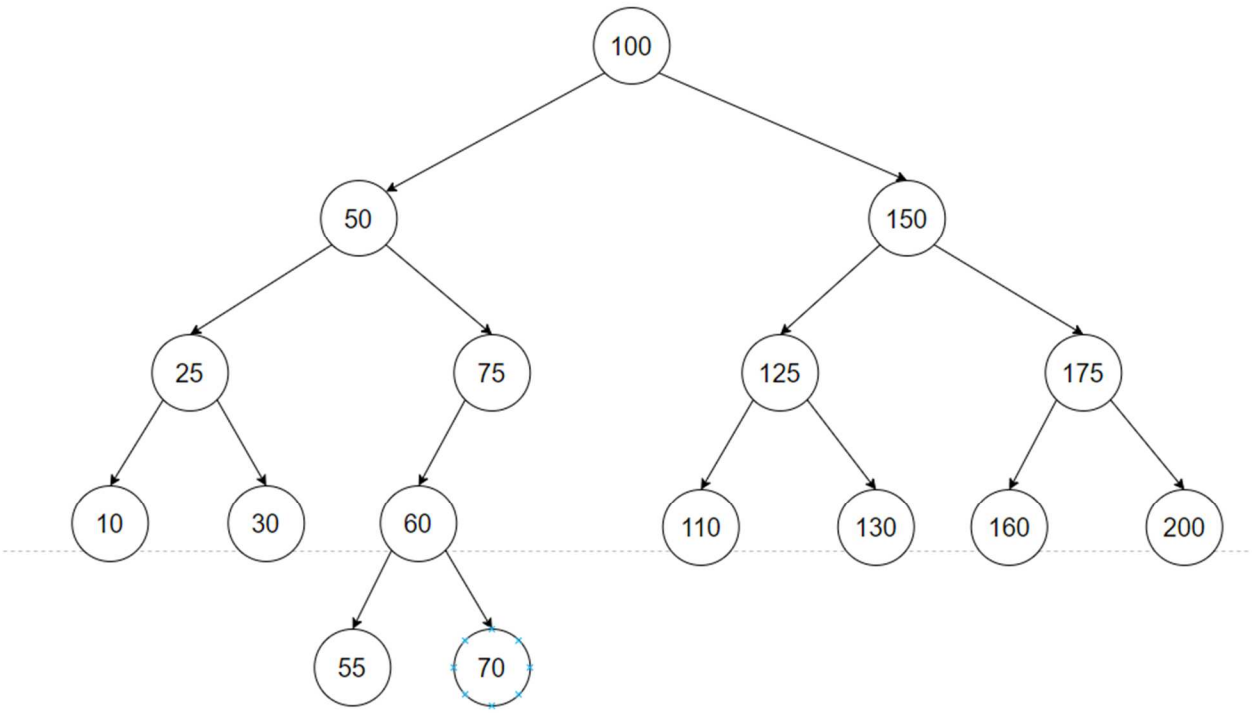
UNLaM
Dto. Ingeniería e Investigaciones Tecnológicas

Eliminar un elemento del árbol

Para eliminar un elemento del árbol, primero debemos buscarlo. Una vez obtenido el nodo, la operación sería como eliminar la raíz del árbol del cual ese elemento es la raíz.

```
196  int eliminarRaizArbolBinBusq(tArbolBinBusq * p)
197  {
198      tNodoArbol ** remp,
199                  * elim;
200      if(!*p)
201          return 0; ///ARBOL_VACIO
202
203      free((*p)->info);
204      if (!(*p)->izq && !(*p)->der)
205      {
206          free(*p);
207          *p = NULL;
208          return 1; ///OK
209      }
210
211      remp = alturaArbolBin(&(*p)->izq)>alturaArbolBin(&(*p)->der)?
212          mayorNodoArbolBinBusq(&(*p)->izq):
213          menorNodoArbolBinBusq(&(*p)->der);
214
215      elim = *remp;
216      (*p)->info = elim->info;
217      (*p)->tamInfo = elim->tamInfo;
218
219      *remp = elim->izq ? elim->izq : elim->der;
220
221      free(elim);
222
223      return 1; ///OK
224  }
```

Por ejemplo, si queremos eliminar la raíz de este árbol:



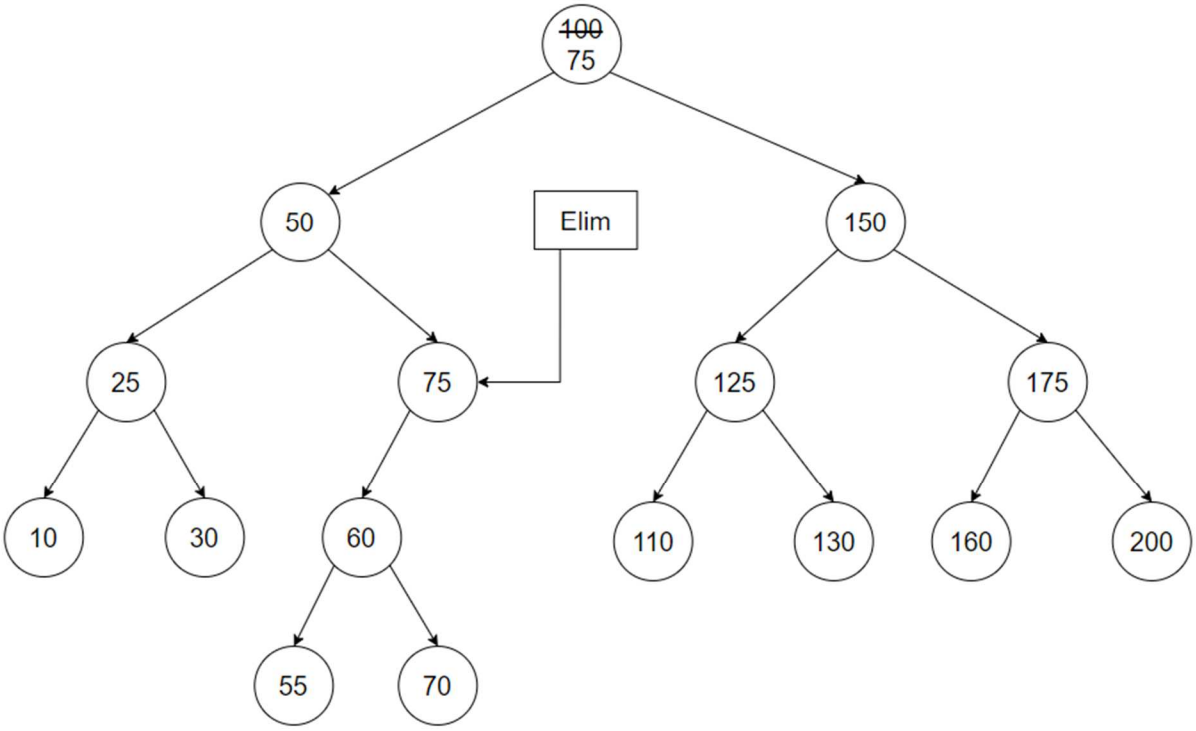
Lo primero que debemos hacer liberar la información que estamos seguros de que va a ser eliminada (ln 203). Luego chequearemos si no tiene hijos. Si no los tiene, directamente eliminamos el nodo (ln 204 a 208), y se terminó la función.

