EJERCICIOS DE ÁRBOLES BINARIOS

1) Supongamos que tenemos una función valor tal que dado un valor de tipo char (una letra del alfabeto) devuelve un valor entero asociado a dicho identificador. Supongamos también la existencia de un árbol de expresión T cuyos nodos hoja son letras del alfabeto y cuyos nodos interiores son los caracteres *, +, -, /. Diseñar una función que tome como parámetros un nodo y un árbol binario y devuelva el resultado entero de la evaluación de la expresión representada.

2) El recorrido en preorden de un determinado árbol binario es: GEAIBMCLDFKJH y en inorden IABEGLDCFMKHJ.

   - Dibujar el árbol binario.
   - Dar el recorrido en postorden.
   - Diseñar una función para dar el recorrido en postorden dado el recorrido en preorden e inorden y escribir un programa para comprobar el resultado del apartado anterior.

3) Implementar una función no recursiva para recorrer un árbol binario en inorden.

4) Implementar una función no recursiva para recorrer un árbol binario en postorden.

5) Escribir una función recursiva que encuentre el número de nodos de un árbol binario.

6) Escribir una función recursiva que encuentre la altura de un árbol binario.

EJERCICIOS DE ÁRBOLES BINARIOS DE BÚSQUEDA (ABB)

1) ¿Puede reconstruirse de forma única un ABB dado su inorden? ¿Y dados el preorden y el postorden?.

2) Construir un ABB con las claves 50, 25, 75, 10, 40, 60, 90, 35, 45, 70, 42.

3) Construir un ABB equilibrado a partir de las claves 10, 75, 34, 22, 64, 53, 41, 5, 25, 74, 20, 15, 90.

4) ¿Bajo qué condiciones puede un árbol ser parcialmente ordenado y binario de búsqueda simultáneamente?. Razonar la respuesta.

Árboles AVL.

Supongamos que deseamos construir un ABB para la siguiente tabla de datos:

1 A	6 H	10 G
2 C	7 I	11 J
3 D	8 B	12 K
4 M	9 F	13 E
5 L	-	-

Valores ejemplo para un ABB.

El resultado se muestra en la figura siguiente:

Como ve el resultado es un árbol muy poco balanceado y con características muy pobres para la búsqueda. Los ABB trabajan muy bien para una amplia variedad de aplicaciones, pero tienen el problema de que la eficiencia en el peor caso es O(n). Los árboles que estudiaremos a continuación nos darán una idea de cómo podría resolverse el problema garantizando en el peor caso un tiempo O(log2 n).

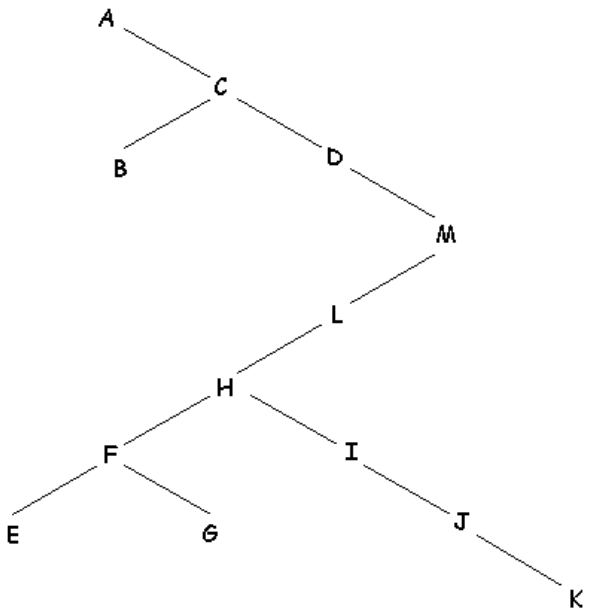
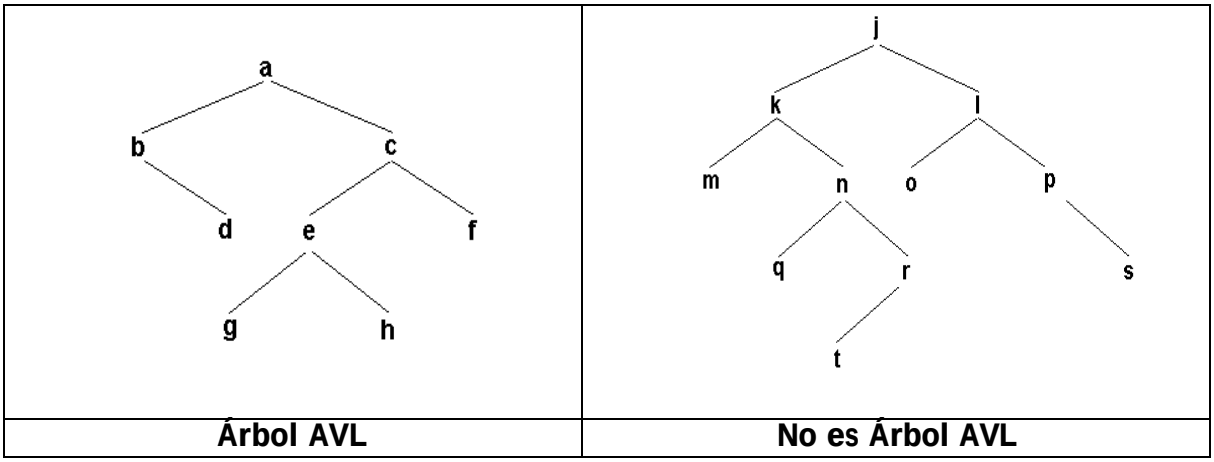


