ASIGNATURA	CURSO	GRUPO	CALIFICACIONES
PROGRAMACIÓN II		**1 2 3 4 5**	1 2 3 4 5A 5B
CONVOCATORIA			
PRIMERA OPORTUNIDAD (JUNIO)			

EJERCICIO 1

Dados los siguientes supuestos prácticos decidir: 1) cuál es la **MEJOR estructura de datos,** y 2) su **MEJOR implementación** para resolver el problema (para ser considerada correcta, la respuesta para AMBAS deberá estar **justificada):**

1. Instante, conocido servicio de mensajería rápida, desea premiar la fidelidad de sus clientes repartiendo mensualmente puntos en función del número de envíos que hagan con la compañía. Así tendrá caracterizado el tipo de cliente—vip, habitual o potencial—que podrá canjear dichos puntos por diferentes servicios. El sistema de información de la empresa necesita acceder eficientemente a los datos del cliente minimizando tiempo y recursos del proceso de búsqueda.

   SOLUCIÓN: AVL (clave =DNI) es la estructura realiza búsquedas de la manera más eficiente. Dinámico pues no tenemos datos que determinen el número de usuarios. El tipo de cliente es una propiedad que se almacena con el resto de datos del cliente.

2. Anticuarius es una nueva casa de subastas online. Pronto dispondrán de un portal web donde el usuario podrá consultar el horario en que se subastará cada pieza de arte así como los precios de salida. ¿Qué estructura única permitiría buscar la puja más barata, la más cara, o la n-ésima según el horario de la subasta?

   SOLUCIÓN: Una lista multienlazada con dos criterios de orden, cuantía y horario de la subasta, permite recorrer la estructura para encontrar la puja más barata, la más cara, la primera subasta,… La implementación es dinámica puesto que no hay datos en el enunciado que nos informen de ningún límite diario en cuanto al número de piezas a subasta. Otra opción sería una multilista, compuesta por una lista estática con horarios (24 diarios) de listas dinámicas (no hay límite) con las piezas (incluyen sus precios de salida). Así se puede obtener, para una franja horaria, la subasta más barata o la más cara. Además sabríamos cuál es la subasta n-ésima (la lista que cuelga del elemento n-ésimo de la lista de horarios). También podríamos recorrer toda la estructura para obtener la más barata de todas las franjas horarias.

3. Para la puja necesitará una estructura de datos reutilizable que almacene las sucesivas ofertas por la pieza subastada. Cada puja debe superar la última realizada. Al finalizar el tiempo, se contacta con el usuario que hubiera hecho la puja más alta. Si no respondiera con el siguiente, y así hasta encontrar una respuesta o dar la puja por desierta. Las restricciones de tiempo obligan a procesar no más de 300 solicitudes.

   SOLUCIÓN: Una pila permite apilar las sucesivas pujas (cuantía y datos del cliente), siempre que superen a la de la cima. Al finalizar el proceso, en la parte superior de la estructura encontraríamos la más alta, debajo la más alta de las restantes, y así sucesivamente. El proceso se limita a 300 solicitudes válidas, así que la pila sería estática. Otra posibilidad es usar una cola de prioridad implementada como un montículo Max. Se inserta un elemento en el montículo si supera el valor del elemento al frente de la cola. Al final, el mayor valor se encuentra en la raíz del árbol, y si se extrae, se reemplaza por el mayor de los restantes (siguiente en ser extraído). Las pujas salen en orden descendente de cuantía.

EJERCICIO 2

Contestar Verdadero o Falso y explicar **el porqué** a las siguientes preguntas (para ser considerada correcta, la respuesta deberá estar **justificada**):

1) Las precondiciones establecen qué controles debemos efectuar sobre los valores de los parámetros de entrada de una operación para que ésta tenga éxito.

   FALSO: Las precondiciones son como un contrato entre programador del tipo y usuario y establecen las condiciones que el usuario debe asegurar que se cumplan para que la operación tenga éxito. Así que no determinan qué controles hay que efectuar DENTRO de la operación (por parte del programador del tipo).