Pero si tiene hijos, debemos encontrarle un reemplazante al nodo para que sea la raíz del árbol. Pero debemos hacerlo con un criterio, para intentar desbalancear lo menos posible al árbol al eliminar los nodos. Hay siempre 2 candidatos a remplazar la raíz y que el árbol siga teniendo los mismos nodos a la izquierda y a la derecha. Y son los que, mirando los nodos en orden, están al lado del nodo raíz que queremos eliminar … 75 **100** 110 … Dicho de otra manera el mayor de los menores (75) o el menor de los mayores (110). O también el de más a la derecha de los de la izquierda (75), o el de más a la izquierda de la derecha (110). Si bien el más fácil de eliminar de ese árbol es el 110,

porque no tiene hijos, sería mejor elegir el 75, porque al eliminarlo vamos a estar reduciendo el desbalanceo que está afectando a ese árbol.

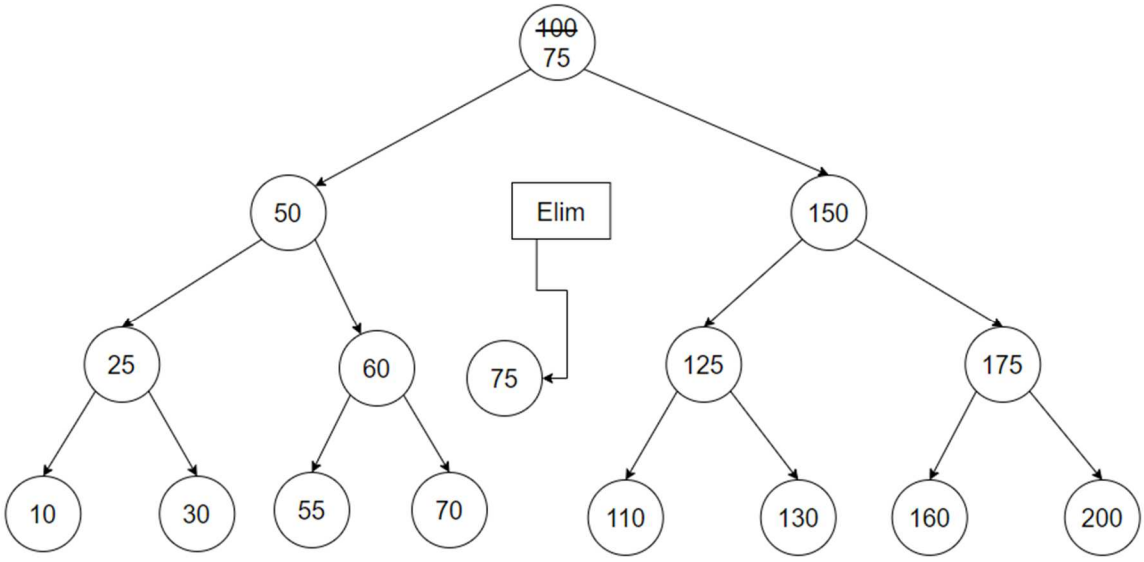
Es por eso que vamos a seleccionar el candidato a remplazo que corresponda a la rama más alta entre la izquierda y la derecha (ln 211). Y, en caso de que sea la rama de la izquierda seleccionaremos el mayor de esa rama (ln 212), y si es la derecha seleccionaremos el menor (ln 213).

Una vez seleccionado el nodo remplazamos la información de la raíz (junto con el tamaño que le corresponde) (ln 216 y 217).



Reasignamos los hijos del nodo a eliminar su padre (ln 219). Aquí cabe destacar que por el proceso de selección realizado el nodo que estamos queriendo eliminar tiene solo un hijo. El izquierdo si era el mayor de los menores o el derecho si era el menor de los

mayores. Para el caso del 75 tiene solo el hijo izquierdo que es reasignado a ser hijo del 50.