ABB poco balanceado.

ÁRBOLES EQUILIBRADOS AVL.

Diremos que un árbol binario está equilibrado (en el sentido de Addelson-Velskii y Landis) si, para cada uno de sus nodos ocurre que las alturas de sus dos subárboles difieren como mucho en 1. Los árboles que cumplen esta condición son denominados a menudo árboles AVL.

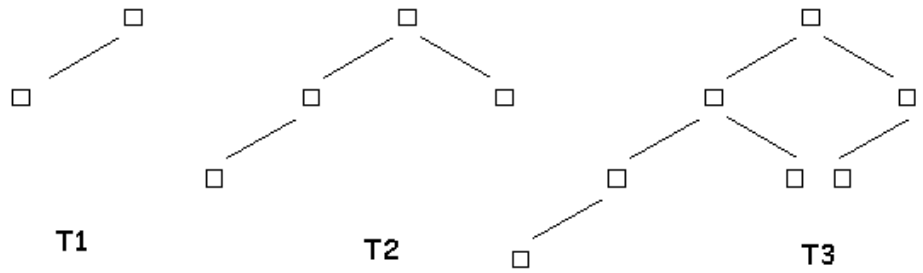
En la primera figura se muestra un árbol que es AVL, mientras que el de la segunda no lo es al no cumplirse la condición en el nodo ***k***.



Árbol AVL	**No es Árbol AVL**

A través de los árboles AVL llegaremos a un procedimiento de búsqueda análogo al de los ABB pero con la ventaja de garantizaremos un caso peor de O(log2 n), manteniendo el árbol en todo momento equilibrado. Para llegar a este resultado, podríamos preguntarnos cual podría ser el peor AVL que podríamos construir con ***n*** nodos, o dicho de otra forma cuanto podríamos permitir que un árbol binario se desequilibrara manteniendo la propiedad de AVL.

Para responder a la pregunta podemos construir para una altura ***h*** el AVL ***Th***, con mínimo número de nodos. Cada uno de estos árboles mínimos debe constar de una raíz, un subárbol AVL mínimo de altura ***h-1*** y otro subárbol AVL también mínimo de altura ***h-2***. Los primeros ***Ti*** pueden verse en la siguiente figura:



Los primeros Ti.

Es fácil ver que el número de nodos ***n(Th)*** está dado por la relación de recurrencia [1]:

n(Th) = 1 + n(Th-1) + n(Th-2)

Relación similar a la que aparece en los números de Fibonacci (***Fn = Fn-1 + Fn-2***), de forma que la siguiente sucesión, de valores para ***n(Th)*** está relacionada con los valores de la siguiente sucesión de Fibonacci:

AVL -> -, -, 1, 2, 4, 7, 12,...
FIB -> 1, 1, 2, 3, 5, 8, 13,...

es decir ***[2]***,

n(Th) = Fh+2 - 1

Resolviendo [1] y utilizando [2] llegamos tras algunos cálculos a:

$$log_2(n+1) <= h < 1.44\ log_2(n+2)-0.33$$

o dicho de otra forma, la longitud de los caminos de búsqueda (o la altura) para un AVL de ***n nodos***, nunca excede al 44% de la longitud de los caminos (o la altura) de un árbol completamente equilibrado con esos ***n*** nodos. En consecuencia, aún en el peor de los casos llevaría un tiempo ***O(log2 n)*** al encontrar un nodo con una clave dada.