   VERDADERO: Las precondiciones son como un contrato entre programador del tipo y usuario y establecen las condiciones que el usuario debe asegurar que se cumplan para que la operación tenga éxito. Por tanto, indirectamente establecen qué controles hay que efectuar ANTES de llamar a la operación (por parte del usuario).

2) Una cola de prioridad puede implementarse a partir de una lista ordenada.

   CIERTO: Si usamos una única lista, los elementos se ordenan por su prioridad, se extrae sólo por un extremo y cada nuevo elemento se inserta al final de los de su misma prioridad.

3) Una lista doblemente enlazada permite recorrer la lista en función de dos criterios de ordenación distintos.

   FALSO: Sólo permite enlazar cada elemento con su siguiente y su anterior, siendo el mismo criterio de ordenación.

4) Para que un árbol binario sea considerado árbol AVL, la condición suficiente que debe cumplir todo nodo N es que la diferencia de altura entre el subárbol derecho y el izquierdo de ese nodo no sea superior, en valor absoluto, a una unidad.

   FALSO. Es condición necesaria pero no suficiente. Además debe cumplir que los elementos con clave menor que la contenida en N se encuentren ubicados en su subárbol izquierdo y los elementos con clave mayor que la contenida en N se encuentren ubicados en su subárbol derecho

EJERCICIO 3

Dado el siguiente código, indicar cuál(es) es(son) el(los) error(es) (de sintaxis, diseño o ejecución) que hemos cometido, en qué línea y porqué:

```
01  program ejercicio_3;
02   type
03     tPEntero = ^integer;
04   var
05     a, b: tPEntero;
06
07   procedure alCubo(var x: tPEntero);
08   begin
09     x^ := x^ * x^ * x^;
10     writeln(x^);
11     dispose(x);
12     x := NIL;
13   end;
14
15   begin
16     new(a);
17     a^ := 18;
18     b := a;
19     alCubo(a);
20     alCubo(b);
21   end.
```

SOLUCIÓN: *El procedimiento* `alCubo` *hace el* `dispose` *sobre el puntero que se le envía, destruyendo la variable dinámica a la que apunta. Como* `a` *y* `b` *apuntan a la misma variable, una vez que se hace la llamada a* `alCubo(a)` *la siguiente llamada* `alCubo(b)` *daría un error de ejecución al no existir ya la variable* `b^` *y tratar de acceder a ella.*

EJERCICIO 4

A) Dado el tipo abstracto de datos (TAD) tArbolBin que sirve para representar árboles binarios de enteros, del que sólo se conoce la parte siguiente de la interfaz:

```
type
     tDato= integer;
     tArbolBin= …;

function EsArbolVacio (a: tArbolBin): boolean;
procedure ArbolVacio (var a: tArbolBin);
function Raiz (a: tArbolBin): tDato;
procedure ConstruirArbol (Izq, Der: tArbolBin;
                          dato: tDato;
                          var A: tArbolBin);
function HijoIzquierdo (a: tArbolBin): tArbolBin;
function HijoDerecho (a: tArbolBin): tArbolBin;

Precondición común a las operaciones Raiz, HijoIzquierdo e HijoDerecho es que el
árbol no sea vacío.
Precondición para ConstruirArbol es que hay memoria suficiente para crear el árbol.
```