Una vez separado el nodo que contenía el 75, es el momento de eliminarlo (ln 221).

```c
/** ************************************************************************
 **     main.c      prueba de primitivas del  TDA ÁRBOL  binario de búsqueda
 ** ************************************************************************ **/

#include "main.h"

int main()
{
    tArbolBinBusq ar;
    int vec[] = {10,25,50,60,75,100,110,125,150,175,200};
    //int vec[] = {25,50,75,100,125,150,175};
    int dato;

    crearArbolBinBusq(&ar);

    cargarDesdeDatosOrdenadosArbolBinBusq(&ar, vec, sizeof(vec)/sizeof(int),
                                          leerDesdeVectorEnteros,NULL);

    ///eliminarRaizArbolBinBusq(&ar);

    recorrerEnOrdenArbolBinBusq(&ar,NULL,imprimir);
    printf("\n\n");
    recorrerPreOrdenArbolBinBusq(&ar,NULL,imprimir);
    printf("\n\n");
    recorrerPosOrdenArbolBinBusq(&ar,NULL,imprimir);
    printf("\n\n");
    recorrerEnOrdenInversoArbolBinBusq(&ar,NULL,imprimirConForma);

    printf("\n\n");
    if (esCompletoArbolBin(&ar))
        printf("\nCompleto");
    if (esBalanceadoArbolBin(&ar))
        printf("\nBalanceado");
    if (esAVLArbolBin(&ar))
        printf("\nAVL");

    printf("\n\n");
    if (esCompleto2ArbolBin(&ar))
        printf("\n2 Completo");
    if (esBalanceado2ArbolBin(&ar))
        printf("\n2 Balanceado");
    if (esAVL2ArbolBin(&ar))
        printf("\n2 AVL");

    mayorElemNoClaveArbolBinBusq(&ar, &dato, sizeof(dato),cmp_ent);
    printf("\n\nMayor No Clave: %d", dato);

    menorElemNoClaveArbolBinBusq(&ar, &dato, sizeof(dato),cmp_ent);
    printf("\n\nMenor No Clave: %d", dato);

    return 0;
}

void imprimirConForma(void * info, unsigned tam, unsigned n, void * params)
{
    int * i = (int *) info;
    printf("%*s-%3d-\n",n*3,"", *i);
}

void imprimir(void * info, unsigned tam, unsigned n, void * params)
{
    int * i = (int *) info;
    printf("-%d-", *i);
}

unsigned leerDesdeVectorEnteros(void ** d, void * vec, unsigned pos, void * params)
```

```
67  {
68      *d = malloc(sizeof(int));
69      if(!*d)
70          return 0;
71      memcpy(*d, vec+(sizeof(int)*pos), sizeof(int));
72      return sizeof(int);
73  }
74
75  int cmp_ent(const void *v1, const void *v2)
76  {
77      return *((int*)v1)-*((int*)v2);
78  }
79
80  /** *************************************************************************
81   **  FIN - main.c      prueba de primitivas del  TDA ÁRBOL  binario de búsqueda
82   ** ************************************************************************* **/
83
84
85   /** *************************************************************************
86   **     main.h      prueba de primitivas del  TDA ÁRBOL  binario de búsqueda
87   ** ************************************************************************* **/
88
89  #ifndef ARBOL_H_
90  #define ARBOL_H_
91
92  #include <stdio.h>
93  #include <stdlib.h>
94  #include <string.h>
95
96  #include "../Biblioteca/arbol_bin_busq.h"
97
98  int cmp_ent(const void *v1, const void *v2);
99
100 void imprimirConForma(void *info, unsigned tam, unsigned n, void *params);
101 void imprimir(void *info, unsigned tam, unsigned n, void *params);
102
103 unsigned leerDesdeVectorEnteros(void **d, void *vec, unsigned pos, void *params);
104
105 #endif // ARBOL_H_
106
107 /** *************************************************************************
108  **  FIN - main.h      prueba de primitivas del  TDA ÁRBOL  binario de búsqueda
109  ** ************************************************************************* **/
110
111
112  /** *************************************************************************
113  **     arbol_bin_busq.c    definición primitivas  TDA ÁRBOL  bin. de búsqueda
114  ** ************************************************************************* */
115
116
117 #include"arbol_bin_busq.h"
118
119 #define MINIMO(X,Y) ((X)<(Y)?(X):(Y))
120
121 tNodoArbol **mayorNodoArbolBinBusq(const tArbolBinBusq *p);
122
123 tNodoArbol **menorNodoArbolBinBusq(const tArbolBinBusq *p);
124
125 tNodoArbol **mayorRecNodoArbolBinBusq(const tArbolBinBusq *p);
126
127 tNodoArbol **menorRecNodoArbolBinBusq(const tArbolBinBusq *p);
128
129 tNodoArbol **buscarNodoArbolBinBusq(const tArbolBinBusq *p, const void *d,
130                                     int (*cmp)(const void *, const void *));
131
132 tNodoArbol **buscarRecNodoArbolBinBusq(const tArbolBinBusq *p, const void *d,
```

```
133                                     int (*cmp)(const void *, const void *));
134
135  int esCompletoHastaNivelArbolBin(const tArbolBinBusq *p, int n);
136