Parece, pues, que el único problema es el mantener siempre tras cada inserción la condición de equilibrio, pero esto puede hacerse muy fácilmente sin más que hacer algunos reajustes locales, cambiando punteros.

Antes de estudiar detalladamente este tipo de árboles realizamos la declaración de tipos siguiente:

```
typedef int tElemento;

typedef struct NODO_AVL {
   tElemento elemento;
   struct AVL_NODO *izqda;
   struct AVL_NODO *drcha;
   int altura;
} nodo_avl;

typedef nodo_avl *arbol_avl;

#define AVL_VACIO NULL

#define maximo(a,b) ((a>b)?(a):(b))
```

En muchas implementaciones, para cada nodo no se almacena la altura real de dicho nodo en el campo que hemos llamada altura, en su lugar se almacena un valor del conjunto {-1, 0, 1} indicando la relación entre las alturas de sus dos hijos. En nuestro caso almacenamos la altura real por simplicidad. Por consiguiente podemos definir la siguiente macro:

```
#define altura(n) (n?n->altura:-1)
```

La cual nos devuelve la altura de un nodo_avl.

Con estas declaraciones la funciones de creación y destrucción para los árboles AVLpueden ser como sigue:

```
arbolAVL Crear_AVL()
{
   return AVL_VACIO;
}

void Destruir_AVL (arbolAVL A)
{
  if (A) {
      Destruir_AVL(A->izqda);
      Destruir_AVL(A->drcha);
      free(A);
   }
}
```

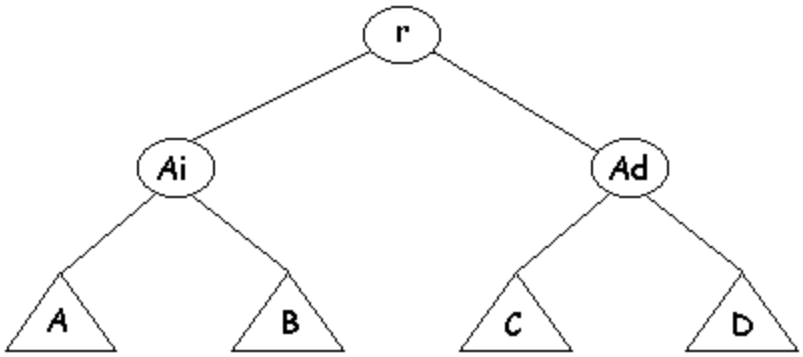
Es sencillo realizar la implementación de una función que podemos llamar miembro que nos devuelve si un elemento pertenece al árbol AVL. Podría ser la siguiente:

```
int miembro_AVL(tElemento e,arbolAVL A)
{
   if (A == NULL)
       return 0;

   if (e == A->elemento)
       return 1;
   else
      if (e < A->elemento)
         return miembro_AVL(e,A->izqda);
      else
         return miembro_AVL(e,A->drcha);
}
```

Veamos ahora la forma en que puede afectar una inserción en un árbol AVL y la forma en que deberíamos reorganizar los nodos de manera que siga equilibrado. Consideremos el esquema general de la siguiente figura, supongamos que la inserción ha provocado que el subárbol que cuelga de ***Ai*** pasa a tener una altura 2 unidades mayor que el subárbol que cuelga de ***Ad***. ¿Qué operaciones son

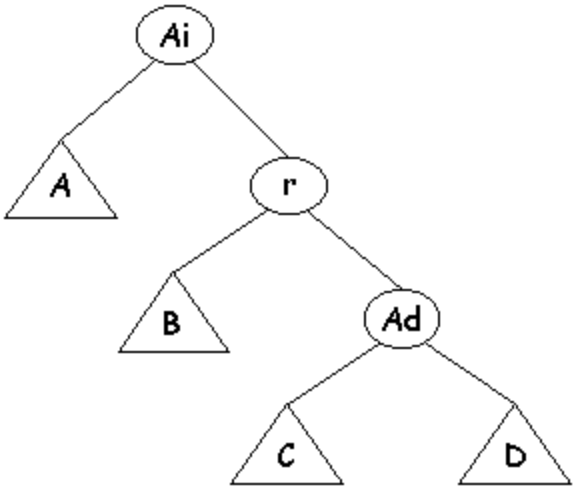
necesarias para que el nodo ***r*** tenga 2 subárboles que cumplan la propiedad de árboles AVL?.



