ASIGNATURA		CURSO	CALIFICACIÓN
ESTRUCTURA DE DATOS Y DE LA INFORMACIÓN		**2003 / 2004**	
TITULACIÓN	GRUPO	CONVOCATORIA	
		EXTRAORDINARIA-SEPTIEMBRE	
APELLIDOS		NOMBRE	

EJERCICIO 1 (3 PUNTOS)

Dado el tipo abstracto de datos (TAD) `tArbolBin` que sirve para representar árboles binarios con nodos coloreados (rojo o negro), del que sólo se conoce la parte de la interfaz siguiente:

```
Type
   tValor = …;
   tColor = (rojo,negro);
   tInfo = record
            info: tValor;
            color: tColor;
           end;
   tArbolBin = …
   function EsArbolVacio(a: tArbolBin): boolean;
   function Color       (a: tArbolBin): tColor;
   function RamaIzqda   (a: tArbolBin): tArbolBin;
   function RamaDcha    (a: tArbolBin): tArbolBin;

   (Las funciones Color, RamaIzqda y RamaDcha tienen como
   precondición que el árbol no sea vacio)
```

A) Definir una operación que cuente el número de nodos rojos de un árbol binario.

`numNodosRojos(tArbolBin)→ integer`

B) Un *árbol rojo-negro* cumple las siguientes propiedades:

1. Todos los nodos hoja tienen color negro
2. El número de nodos rojos del subárbol izquierdo y del subárbol derecho difiere como mucho en 1.
3. Tanto el subárbol derecho como el izquierdo son árboles *rojo-negro*

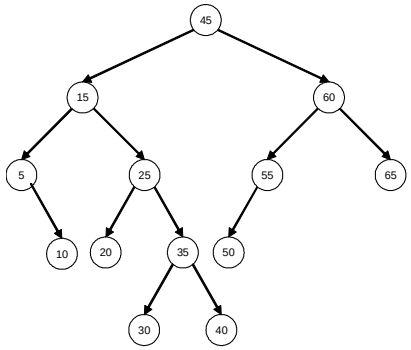
(NOTA: El árbol vacío es rojo-negro)

Construir una operación que decida si un árbol es rojo-negro.

`EsArbolRojoNegro (tArbolBin)→ boolean`

EJERCICIO 2 (3 PUNTOS)

A) Sea el árbol binario AVL de la figura siguiente:

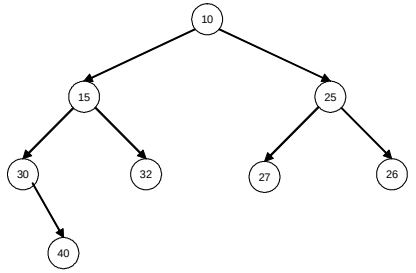


Se pide:

Realizar las operaciones que se detallan a continuación, dibujar el estado del árbol después de cada operación indicando el Factor de Equilibrio (FE) de cada nodo, reflejar las reestructuraciones que sean necesarias (dibujando los estados intermedios) e indicar el tipo de rotación aplicado cuando proceda.

- Inserción del elemento 28
- Sobre el resultado de la inserción anterior, eliminar el elemento 65

B) Dada la estructura de la figura siguiente:



Se pide:

- Marcar con una cruz qué tipo(s) de árbol representa, justificando la respuesta:

	Si	No	Justificación
Montículo Min			
Árbol binario de búsqueda			
Árbol lleno			
Árbol AVL			

EJERCICIO 3 (4 PUNTOS)

Una tabla de símbolos es una estructura que almacena convenientemente pares de la forma (nombre, lista de valores), donde cada nombre tiene asociado un conjunto de valores enteros posibles. La tabla se encuentra ordenada por nombres, y el número máximo de nombres y valores posibles no es conocido a priori.

Esta estructura se implementa mediante una multilista que mantiene **ordenadamente** los nombres en una lista y, además, hace uso del *TADLista* para guardar, para cada nombre, las lista de valores asociados.

Se pide:

A) Realizar en lenguaje Pascal la definición de tipos del *TADLista* (que mantiene una lista de valores) y el *TADTabladeSimbolos* (que se corresponde con la estructura completa de nombres y valores). Deberá justificarse la elección de la implementación de ambas (estática o dinámica).

