ASIGNATURA		CURSO	CALIFICACIÓN
ESTRUCTURA DE DATOS Y DE LA INFORMACIÓN		**2004 / 2005**	
TITULACIÓN	GRUPO	CONVOCATORIA	
		ORDINARIA – JUNIO	
APELLIDOS		NOMBRE	

EJERCICIO 1 (3 PUNTOS)

A) Dados los siguientes supuestos prácticos decidir cuál es la **MEJOR estructura de datos** y su **MEJOR implementación** (estática o dinámica) para resolver el problema, **justificando** la respuesta:

1. Un proveedor de servicios de descarga de ficheros que desea un sistema para la gestión del ancho de banda entre sus clientes que le permita atender antes a los que demanden ficheros más pequeños, manteniendo a los demás en espera. Los clientes deben estar previamente suscritos al servicio.

2. Queremos implementar un sistema de gestión de venta de entradas por internet. Las *N* entradas disponibles se asignan a las primeras *N* peticiones, las demás peticiones permanecen en espera. Transcurrido un tiempo limitado para hacer efectivo el pago, las entradas que han quedado libres se irán asignado a las personas en espera. ¿Qué estructura es adecuada para almacenar la lista de espera?

3. Los trenes actuales disponen en cada vagón de un sistema que monitoriza diversas variables como nivel de humo, temperatura, etc. Las locomotoras más potentes pueden transportar como máximo 10 vagones. Se desea una estructura que, teniendo almacenadas las variables monitorizadas de todos los vagones de los 30 trenes de la empresa, permita acceder, de forma eficiente, a las variables de un vagón a través de su número de serie.

B) Contestar **Verdadero o Falso** a las siguientes preguntas, **justificando** la respuesta:

1. Un montículo sirve para implementar una cola de prioridad

2. Los árboles binarios de búsqueda son las estructuras que siempre garantizan la mayor rapidez en las operaciones de búsqueda (realizan un menor número de comparaciones entre sus claves).

3. Una pila es la estructura más adecuada para invertir una cadena de caracteres.

4. En una lista ordenada la operación de búsqueda es siempre igual de eficiente (realiza el mismo número de comparaciones) en una implementación dinámica que en una estática.

C) Sea una implementación dinámica del TAD lista, explica los errores que presentaría el código siguiente respecto a su funcionamiento y/o el cumplimiento de la especificación de la operación (objetivo, entradas, salidas y precondiciones):

```
function Localiza_elemento (x: tInfo;  Lista: tLista): tPos;
{ Objetivo:  Busca el primer elemento de contenido x en la Lista
  Entrada:   x: contenido del elemento que buscamos
             Lista: acceso al principio de la lista
  Salida:    posición del elemento encontrado o nulo si el dato no esta en la lista
  PreCD:     Lista inicializada}

begin
   while (Lista ^.info<>x) and (Lista<> nulo) do
      Lista:= Lista^.sig;
   if (Lista^.info<>x)
      then  Localiza_elemento:=nulo
      else  Localiza_elemento:=Lista
end;
```

EJERCICIO 2 (4 PUNTOS)

En la democracia española se usa la ley D'hont para convertir los votos recibidos por un partido en una circunscripción electoral en sus escaños correspondientes. El algoritmo de la ley D'hont es como sigue:

1. Se determina el número de escaños (N) de la circunscripción electoral.
2. Se cuentan los votos obtenidos por cada partido en la circunscripción electoral.
3. Se dividen los votos obtenidos por los distintos partidos por 1, por 2, por 3, etc. hasta el número total de escaños de la circunscripción. Obtenemos así N colas. La cola *i* tendrá los votos de cada partido dividido por *i*.
4. La asignación de escaños se hace cogiendo los N valores mayores de las colas.

Un ejemplo clarificará el algoritmo. Supongamos que tenemos una circunscripción con 6 escaños en la que han obtenido votos los siguientes cuatro partidos: A (168 votos), B (104 votos), C (72 votos) y D (64 votos). Creamos entonces 6 colas que contienen los votos divididos (de forma entera) sucesivamente por 1, por 2, etc.