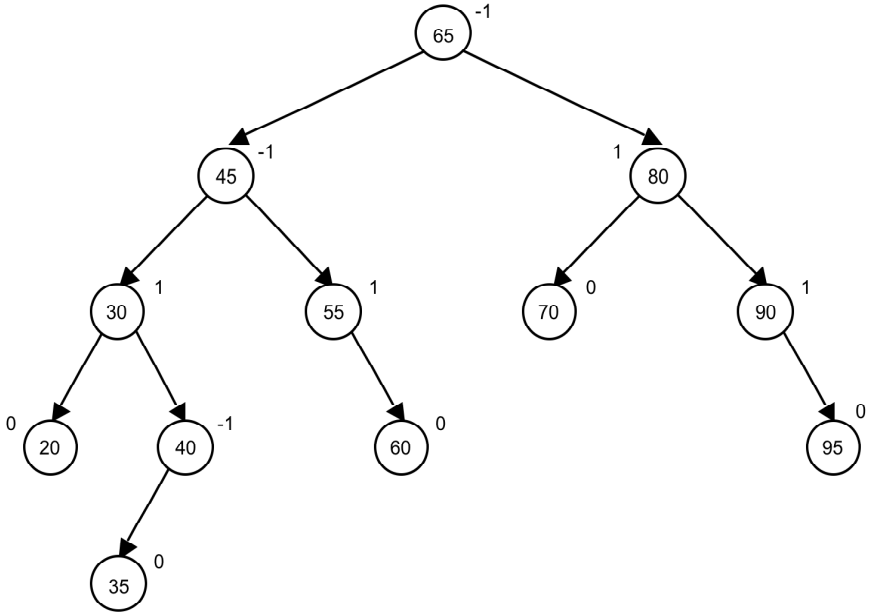
Se pide implementar, usando el TAD tArbolBin, la operación Espejo:

```
procedure Espejo(Arbol: tArbolBin; var Aespejo: tArbolBin);
```

que dado un árbol binario A creará otro que sea su imagen especular, es decir, su simétrico.

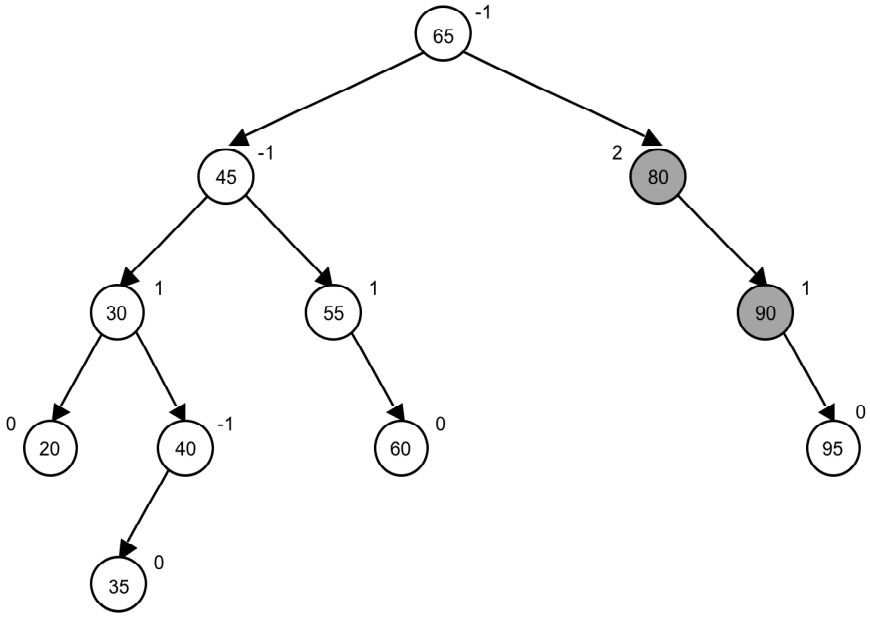
```
Procedure Espejo (Arbol: tArbol; var Aespejo: tArbol);
(*Precond: se supone memoria suficiente para hacer la operación*)
var HIesp, HDesp: tArbol;
begin
       ArbolVacio (Aespejo);
       if not EsArbolVacio (arbol)
       then begin
          Espejo (HijoIzquierdo(Arbol), HDesp);
          Espejo (HijoDerecho(Arbol), HIesp);
          ConstruirArbol (HIesp, HDesp, Raiz (arbol), Aespejo)
       end;
end;
```