B) Suponiendo implementadas las funciones siguientes para el TADLista,

ListaVacia (Lista) → Lista
{Objetivo: Crear una lista vacía
Salida: Una lista vacía}

EsListaVacía (Lista) → Boolean
{Objetivo: Determinar si una lista está vacía
Entrada: *Lista*
Salida: *true* si la lista está vacía, *false* en caso contrario.
PreCond: La *Lista* debe estar inicializada}

Insertar (Lista, valor) → Lista
{Objetivo: Añadir a la lista un valor (se permiten repetidos)
Entrada: *Lista* y *Valor*
Salida: *Lista* con un nuevo *valor*
PreCond: La *lista* debe estar inicializada, y se supone memoria suficiente}

BorrarLista (Lista) → Lista
{Objetivo: Elimina todos los elementos de la *lista*
Entrada: *Lista* a eliminar
Salida: *Lista* vacia
PreCond: La *lista* debe estar inicializada}

Realizar la implementación de las operaciones definidas a continuación para el *TADTabladeSimbolos*, usando la definición de tipos especificada en el apartado **A). `Tabla`** es de tipo *tTabla*; **`nombre`** es de tipo *tNombre*, **`valor`** es de tipo *tValor,* y ***`tLista`*** es la lista de valores.

TablaVacia (Tabla) → Tabla
{Objetivo: Crear una Tabla vacía
Salida: Una Tabla vacía}

EsTablaVacia (Tabla) → Boolean
{Objetivo: Determinar si una Tabla está vacía
Entrada: *Tabla*
Salida: *true* si la tabla está vacía, *false* en caso contrario.
PreCond: La *tabla* debe estar inicializada }

Existe (Tabla, nombre) → boolean
{Objetivo: Indica si el *nombre* está en en la *tabla*
Entrada: *Tabla* de nombres y valores, *Nombre* a buscar
Salida: *true* si el *nombre* está en el *Tabla*, *false* en caso contrario
PreCond: La *tabla* debe estar inicializada }

Valores (Tabla, nombre) → Lista
{Objetivo: Devuelve la lista de valores asociados al *nombre*
Entrada: *Tabla* de nombres y valores, y *nombre* a encontrar
Salida: Lista de valores asociados
PreCond: La *tabla* debe estar inicializada, y el *nombre* debe existir en la *tabla* }

InsertarEnTabla (Tabla, nombre, valor) → Tabla
{Objetivo: Añadir a la tabla un par elemento-valor
Entrada: *Tabla, Nombre* y *Valor*
Salida: *Tabla* con un nuevo *valor* para el *nombre* si éste ya existía,
o un nuevo par nombre-valor si el *nombre* no pertenecía a la tabla
PreCond: La *tabla* debe estar inicializada, y se supone memoria suficiente}

BorrarEnTabla (Tabla, nombre) → Tabla
{Objetivo: Elimina el *nombre* de la *tabla* y la *lista* de valores asociados
Entrada: *Tabla, Nombre* a buscar
Salida: La *tabla* sin el *nombre*
PreCond: El *nombre* debe pertenecer a la *tabla* }

SOLUCIONES

EJERCICIO 1

A) 1'5 puntos

```
function numNodosRojos (a: tArbolBin): integer;
begin
  if EsArbolVacio(a)
  then numNodosRojos:= 0
  else if Color(a)=rojo
  then numNodosRojos:= 1 + numNodosRojos (RamaIzqda(a))
                         + numNodosRojos(RamaDcha(a))
  else numNodosRojos:= numNodosRojos (RamaIzqda(a))
                         + numNodosRojos (RamaDcha(a));
end;
```

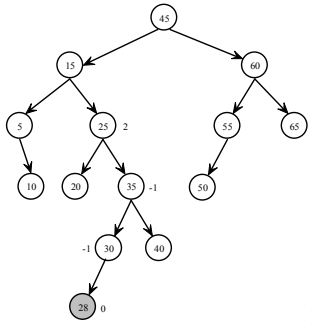
B) 1,5 puntos

```
function EsArbolRojoNegro (a: tArbolBin): boolean;
begin
  if EsArbolVacio (a)
  then EsArbolRojoNegro:= true                        (* Arbol vacio *)
  else if EsArbolVacio (RamaIzqda(a)) and
          EsArbolVacio (RamaDrcha(a))                 (* Hoja *)
       then EsArbolRojoNegro:= Color(a)= negro        (* Es negro *)
       else EsArbolRojoNegro:= (abs(numNodosRojos(RamaIzqda(a))
                              - numNodosRojos(RamaDcha(a)))<=1) and
                                 EsArbolRojoNegro(RamaIzqda(a)) and
                                 EsArbolRojoNegro(RamaDcha(a));
end;
```