137  int esAVL2CalculoArbolBin(const tArbolBinBusq *p);
138
139  const tArbolBinBusq * mayorNodoNoClaveArbolBinBusq(const tArbolBinBusq *p, const
tArbolBinBusq *mayor,
140                                     int (*cmp)(const void *, const void *));
141
142  const tArbolBinBusq * menorNodoNoClaveArbolBinBusq(const tArbolBinBusq *p, const
tArbolBinBusq *menor,
143                                     int (*cmp)(const void *, const void *));
144
145  const tArbolBinBusq * buscarNodoNoClaveArbolBinBusq(const tArbolBinBusq *p, const
void *d,
146                                     int (*cmp)(const void *, const void *));
147
148  void crearArbolBinBusq(tArbolBinBusq *p)
149  {
150      *p = NULL;
151  }
152
153
154  /// Reservar Memoria Nodo
155  #define reservarMemoriaNodo( X , Y , Z , W ) (                              \
156                ( ( X ) = (typeof( X ))malloc( Y ) ) == NULL ||
    \
157                ( ( Z ) = malloc( W ) ) == NULL   ?                            \
158                     free( X ), 0 : 1  )
159
160  int insertarArbolBinBusq(tArbolBinBusq *p, const void *d, unsigned tam,
161                           int (*cmp)(const void *, const void *))
162  {
163      tNodoArbol *nue;
164      int         rc;
165
166      while(*p)
167      {
168          if((rc = cmp(d, (*p)->info)) < 0)
169              p = &(*p)->izq;
170          else if (rc > 0)
171              p = &(*p)->der;
172          else
173              return CLA_DUP;
174      }
175      if(!reservarMemoriaNodo(nue, sizeof(tNodoArbol), nue->info, tam))
176          return SIN_MEM;
177      nue->tamInfo = tam;
178      memcpy(nue->info, d, tam);
179      nue->der = nue->izq = NULL;
180      *p = nue;
181      return TODO_BIEN;
182  }
183
184
185  int insertarRecArbolBinBusq(tArbolBinBusq *p, const void *d, unsigned tam,
186                              int (*cmp)(const void *, const void *))
187  {
188      tNodoArbol *nue;
189      int         rc;
190
191      if(*p)
192      {
193          if((rc = cmp(d, (*p)->info)) < 0)
194              return insertarRecArbolBinBusq(&(*p)->izq, d, tam, cmp);
```

```
195         if (rc > 0)
196             return insertarRecArbolBinBusq(&(*p)->der, d, tam, cmp);
197         return CLA_DUP;
198     }
199     if(!reservarMemoriaNodo(nue, sizeof(tNodoArbol), nue->info, tam))
200         return SIN_MEM;
201     nue->tamInfo = tam;
202     memcpy(nue->info, d, tam);
203     nue->der = NULL;
204     nue->izq = NULL;
205     *p = nue;
206     return TODO_BIEN;
207 }
208
209 void recorrerEnOrdenRecArbolBinBusq(const tArbolBinBusq * p, unsigned n, void *
params,
210                                     void (*accion)(void *, unsigned, unsigned, void
*))
211 {
212     if(!*p)
213         return;
214     recorrerEnOrdenRecArbolBinBusq(&(*p)->izq, n+1, params, accion);
215     accion((*p)->info, (*p)->tamInfo, n, params);
216     recorrerEnOrdenRecArbolBinBusq(&(*p)->der, n+1, params, accion);
217 }
218
219 void recorrerEnOrdenArbolBinBusq(const tArbolBinBusq * p, void * params,
220                                  void (*accion)(void *, unsigned, unsigned, void *))
221 {
222     recorrerEnOrdenRecArbolBinBusq(p, 0, params, accion);
223 }
224
225 void recorrerEnOrdenInversoRecArbolBinBusq(const tArbolBinBusq * p, unsigned n, void
* params,
226                                            void (*accion)(void *, unsigned, unsigned
, void *))
227 {
228     if(!*p)
229         return;
230     recorrerEnOrdenInversoRecArbolBinBusq(&(*p)->der, n+1, params, accion);
231     accion((*p)->info, (*p)->tamInfo, n, params);
232     recorrerEnOrdenInversoRecArbolBinBusq(&(*p)->izq, n+1, params, accion);
233 }
234
235 void recorrerEnOrdenInversoArbolBinBusq(const tArbolBinBusq * p, void * params,
236                                         void (*accion)(void *, unsigned, unsigned,
void *))
237 {
238    recorrerEnOrdenInversoRecArbolBinBusq(p, 0, params, accion);
239 }
240
241 void recorrerPreOrdenRecArbolBinBusq(const tArbolBinBusq * p, unsigned n, void *
params,
242                                      void (*accion)(void *, unsigned, unsigned, void
*))
243 {
244     if(!*p)
245         return;
246     accion((*p)->info, (*p)->tamInfo, n, params);
247     recorrerPreOrdenRecArbolBinBusq(&(*p)->izq, n+1, params, accion);
248     recorrerPreOrdenRecArbolBinBusq(&(*p)->der, n+1, params, accion);
249 }
250
251 void recorrerPreOrdenArbolBinBusq(const tArbolBinBusq * p, void * params,
252                                   void (*accion)(void *, unsigned, unsigned, void
*))
```

```
253  {
254      recorrerPreOrdenRecArbolBinBusq(p, 0, params, accion);
255  }
256
257