B) Dado el siguiente árbol AVL

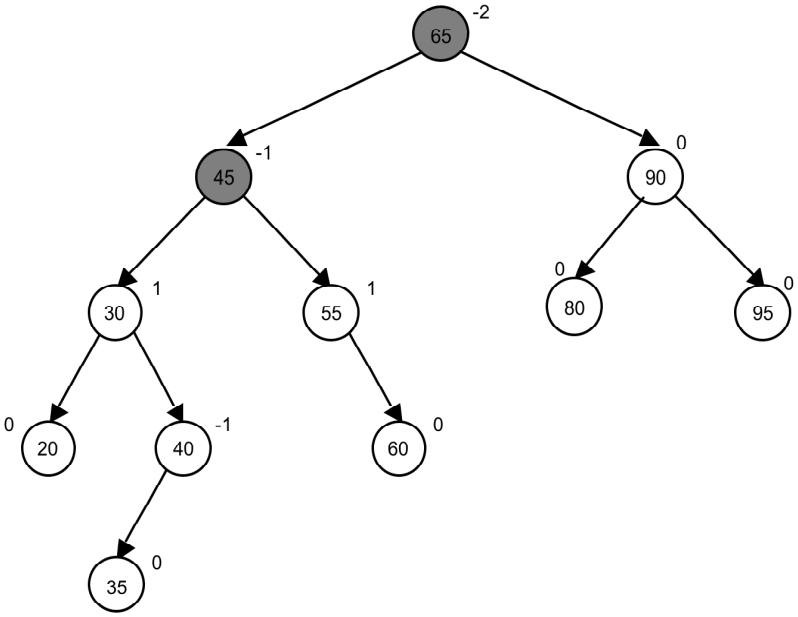


Se pide eliminar la clave 70 y mostrar las transformaciones que sufre el árbol: factores de equilibrio y, si hay que aplicar rotaciones, indicar su nombre, los nodos involucrados y el resultado de aplicarla.

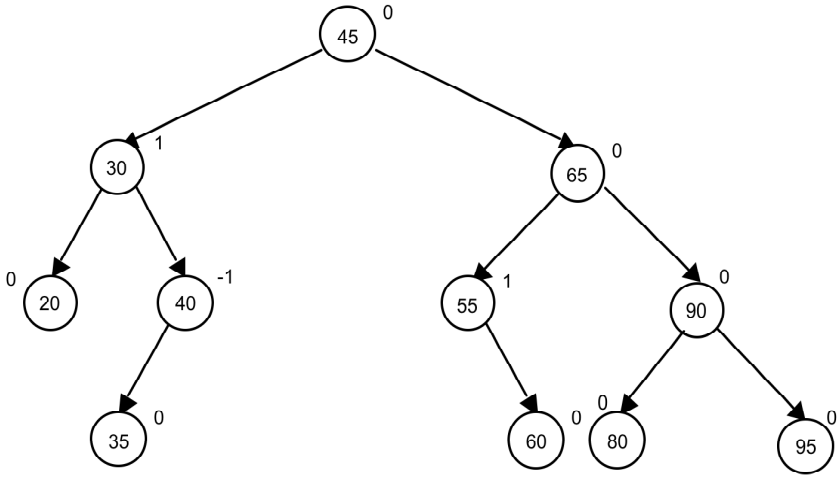
SOLUCIÓN: Eliminamos la clave 70



Se produce un desequilibrio en 80 y hay que aplicar rotación DD: nodos implicados 80-90