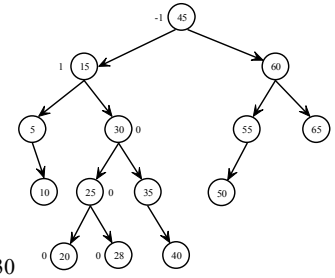
EJERCICIO 2

A) 1,5 puntos

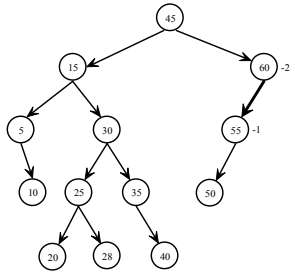
Insertar 28



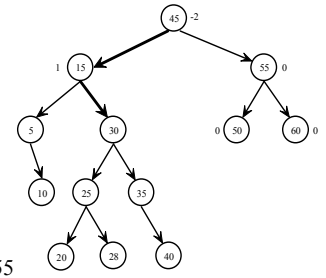
DI 25-35-30



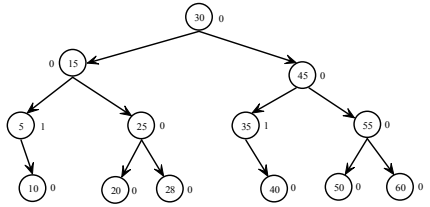
Eliminar 65



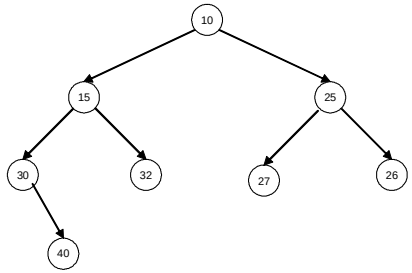
II:60-55



ID 45-15-30



B) 1,5 puntos



	Si	No	Justificación
Montículo		X	El valor de la clave de cada nodo es menor que cualquiera de las de sus hijos, pero el árbol no es completo (lleno hasta el penúltimo nivel y los nodos del último nivel lo más a la izquierda posible)
Árbol binario de búsqueda		X	El valor de las claves de cada nodo debe ser mayor que todas las de su subárbol izquierdo y menor que todas las de su subárbol derecho
Árbol lleno		X	No todos los nodos tienen dos descendientes
Árbol AVL		X	No es árbol binario de búsqueda y por tanto, no puede ser árbol AVL, aunque está equilibrado

EJERCICIO 2

A)

La implementación más adecuada, tanto para la lista que mantiene los nombres como para la que mantiene los valores, es la dinámica ya que, en ningún caso el número máximo de valores o nombre es conocido ni estable. La definición de tipos sería la siguiente:

```
UNIT TADLista;
INTERFACE
type
        tvalor = integer;
        tPosLista   = ^tNodoLista;
        tNodoLista = record
                        valor: tvalor;
                        sig: tPosLista;
        end;
        tLista = tPosLista;
```

y

```
UNIT TADTabladeSimbolos;
INTERFACE
        USES TADLista;
        const
                nulo=nil;
        type
                tNombre = string;
                tPosTabla   = ^tNodoTabla;
                tNodoTabla = record
                                nombre: tNombre;
                                valores: tLista;
                                sig: tPosTabla;
                end;
                tTabla = tPosTabla;
```

B) Definición + TablaVacia + EsTablaVacia = 0.75
Insertar = 1
Borrar = 1
Existe = 0.625
Valores = 0.625

```
Procedure TablaVacia (var Tabla: tTabla);
{Objetivo:          Crear una Tabla vacía
Salida:     Una Tabla vacía}
begin
        Tabla:=nulo;
end;

function EsTablaVacia (Tabla: tTabla): Boolean;
{Objetivo:          Determinar si una Tabla está vacía
Entrada:   Tabla
Salida:     true si la tabla está vacía, false en caso contrario.
PreCond: La tabla debe estar inicializada }
Begin
         EsTablaVacia:=(Tabla=nulo);
End;
```

```
Function Existe (Tabla: tTabla; nombre: tNombre): boolean;
{Objetivo:          Indica si el nombre está en en la tabla
Entrada:   Tabla, Nombre a buscar
Salida:     true si el nombre está en el Tabla, false en caso contrario
PreCond: La tabla debe estar inicializada }
begin
        if EsTablaVacia (Tabla)
        then Existe:=false
        else begin
                while (Tabla ^. nombre < nombre) and (Tabla ^.sig <> nulo) do
                        Tabla:= Tabla ^.sig;
                Existe:= (Tabla ^. nombre = nombre);
        end
end;
```

```
function Valores (Tabla: tTabla; nombre: tNombre): tLista;
{Objetivo:          Devuelve la lista de valores asociados al nombre
Entrada:   Tabla y nombre a encontrar
Salida:     Lista de valores asociados
PreCond: La tabla debe estar inicializada, y el nombre debe existir en la tabla }
begin
        while (Tabla ^. nombre <> nombre) do
                        Tabla:= Tabla ^.sig;
        Valores:= Tabla^.valores;
end;
```