258  void recorrerPosOrdenRecArbolBinBusq(const tArbolBinBusq * p, unsigned n,
259                                       void * params, void (*accion)(void *, unsigned,
unsigned, void *))
260  {
261      if(!*p)
262          return;
263      recorrerPosOrdenRecArbolBinBusq(&(*p)->izq, n+1, params, accion);
264      recorrerPosOrdenRecArbolBinBusq(&(*p)->der, n+1, params, accion);
265      accion((*p)->info, (*p)->tamInfo, n, params);
266  }
267
268  void recorrerPosOrdenArbolBinBusq(const tArbolBinBusq * p, void * params,
269                                    void (*accion)(void *, unsigned, unsigned, void
*))
270  {
271      recorrerPosOrdenRecArbolBinBusq(p, 0, params, accion);
272  }
273
274  void recorrerEnOrdenSimpleArbolBinBusq(const tArbolBinBusq * p,
275                                         void * params, void (*accion)(void *,
unsigned, void *))
276  {
277      if(!*p)
278          return;
279      recorrerEnOrdenSimpleArbolBinBusq(&(*p)->izq, params, accion);
280      accion((*p)->info, (*p)->tamInfo, params);
281      recorrerEnOrdenSimpleArbolBinBusq(&(*p)->der, params, accion);
282  }
283
284
285  void recorrerPreOrdenSimpleArbolBinBusq(const tArbolBinBusq * p,
286                                          void * params, void (*accion)(void *,
unsigned, void *))
287  {
288      if(!*p)
289          return;
290      accion((*p)->info, (*p)->tamInfo, params);
291      recorrerPreOrdenSimpleArbolBinBusq(&(*p)->izq, params, accion);
292      recorrerPreOrdenSimpleArbolBinBusq(&(*p)->der, params, accion);
293  }
294
295
296  void recorrerPosOrdenSimpleArbolBinBusq(const tArbolBinBusq * p,
297                                          void * params, void (*accion)(void *,
unsigned, void *))
298  {
299      if(!*p)
300          return;
301      recorrerPosOrdenSimpleArbolBinBusq(&(*p)->izq, params, accion);
302      recorrerPosOrdenSimpleArbolBinBusq(&(*p)->der, params, accion);
303      accion((*p)->info, (*p)->tamInfo, params);
304  }
305
306
307  int eliminarRaizArbolBinBusq(tArbolBinBusq * p)
308  {
309      tNodoArbol ** remp,
310                  * elim;
311      if(!*p)
312          return 0; ///ARBOL_VACIO
313
```

```
314       free((*p)->info);
315       if (!(*p)->izq && !(*p)->der)
316       {
317           free(*p);
318           *p = NULL;
319           return 1; ///OK
320       }
321
322       remp = alturaArbolBin(&(*p)->izq)>alturaArbolBin(&(*p)->der)?
323           mayorNodoArbolBinBusq(&(*p)->izq):
324           menorNodoArbolBinBusq(&(*p)->der);
325
326       elim = *remp;
327       (*p)->info = elim->info;
328       (*p)->tamInfo = elim->tamInfo;
329
330       *remp = elim->izq ? elim->izq : elim->der;
331
332       free(elim);
333
334       return 1; ///OK
335   }
336
337
338
339   int eliminarElemArbolBinBusq(tArbolBinBusq * p, void * d, unsigned tam,
340                                int (*cmp)(const void*, const void *))
341   {
342       if(!(p = buscarNodoArbolBinBusq(p,d,cmp)))
343           return 0; ///NO_EXISTE
344       memcpy(d, (*p)->info, MINIMO(tam, (*p)->tamInfo));
345       return eliminarRaizArbolBinBusq(p);
346   }
347
348   int buscarElemArbolBinBusq(const tArbolBinBusq *p, void *d, unsigned tam,
349                                int (*cmp)(const void *, const void *))
350   {
351       if(!(p = buscarNodoArbolBinBusq(p,d,cmp)))
352           return 0; ///NO_EXISTE
353       memcpy(d, (*p)->info, MINIMO(tam, (*p)->tamInfo));
354       return 1;
355   }
356
357
358   unsigned leerDesdeArchivoBin(void ** d, void * pf, unsigned pos, void * params)
359   {
360       unsigned tam = *((int*)params);
361       *d = malloc(tam);
362       if(!*d)
363           return 0;
364       fseek((FILE*)pf, pos*tam, SEEK_SET);
365       return fread(*d, tam, 1, (FILE*)pf);
366   }
367
368
369   int cargarDesdeDatosOrdenadosRec(tArbolBinBusq * p, void * ds,
370                                    unsigned (*leer)(void **, void *, unsigned, void *
params),
371                                    int li, int ls, void * params)
372   {
373       int m = (li+ls)/2,
374           r;
375       if (li>ls)
376           return TODO_BIEN;
377
378       (*p) = (tNodoArbol*) malloc(sizeof(tNodoArbol));
```

```
379      if (!*p || !((*p)->tamInfo = leer(&(*p)->info, ds, m, params)))
380      {
381          free(*p);
382          return SIN_MEM;
383      }
384
385      (*p)->izq = (*p)->der = NULL;
386
387      if((r=cargarDesdeDatosOrdenadosRec(&(*p)->izq, ds, leer, li, m-1, params))!=
TODO_BIEN)
388          return r;
389      return cargarDesdeDatosOrdenadosRec(&(*p)->der, ds, leer, m+1, ls, params);
390  }
391
392