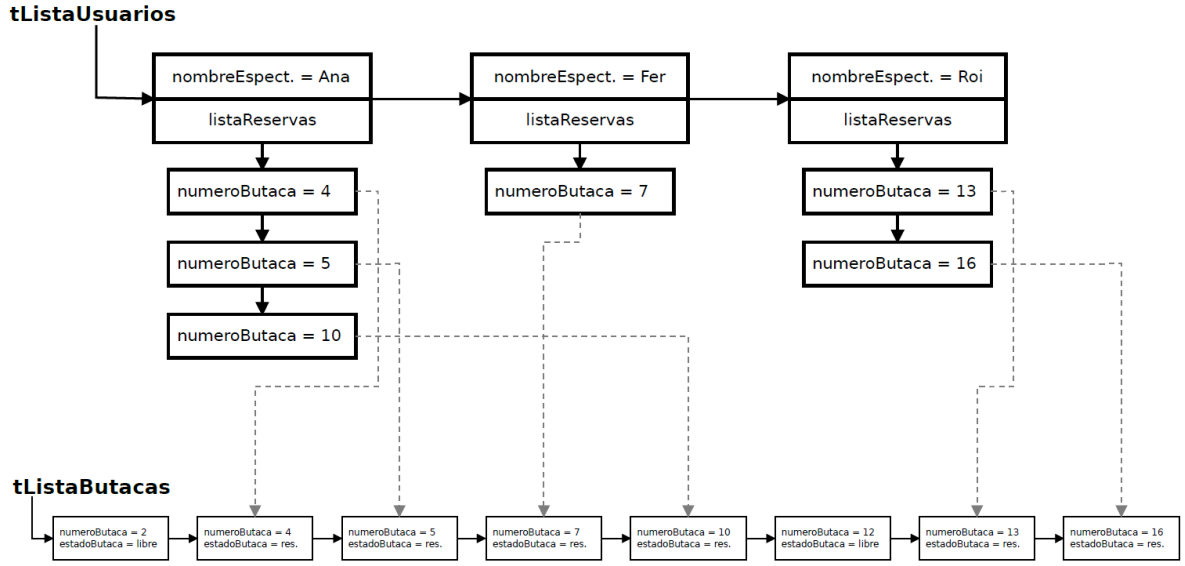


Al aplicar la rotación se desequilibra el nodo 65 y hay que aplicar rotación II. Nodos implicados: 65-45



EJERCICIO 5

En la práctica 2 se implementa un sistema para gestión de las butacas de una sala de espectáculos que consta de una lista de butacas, una lista de usuarios y, para cada cliente, una lista de reservas:



Partiendo del mismo problema, en este ejercicio se proponen una serie de modificaciones, tal como se explicará a continuación.

A)

Para gestionar las peticiones de los usuarios se utilizaba un TAD Cola en la que se almacenaban éstas por orden de llegada. El primer cambio consiste en implementar ésta como una **cola estática de 100 posiciones**. Se pide como ejercicio realizar la nueva definición de tipos de datos de la cola:

- tColaPeticiones — representa una cola de peticiones
- tPeticion — tipo de la petición recibida (char)
- tDatosC — datos de un elemento de la cola (correspondiente a una petición): está compuesto por los siguientes campos:
  - peticion: de tipo tPeticion
  - parametro1: de tipo string
  - parametro2: de tipo string
- NULOC — constante para representar posiciones nulas en la cola

Además, partiendo de la nueva definición de tipos de datos, se pide realizar la implementación de las siguientes operaciones:

- **EsColaVacia (tColaPeticiones) → Boolean**
  Determina si la cola está vacía.
- **InsertarDatoCola (tColaPeticiones, tDatosC) → tColaPeticiones, Boolean**
  Inserta un nuevo elemento (tDatosC) en la cola. Devuelve falso si no hay memoria suficiente para realizar la operación.
- **EliminarDatoCola (tColaPeticiones) → tColaPeticiones**
  Elimina el elemento que está en el frente de la cola.
  PreCD: la cola no está vacía.

```
const
   NULOC=0;
   MAX = 100;

type
   tPeticion=char; //tipos de operaciones V,R,C,P,S,L

   tDatoC=record
      peticion:tPeticion;
      parametro1:string;
      parametro2:string;
   end;

   tColaPeticiones= record
         datos    : array [1..MAX] of tDatoC;
         ini, fin : 0..MAX;
   end;

function sumarUno(i : integer):integer;
begin
    { sumarUno:= (i mod MAX)+1;}
   if (i=MAX) then
      sumarUno:=1
   else sumarUno := i+1
end; { sumarUno }

function  esColaVacia (Cola:tColaPeticiones):boolean;
begin
   esColaVacia := (C.ini = sumarUno(C.fin))
end; { esColaVacia }

function insertarDatoCola(var Cola:tColaPeticiones; dato:tDatoC):boolean;
begin
   if (C.ini=sumarUno(sumarUno(C.fin))) then
      insertarDatoCola := FALSE
   else begin
      insertarDatoCola := TRUE;
      C.fin := sumarUno(C.fin);
      C.datos[C.fin] := d; {tal vez copiar campo a campo}
   end
end; { insertarDatoCola }

procedure EliminarDatoCola(var Cola:tColaPeticiones);
begin
   C.ini := sumarUno(C.ini);
end; { eliminarDatoCola }
```