Para implementar la operación de Inserción desarrollamos las dos operaciones auxiliares siguientes:

```
Procedure Crear_nodo (var nuevo: tPosTabla; nombre: tNombre; valor: tValor);
{Objetivo: crea la variable dinámica asociada a un puntero
Entrada: n: nombre a almacenar en la variable dinámica
        nuevo: puntero a la nueva variable creada
Salida:   nuevo apuntará a la nueva variable creada
PosCond: la variable tiene todos sus campos inicializados}
begin
        new(nuevo);
        nuevo ^.nombre:=nombre;
        nuevo ^.sig:=nulo;
        Insertar (nuevo^.valores, valor);
end;
```

```
function Pos_inser (n: tnombre; Tabla: tTabla ): tPosTabla ;
{Objetivo: buscar la posición de un elemento (si existe) en una tabla ordenada por el campo n,
o la posición donde deberá insertarse si no existe.
Entradas: n, campo de ordenación
          Tabla, acceso a la lista
Salidas: la posición del elemento buscado o del nodo anterior a donde se deberá insertar si no
existe, o nulo si se ha de insertar en primer lugar
Precond: Tabla no vacía}
var anterior: tPosTabla;    {posición del nodo después del que habrá que insertar}
begin
        anterior:= nulo;
        while (n > Tabla^.nombre) and (Tabla ^.sig <>nulo) do
        begin
                anterior:= Tabla;
                Tabla:= Tabla ^.sig;
        end;
        if      n >= Tabla ^. nombre      {Hay que insertar al final o el elemento ya existe}
        then    anterior:= Tabla;
        Pos_inser:=anterior;
end;


procedure InsertarEnTabla (var Tabla: tTabla; nombre: tNombre; valor: tValor);
{Objetivo: Añadir a la tabla un par elemento-valor
Entrada:  Tabla, Nombre y Valor
Salida:    Tabla con un nuevo valor para el nombre si éste ya existía,
           o un nuevo par si el nombre no pertenecía a la tabla
PreCond: La tabla debe estar inicializada, y se supone memoria suficiente}
var pos, nuevo: tPosTabla;
begin
        Crear_nodo (nuevo,nombre,valor);
        if EsTablaVacia(Tabla) then Tabla:=nuevo
        else begin
                pos:= Pos_inser (nombre, Tabla);
                if pos =nulo then  {se añade como primer nodo}
                begin
                        nuevo^.sig:= Tabla;
                        Tabla:=nuevo;
                end
                else    if pos ^.nombre <> nombre
                        then begin   {no existe ese nombre y se añade en el medio o al final}
                                nuevo^.sig:= pos ^.sig;
                                pos ^.sig:=nuevo;
                        end
                        else {el nombre ya existe en la lista, solo hay que insertar el valor}
                                Insertar (pos^.valores, valor);
        end
end;
```

```
procedure BorrarEnTabla (var Tabla: tTabla; nombre: tNombre);
{Objetivo: Elimina el nombre de la tabla y la lista de valores asociados
Entrada:  Tabla, Nombre a buscar
Salida:    La tabla sin el nombre
PreCond: El nombre debe pertenecer a la tabla }
Var       pos: tPosTabla;
begin
        if  nombre=Tabla^.nombre{si hay que eliminar el primero}
        then  begin
                borrar:= Tabla;
                Tabla:= Tabla^.sig
        end
        else begin                          {si hay que eliminar al medio o al final}
                pos:=Tabla;
                while (pos^.sig^. nombre <> nombre) do
                        pos:= pos^.sig;
                {al salir del bucle pos^.sig  es el nodo a eliminar}
                borrar:= pos^.sig;
                pos^.sig:= pos^.sig^.sig;
        end; {else}
        BorrarLista(borrar^.valores);
        dispose(borrar)
end;
```