393  int cargarArchivoBinOrdenadoAbiertoArbolBinBusq(tArbolBinBusq * p, FILE * pf,
394                                                  unsigned tamInfo)
395  {
396      int cantReg;
397      if(*p || !pf)
398          return 0;
399      fseek(pf, 0L, SEEK_END);
400      cantReg = ftell(pf)/tamInfo;
401      return cargarDesdeDatosOrdenadosRec(p, pf, leerDesdeArchivoBin, 0, cantReg-1, &
tamInfo);
402  }
403
404  int cargarArchivoBinOrdenadoArbolBinBusq(tArbolBinBusq * p, const char * path,
405                                           unsigned tamInfo)
406  {
407      int cantReg,
408          r;
409      FILE * pf;
410      if(*p)
411          return SIN_INICIALIZAR;
412      if(!(pf= fopen(path, "rb")))
413          return ERROR_ARCH;
414      fseek(pf, 0L, SEEK_END);
415      cantReg = ftell(pf)/tamInfo;
416      r = cargarDesdeDatosOrdenadosRec(p, pf, leerDesdeArchivoBin, 0, cantReg-1, &
tamInfo);
417      fclose(pf);
418      return r;
419  }
420
421
422  int cargarDesdeDatosOrdenadosArbolBinBusq(tArbolBinBusq * p, void * ds, unsigned
cantReg,
423                                            unsigned (*leer)(void **, void *, unsigned
, void *params),
424                                            void * params)
425  {
426      if(*p || !ds)
427          return 0;
428      return cargarDesdeDatosOrdenadosRec(p, ds, leer, 0, cantReg-1, params);
429  }
430
431
432  tNodoArbol ** buscarNodoArbolBinBusq(const tArbolBinBusq * p, const void * d,
433                                       int (*cmp)(const void*, const void *))
434  {
435      int rc;
436      while(*p && (rc=cmp(d, (*p)->info)))
437      {
438          if(rc<0)
439              p = &(*p)->izq;
```

```
440         else
441             p = &(*p)->der;
442     }
443     if(!*p)
444         return NULL;
445     return (tNodoArbol **)p;
446 }
447
448
449 tNodoArbol ** buscarRecNodoArbolBinBusq(const tArbolBinBusq * p, const void * d,
450                                         int (*cmp)(const void*, const void *))
451 {
452     int rc;
453     if(!*p)
454         return NULL;
455     if(*p && (rc=cmp(d, (*p)->info)))
456     {
457         if(rc<0)
458             return buscarRecNodoArbolBinBusq(&(*p)->izq, d, cmp);
459         return buscarRecNodoArbolBinBusq(&(*p)->der, d, cmp);
460     }
461
462     return (tNodoArbol **) p;
463 }
464
465
466 tNodoArbol ** mayorNodoArbolBinBusq(const tArbolBinBusq * p)
467 {
468     if(!*p)
469         return NULL;
470     while((*p)->der)
471         p = &(*p)->der;
472     return (tNodoArbol **) p;
473 }
474
475
476 tNodoArbol ** menorNodoArbolBinBusq(const tArbolBinBusq * p)
477 {
478     if(!*p)
479         return NULL;
480     while((*p)->izq)
481         p = &(*p)->izq;
482     return (tNodoArbol **) p;
483 }
484
485
486 tNodoArbol ** mayorRecNodoArbolBinBusq(const tArbolBinBusq * p)
487 {
488     if(!*p)
489         return NULL;
490     if(!(*p)->der)
491         return (tNodoArbol **) p;
492     return mayorRecNodoArbolBinBusq(&(*p)->der);
493 }
494
495
496 tNodoArbol ** menorRecNodoArbolBinBusq(const tArbolBinBusq * p)
497 {
498     if(!*p)
499         return NULL;
500     if(!(*p)->izq)
501         return (tNodoArbol **) p;
502     return menorRecNodoArbolBinBusq(&(*p)->izq);
503 }
504
505
```

```c
int mayorElemNoClaveArbolBinBusq(const tArbolBinBusq *p, void *d, unsigned tam,
                                 int (*cmp)(const void *, const void *))
{
    const tArbolBinBusq * mayor = p;
    if (!*p)
        return 0;
    mayor = mayorNodoNoClaveArbolBinBusq(&(*p)->izq, mayor, cmp);
    mayor = mayorNodoNoClaveArbolBinBusq(&(*p)->der, mayor, cmp);

    memcpy(d, (*mayor)->info, MINIMO(tam, (*mayor)->tamInfo));

    return 1;
}

const tArbolBinBusq * mayorNodoNoClaveArbolBinBusq(const tArbolBinBusq *p, const
tArbolBinBusq *mayor,
                                 int (*cmp)(const void *, const void *))
{
    if (!*p)
        return mayor;
    if (cmp((*p)->info, (*mayor)->info)>0)
        mayor = p;
    mayor = mayorNodoNoClaveArbolBinBusq(&(*p)->izq, mayor, cmp);
    mayor = mayorNodoNoClaveArbolBinBusq(&(*p)->der, mayor, cmp);
    return mayor;
}

int menorElemNoClaveArbolBinBusq(const tArbolBinBusq *p, void *d, unsigned tam,
                                 int (*cmp)(const void *, const void *))
{
    const tArbolBinBusq * menor = p;
    if (!*p)
        return 0;
    menor = menorNodoNoClaveArbolBinBusq(&(*p)->izq, menor, cmp);
    menor = menorNodoNoClaveArbolBinBusq(&(*p)->der, menor, cmp);

    memcpy(d, (*menor)->info, MINIMO(tam, (*menor)->tamInfo));

    return 1;
}

const tArbolBinBusq * menorNodoNoClaveArbolBinBusq(const tArbolBinBusq *p, const