B)

El segundo cambio consiste en usar los TADs definidos en la práctica para crear un procedimiento que busque el bloque de butacas libres contiguas más grande (no necesariamente correlativas en número) en una lista de butacas no vacía:

```
procedure mayorBloqueContiguo(listaButacas: tListaButacas;
                              VAR bInicio:tNumButaca;
                              VAR tamano: integer);
```

Una vez terminado el procedimiento, si hay butacas libres `tamano` indicará el número de butacas del mayor bloque y `bInicio` la butaca inicial de dicho bloque. Si no hay butacas libres `tamano` será igual a 0 y el valor de `bInicio` será irrelevante.

Para el manejo de las estructuras se **utilizarán las operaciones disponibles en las interfaces del TAD Lista Butacas** que se adjunta en el ANEXO a este examen.

SOLUCIÓN:

```
procedure mayorBloqueContiguo(listaButacas: tListaButacas;
                              VAR bInicio:tNumButaca;
                              VAR tamano: integer);
var
  p : tPosB;
  bInicioAux : tNumButaca;
  tamanoAux : integer;
begin
  p := primeraB(listaButacas);
  bInicioAux := obtenerDatoB(p, listaButacas).numeroButaca;
  bInicip := bInicioAux;
  tamanoAux := 0;
  tamano := tamanoAux;
  repeat
    while (p<>NULOB) and
       (ObtenerDatoB(p,listaButacas).estadoButaca=libre) do begin
        tamanoAux := tamanoAux + 1;
        p := siguienteB(p, listaButacas)
    end;

    {si encontramos un trozo más grande que anterior, nos lo quedamos}
    if tamanoAux > tamano then begin
      tamano := tamanoAux;
      bInicio := bInicioAux;
    end

    if (p<>NULOB)  then
      if siguienteB(p, listaButacas)=NULOB then
        p := NULOB { evitar bucle infinito si última butaca ocupada}
      else begin
        p :=  siguienteB(p, listaButacas);
          bInicioAux := ObtenerDatoB(p, listaButacas).numeroButaca;
          tamanoAux := 0;
      end;
  until p=NULOB
end; { mayorBloqueContiguo }
```

Otra posible solución:

```
procedure mayorBloqueContiguo(lB: tListaButacas;
                              VAR bInicio:tNumButaca;
                              VAR tamano: integer);
{ Precd: lista de butacas no vacia }
var
    p: tPosB;
    tempB: tNumButaca;
    segmento: integer;
begin
    bInicio:= 0; tamano:= 0;
    p:= PrimeraB (lB);
    while p<> NULOB do begin
        if ObtenerDatoB (p, lB).estadoButaca= libre then begin
            {Anotar la primera libre}
            tempB:= ObtenerDatoB (p, lB).numeroButaca;
            segmento:= 1;
            {Bucle para recorrer mientras sea libre}
            while  (siguiente(p, lB) <> NULOB) and
                   (obtenerDatoB (siguiente(p, lB), lB).estadoButaca = libre)
            do begin
                segmento:= segmento + 1;
                p:= siguiente(p, listaButacas);
            end;
            {Si se encuentra un segmento más grande}
            if segmento > tamano then begin
                tamano:= segmento;
                bInicio:= tempB;
            end;
        end;
        p:= SiguienteB (p, lB);
    end;
end;
```