Esquema general de árbol AVL.

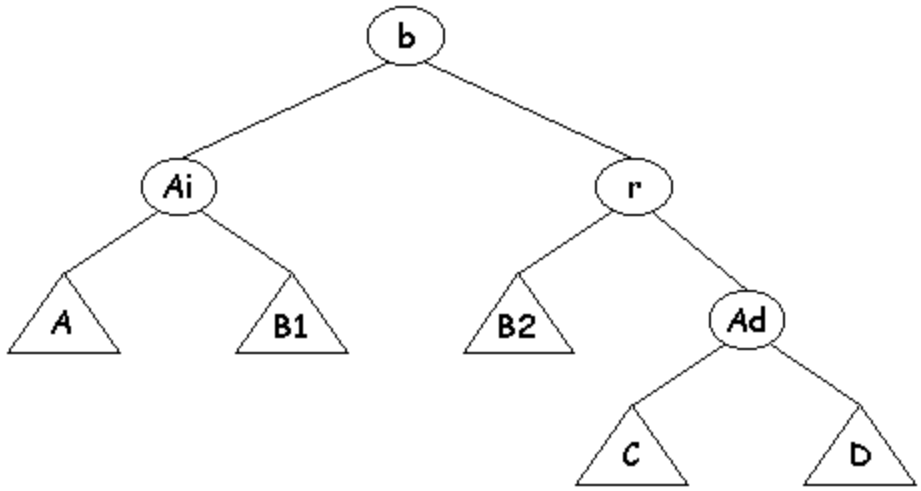
Para responder a esto estudiaremos dos situaciones distintas que requieren 2 secuencias de operaciones distintas:

La inserción se ha realizado en el árbol A. La operación a realizar es la de una rotación simple a la derecha sobre el nodo r resultando el árbol mostrado en la siguiente figura.



Rersultado de rotación simple a la derecha sobre r.

La inserción se ha realizado en el árbol B. (supongamos tiene raíz b, subárbol izquierdo B1 y subárbol derecho B2). La operación a realizar es la rotación doble izquierda-derecha la cual es equivalente a realizar una rotación simple a la izquierda sobre el nodo Ai y después una rotación simple a la derecha sobre el nodo r (por tanto, el árbol B queda dividido). El resultado se muestra en la figura siguiente:



Resultado de rotación doble izquierda-derecha sobre r.

En el caso de que la inserción se realice en el subárbol Ad la situación es la simétrica y para las posibles violaciones de equilibrio se aplicará la misma técnica mediante la rotación simple a la izquierda o la rotación doble izquierda-derecha. Se puede comprobar que si los subárboles Ad y Ai son árboles AVL, estas operaciones hacen que el árbol resultante también sea AVL. Por último, destacaremos que para realizar la implementación definitiva en base a la declaración de tipos que hemos propuesto tendremos que realizar un ajuste de la altura de los nodos involucrados en la rotación además del ya mencionado ajuste de punteros. Por ejemplo: En la rotación simple que se ha realizado en la primera de las situaciones, el campo de altura de los nodos r y Ai puede verse modificado.

Estas operaciones básicas de simple y doble rotación se pueden implementar de la siguiente forma:

```
void Simple_derecha(arbolAVL *A)
{
   nodoAVL *p;

   p = (*A)->izqda;
   (*A)->izqda = p->drcha;
   p->drcha = (*A);
   (*A) = p;
   /* Ajustamos las alturas */
   p = (*A)->drcha;
   p->altura = maximo(altura(p->izqda),altura(p->drcha))+1;
   (*A)->altura = maximo(a1tura((*T)->izqda),altura((*T)->drcha))+1;
}

void Simple_izquierda(arbolAVL *A)
{
   nodoAVL *p;
```

```
   p = (*A)->drcha;
   (*A)->drcha = p->izqda;
   p->izqda = (*A);
   (*A) = p;
   /*Ajustamos las alturas */
   p = (*A)->izqda;
   p->altura = maximo(altura(p->izqda),altura(p->drcha))+1;
   (*A)->altura = maximo(altura((*A)->izqda),altura((*A)->drcha))+1;
}

void Doble_izquierda_derecha (arbolAVL *AT)
{
   simple_izquierda(&((*A)->izqda));
   simple_derecha(A);
}

void Doble_derecha_izquierda (arbolAVL *A)
{
   simple_derecha(&((*A)->drcha));
   simple_izquierda(A);
}
```