tArbolBinBusq *menor,
                                 int (*cmp)(const void *, const void *))
{
    if (!*p)
        return menor;
    if (cmp((*p)->info, (*menor)->info)<0)
        menor = p;
    menor = menorNodoNoClaveArbolBinBusq(&(*p)->izq, menor, cmp);
    menor = menorNodoNoClaveArbolBinBusq(&(*p)->der, menor, cmp);
    return menor;
}

int buscarElemNoClaveArbolBinBusq(const tArbolBinBusq *p, void *d, unsigned tam,
                                 int (*cmp)(const void *, const void *))
{
    const tArbolBinBusq * busq = NULL;
    if (!*p)
        return 0;

    if(cmp((*p)->info, d)==0)
        busq = p;
    else if ((busq = buscarNodoNoClaveArbolBinBusq(&(*p)->izq, d, cmp))==NULL){
        busq = buscarNodoNoClaveArbolBinBusq(&(*p)->der, d, cmp);
    }
```

```
570
571     if (busq==NULL)
572         return 0;
573
574     memcpy(d, (*busq)->info, MINIMO(tam, (*busq)->tamInfo));
575
576     return 1;
577 }
578
579 const tArbolBinBusq * buscarNodoNoClaveArbolBinBusq(const tArbolBinBusq *p, const
void *d,
580                                     int (*cmp)(const void *, const void *))
581 {
582     const tArbolBinBusq * busq = NULL;
583     if (!*p)
584         return NULL;
585     if (cmp((*p)->info, d)<0)
586         return p;
587     if((busq = buscarNodoNoClaveArbolBinBusq(&(*p)->izq, busq, cmp)))
588         return busq;
589     return buscarNodoNoClaveArbolBinBusq(&(*p)->der, busq, cmp);
590 }
591
592 unsigned alturaArbolBin(const tArbolBinBusq * p)
593 {
594     int hi,
595         hd;
596     if(!*p)
597         return 0;
598     hi= alturaArbolBin(&(*p)->izq);
599     hd= alturaArbolBin(&(*p)->der);
600     return (hi>hd ? hi : hd) +1;
601 }
602
603
604 unsigned cantNodosArbolBin(const tArbolBinBusq * p)
605 {
606     if(!*p)
607         return 0;
608     return cantNodosArbolBin(&(*p)->izq)
609          + cantNodosArbolBin(&(*p)->der)
610          + 1;
611 }
612
613
614 int mayorElemArbolBinBusq(const tArbolBinBusq * p, void * d, unsigned tam)
615 {
616     if(!(p = mayorNodoArbolBinBusq(p)))
617         return 0;
618     memcpy(d, (*p)->info, MINIMO(tam, (*p)->tamInfo));
619     return 1;
620 }
621
622
623 int menorElemArbolBinBusq(const tArbolBinBusq * p, void * d, unsigned tam)
624 {
625     if(!(p = menorNodoArbolBinBusq(p)))
626         return 0;
627     memcpy(d, (*p)->info, MINIMO(tam, (*p)->tamInfo));
628     return 1;
629 }
630
631 unsigned cantNodosHastaNivelArbolBin(const tArbolBinBusq *p, int n)
632 {
633     if(!*p)
634         return 0;
```

```
635     if(n==0)
636         return 1;
637     return cantNodosHastaNivelArbolBin(&(*p)->izq, n-1)
638          + cantNodosHastaNivelArbolBin(&(*p)->der, n-1)
639          + 1;
640 }
641
642 int esCompletoHastaNivelArbolBin(const tArbolBinBusq *p, int n)
643 {
644     if(!*p)
645         return n<0;
646     if(n==0)
647         return 1;
648     return esCompletoHastaNivelArbolBin(&(*p)->izq,n-1) &&
649            esCompletoHastaNivelArbolBin(&(*p)->der,n-1);
650 }
651
652 int esCompletoArbolBin(const tArbolBinBusq *p)
653 {
654     return esCompletoHastaNivelArbolBin(p,alturaArbolBin(p)-1);
655 }
656
657 int esBalanceadoArbolBin(const tArbolBinBusq *p)
658 {
659     return esCompletoHastaNivelArbolBin(p,alturaArbolBin(p)-2);
660 }
661
662 int esAVLArbolBin(const tArbolBinBusq *p)
663 {
664     int hi,
665         hd;
666     if(!*p)
667         return 1;
668     hi = alturaArbolBin(&(*p)->izq);
669     hd = alturaArbolBin(&(*p)->der);
670     if (abs(hi-hd)>1)
671         return 0;
672     return esAVLArbolBin(&(*p)->izq) &&
673            esAVLArbolBin(&(*p)->der);
674 }
675
676 int esCompleto2ArbolBin(const tArbolBinBusq *p)
677 {
678     int h = alturaArbolBin(p);
679     return cantNodosArbolBin(p) == pow(2,h)-1;
680 }
681
682 int esBalanceado2ArbolBin(const tArbolBinBusq *p)
683 {
684     int h = alturaArbolBin(p);
685     return cantNodosHastaNivelArbolBin(p, h-2) == pow(2,h-1)-1;
686 }
687
688 int esAVL2CalculoArbolBin(const tArbolBinBusq *p)
689 {
690     int hi,
691         hd;
692     if(!*p)
693         return 0;
694     hi= esAVL2CalculoArbolBin(&(*p)->izq);
695     hd= esAVL2CalculoArbolBin(&(*p)->der);
696     if(hi<0 || hd<0 || abs(hi-hd)>1)
697         return -1;
698     return (hi>hd ? hi : hd) +1;
699 }
700
```

```c
int esAVL2ArbolBin(const tArbolBinBusq *p)
{
    return esAVL2CalculoArbolBin(p)>=0;
}

/** ************************************************************************
 ** FIN -arbol_bin_busq.c    definición primitivas  TDA ÁRBOL  bin. de búsqueda
 ** ********************************************************************** */


 /** ************************************************************************
 **     arbol_bin_busq.h    declaración primitivas  TDA ÁRBOL  bin. de búsqueda
 ** ********************************************************************** */

#ifndef ARBOL_BIN_BUSQ_H_INCLUDED
#define ARBOL_BIN_BUSQ_H_INCLUDED

#include<string.h>
#include<stdlib.h>
#include<stdio.h>
#include<math.h>

#define CLA_DUP 0
#define SIN_MEM 0
#define SIN_INICIALIZAR 0
#define ERROR_ARCH 0
#define TODO_BIEN 1


typedef struct sNodoArbol
{
    void *info;
    unsigned tamInfo;
    struct sNodoArbol *izq,
                      *der;
}tNodoArbol;

typedef tNodoArbol *tArbolBinBusq;



void crearArbolBinBusq(tArbolBinBusq *p);

int insertarArbolBinBusq(tArbolBinBusq *p, const void *d, unsigned tam,
                         int (*cmp)(const void *, const void *));

int insertarRecArbolBinBusq(tArbolBinBusq *p, const void *d, unsigned tam,
                            int (*cmp)(const void*, const void *));

void recorrerEnOrdenArbolBinBusq(const tArbolBinBusq * p, void * params,
                                 void (*accion)(void *, unsigned, unsigned, void
*));

void recorrerEnOrdenInversoArbolBinBusq(const tArbolBinBusq * p, void * params,
                                        void (*accion)(void *, unsigned, unsigned,
void *));

void recorrerPreOrdenArbolBinBusq(const tArbolBinBusq * p, void * params,
                                  void (*accion)(void *, unsigned, unsigned, void
*));

void recorrerPosOrdenArbolBinBusq(const tArbolBinBusq * p, void * params,
                                  void (*accion)(void *, unsigned, unsigned, void
*));

void recorrerEnOrdenSimpleArbolBinBusq(const tArbolBinBusq *p, void *params,
```

```
763                                         void (*accion)(void *, unsigned, void *));
764
765 void recorrerPreOrdenSimpleArbolBinBusq(const tArbolBinBusq *p, void *params,
766                                         void (*accion)(void *, unsigned, void *));
767
768 void recorrerPosOrdenSimpleArbolBinBusq(const tArbolBinBusq *p, void *params,
769                                         void (*accion)(void *, unsigned, void *));
770
771 int eliminarRaizArbolBinBusq(tArbolBinBusq *p);
772
773 int eliminarElemArbolBinBusq(tArbolBinBusq *p, void *d, unsigned tam,
774                              int (*cmp)(const void *, const void *));
775
776 int buscarElemArbolBinBusq(const tArbolBinBusq *p, void *d, unsigned tam,
777                              int (*cmp)(const void *, const void *));
778
779 int cargarArchivoBinOrdenadoAbiertoArbolBinBusq(tArbolBinBusq *p, FILE *pf,
780                                         unsigned tamInfo);
781
782 int cargarArchivoBinOrdenadoArbolBinBusq(tArbolBinBusq * p, const char * path,
783                                         unsigned tamInfo);
784
785 int cargarDesdeDatosOrdenadosArbolBinBusq(tArbolBinBusq *p,
786                         void *ds, unsigned cantReg,
787                         unsigned (*leer)(void **, void *, unsigned, void *params),
788                         void * params);
789
790
791 int mayorElemNoClaveArbolBinBusq(const tArbolBinBusq *p, void *d, unsigned tam,
792                                  int (*cmp)(const void *, const void *));
793
794 int menorElemNoClaveArbolBinBusq(const tArbolBinBusq *p, void *d, unsigned tam,
795                                  int (*cmp)(const void *, const void *));
796
797 int buscarElemNoClaveArbolBinBusq(const tArbolBinBusq *p, void *d, unsigned tam,
798                                  int (*cmp)(const void *, const void *));
799
800
801
802 ///Utils
803
804 unsigned alturaArbolBin(const tArbolBinBusq *p);
805
806 unsigned cantNodosArbolBin(const tArbolBinBusq *p);
807
808 unsigned cantNodosHastaNivelArbolBin(const tArbolBinBusq *p, int n);
809
810 int mayorElemArbolBinBusq(const tArbolBinBusq *p, void *d, unsigned tam);
811
812 int menorElemArbolBinBusq(const tArbolBinBusq *p, void *d, unsigned tam);
813
814 int esCompletoArbolBin(const tArbolBinBusq *p);
815
816 int esBalanceadoArbolBin(const tArbolBinBusq *p);
817
818 int esAVLArbolBin(const tArbolBinBusq *p);
819
820 int esCompleto2ArbolBin(const tArbolBinBusq *p);
821
822 int esBalanceado2ArbolBin(const tArbolBinBusq *p);
823
824 int esAVL2ArbolBin(const tArbolBinBusq *p);
825
826
827 #endif // ARBOL_BIN_BUSQ_H_INCLUDED
828
```

```
/** ************************************************************************
 **  FIN - arbol_bin_busq.h  declaración primitivas  TDA ÁRBOL  bin. de búsqueda
 ** ************************************************************************ */
```