Obviamente, el reajuste en los nodos es necesario tanto para la operación de inserción como para la de borrado. Por consiguiente, se puede programar la inserción de forma que descendamos en el árbol hasta llegar a una hoja donde insertar y después recorrer el mismo camino hacia arriba realizando los ajustes necesarios (igualmente en el borrado se realizaría algo similar). Para hacer más fácil la implementación, construiremos la función `ajusta_avl(e,&T)` cuya misión consiste en ajustar los nodos que existen desde el nodo conteniendo la etiqueta e hasta el nodo raiz en el árbol T. La usaremos como función auxiliar para implementar las funciones de inserción y de borrado. El código es el siguiente:

```
void ajusta_AVL (tElemento e, arbolAVL *A)
{
   if (!(*A))
      return;
   if (e > (*A)->elemento)
      ajusta_AVL(e,&((*A)->drcha));
   else if (e < (*A)->elemento)
           ajusta_avl(e,&((*A)->izqda));

   switch (altura((*A)->izqda)-altura((*A)->drcha)) {
   case 2:
      if (altura((*A)->izqda->izqda) > altura((*A)->izqda->drcha))
         simple_derecha(A);
      else doble_izquierda_derecha(A);
      break;
   case -2:
      if (altura((*A)->drcha->drcha) > altura((*A)->drcha->izqda))
         simple_izquierda(A);
      else doble_derecha_izquierda(A);
      break;
```

```
   default:
      (*A)->altura = maximo(altura((*A)->izqda),altura((*A)->drcha))+1;
   }
}
```

Para la operación de inserción se deberá profundizar en el árbol hasta llegar a un nodo hoja o un nodo con un solo hijo de forma que se añade un nuevo hijo con el elemento insertado. Una vez añadido sólo resta ajustar los nodos que existen en el camino de la raíz al nodo insertado. El código es el siguiente:

```
void insertarAVL (tElemento e, arbolAVL *A)
{
   nodoAVL **p;

   p=T;
   while (*p!=NULL)
      if ((*p)->elemento > e)
         p = &((*p)->izqda);
      else p = &((*p)->drcha);

   (*p)=(nodo_avl *)malloc(sizeof(nodoAVL));
   if (!(*p))
      error("Error: Memoria insuficiente.");

   (*p)->elemento = e;
   (*p)->altura = 0;
   (*p)->izqda = NULL;
   (*p)->drcha = NULL;
   ajustaAVL(e,A);
}
```

En el caso de la operación de borrado es un poco más complejo pues hay que determinar el elemento que se usará para la llamada a la función de ajuste. Por lo demás es muy similar al borrado en los árboles binarios de búsqueda. En la implementación que sigue usaremos la variable elem para controlar el elemento involucrado en la función de ajuste.

```
void borrarAVL (tElemento e, arbolAVL *A)
{
   nodoAVL **p,**aux,*dest;
   tElemento elem;

   p=A;
   elem=e;
   while ((*p)->elemento!=e) {
      elem=(*p)->elemento;
      if ((*p)->elemento > e)
         p=&((*p)->izqda);
      else p=&((*p)->drcha);
   }

   if ((*p)->izqda!=NULL && (*p)->drcha!=NULL) {
      aux=&((*p)->drcha);
      elem=(*p)->elemento;
      while ((*aux)->izqda) {
```

```
            elem=(*aux)->elemento;
            aux=&((*aux)->izqda);
         }
         (*p)->elemento = (*aux)->elemento;
         p=aux;
      }
      if ((*p)->izqda==NULL && (*p)->drcha==NULL) {
         free(*p);
         (*p) = NULL;
      } else if ((*p)->izqda == NULL) {
         dest = (*p);
         (*p) = (*p)->drcha;
         free(dest);
      } else {
         dest = (*p);
         (*p) = (*p)->izqda;
         free(dest);
      }
      ajustaAVL(elem,A);
}
```