	Cola ÷1	Cola ÷2	Cola ÷3	Cola ÷4	Cola ÷5	Cola ÷6	
A	168	84	56	42	33	28	← Frentes
B	104	52	34	26	20	17	
C	72	36	24	18	14	12	
D	64	32	21	16	12	10	← Fines

La asignación de escaños se hará como sigue:

- El valor mayor en los frentes de las colas es 168 ⇒ 1 escaño para A y sacamos 168
- El valor mayor en los frentes de las colas es 104 ⇒ 1 escaño para B y sacamos 104
- El valor mayor en los frentes de las colas es 84 ⇒ 1 escaño para A y sacamos 84
- El valor mayor en los frentes de las colas es 72 ⇒ 1 escaño para C y sacamos 72
- El valor mayor en los frentes de las colas es 64 ⇒ 1 escaño para D y sacamos 64
- El valor mayor en los frentes de las colas es 56 ⇒ 1 escaño para A y sacamos 56

Resultado final: A (3 escaños), B (1 escaño), C (1 escaño) y D (1 escaño).

Se pide:

A) IMPLEMENTAR el TAD Cola. La especificación del TAD incluye los siguientes tipos y funciones en su interfaz:

tCola: Tipo que representa a la cola FIFO (First In, First Out)
tInfoCola: Tipo que representa la información de la cola, en este caso un registro con dos campos: nombre (un string para almacenar el nombre del partido) y votos (un integer que indica los votos).

ColaVacia() → tCola
Objetivo: Crea una cola vacía

InsertarCola(tCola, tInfoCola) → tCola, boolean
Objetivo: Inserta el valor en la Cola (situándose en el final de la misma)
Salidas: La cola con el valor insertado en su final y un valor boolean que es falso si no se ha podido realizar la inserción y cierto en caso contrario

EliminarCola(tCola) → tCola
Objetivo: Elimina el elemento del frente de la Cola
PreCD: La cola no está vacía

FrenteCola(tCola) → tInfoCola
Objetivo: Devuelve el elemento del frente de la cola pero sin eliminarlo de la misma
PreCD: La cola no está vacía

EsColaVacia(tCola) → boolean
Objetivo: Determina si una cola es vacía o no

VaciarCola(tCola) → tCola
Objetivo: Elimina todos los elementos de la cola dejándola vacía

B) Nos facilitan una implementación del TAD lista ordenada de partidos del cual sólo conocemos los siguientes tipos y funciones presentes en su interfaz:

tLista: Tipo que representa a la lista ordenada de partidos
tPos: Tipo que representa a un apuntador a una posición de la lista
tInfo: Tipo que representa la información de la lista, en este caso un registro con tres campos: nombre (un string para almacenar el nombre de un partido - criterio por el cual ordenamos), votos (un integer para almacenar los votos) y escanos (un integer para almacenar el número de escaños).

NOTA: Se considera que la siguiente precondición se aplica a todas las siguientes funciones: la lista no está vacía y si se pasa una posición por parámetro dicha posición será válida en la lista

Primero (tLista) → tPos
Objetivo: Devuelve la posición del primer elemento de la lista

Ultimo (tLista) → tPos
Objetivo: Devuelve la posición del último elemento de la lista

Siguiente (tLista, tPos) → tPos
Objetivo: Devuelve la posición del elemento siguiente al indicado

Anterior (tLista, tPos) → tPos
Objetivo: Devuelve la posición del elemento anterior al indicado

DevolverContenido (tLista, tPos) → tInfo
Objetivo: Devuelve el contenido de una posición de la lista

LocalizarPartido (tLista, string) → tPos
Objetivo: Localiza en la lista la posición del partido cuyo nombre coincide con el string pasado por parámetro
Salidas: La posición del valor en la lista o nulo si no existe

ModificarEscanos (tLista, tPos, integer) → tLista
Objetivo: Actualiza el valor de los escaños de un elemento de la lista con el integer pasado por parámetro

Basándote en los TADs cola y lista ordenada **IMPLEMENTA** la siguiente función:

CalculaEscanos(tLista, integer) -> tLista
Objetivo: Dada una lista de partidos con sus respectivos votos, calcula los escaños que les corresponderían
Entradas: Una lista ordenada de partidos con sus respectivos votos y un integer que indica el número de escaños de la circunscripción
Salidas: La misma lista pero con el campo que representa a los escaños actualizado según la ley D'hont
PreCD: Se supone que la lista **no es vacía** y está **correctamente inicializada** con los nombres de los partidos, sus respectivos votos y el valor de los escaños puesto a cero. **El número de escaños máximo de una circunscripción será de 10. El número máximo de partidos no está determinado**

EJERCICIO 3 (3 PUNTOS)

A) Pretendemos contar los elementos que ocupan las posiciones impares entre los índices "ini" y "fin" de un array "a", tomando siempre como precondición que ambos índices se encuentran en el dominio de los definidos para el array "a". Para ello hemos creado la siguiente función sumaPosImpares, pero no está bien definida. ¿Qué terrible error hemos cometido en esta función recursiva? ¿Cómo sería la versión correcta de sumaPosImpares?

```
function sumaPosImpares(a : tArray; ini,fin:integer):integer;
begin
  if (ini mod 2) <> 0 then
    sumaPosImpares := a[ini] + sumaPosImpares (a,ini+2,fin)
  else
    sumaPosImpares := sumaPosImpares(a,ini+1,fin);
end; { sumaPosImpares }
```

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

```
type
    tArbolBin = ...

function EsArbolVacio (a:tArbolBin):boolean;
function Raiz (a:tArbolBin):integer;
function RamaIzda (a:tArbolBin):tArbolBin;
function RamaDcha (a:tArbolBin):tArbolBin;
```

se pide definir una función que obtenga el nivel de la hoja menos profunda en un árbol **no vacío**.

Soluciones

EJERCICIO 1 (3 PUNTOS)

1A)

1. SOLUCIÓN: montículo min estático
2. SOLUCIÓN: Cola dinámica.
3. SOLUCIÓN: Lista ordenada por número de serie y estática (tamaño 30x10) o AVL estático.

1B)

1. SOLUCIÓN: Verdadero
2. SOLUCIÓN: Falso
3. SOLUCIÓN: Verdadero
4. SOLUCIÓN: Falso.

1C)

SOLUCIÓN: fallaría si el elemento no está en la lista (incluye el caso de lista vacía)

EJERCICIO 2 (4 PUNTOS)

A)

```
unit Cola;

interface

const
  NULO_C = nil;
type
  tInfoCola = record
                partido: string;
                votos: integer;
              end;
  tPosCola  = ^tNodoCola;
  tNodoCola = record
               info: tInfoCola;
               sig: tPosCola;
              end;
  tCola     = record
                Frente: tPosCola;
                Fin: tPosCola;
              end;

function ColaVacia:tCola;
function InsertarCola (var C:tCola;Valor:tInfoCola):boolean;
function EliminarCola (var C:tCola): tInfoCola;
function FrenteCola (C:tCola): tInfoCola;
function EsColaVacia (C: tCola): boolean;
procedure VaciarCola (var C:tCola);
```

```
implementation

function ColaVacia:tCola;
begin
  ColaVacia.Frente:=nil;
  ColaVacia.Fin:=nil;
end;



function InsertarCola (var C:tCola;Valor:tInfoCola):boolean;
var
  Nuevo: tPosCola;
begin  // Insertamos por el Fin de la cola
  new(Nuevo);
  if (Nuevo<>NULO_C) then
    begin
      Nuevo^.info:=Valor;
      Nuevo^.sig:=nil;
      if EsColaVacia(C) then
        C.Frente:=Nuevo // Si la cola está vacia tenemos que actualizar C.Frente
      else
        C.Fin^.sig:=Nuevo; // Si no está vacía el último tiene que apuntar al nuevo
      C.Fin:=Nuevo;     // C.Fin se actualiza siempre
      InsertarCola:=true;
    end
  else
    InsertarCola:=false;
end;



function EliminarCola (var C:tCola): tInfoCola;
var
  aux:tPosCola;
begin
  EliminarCola:=C.Frente^.info;
  aux:=C.Frente;
  C.Frente:=C.Frente^.sig;
  dispose(aux);
  if C.Frente=nil then // El único caso en el que hay que modificar C.Fin
    C.Fin:=nil         // es cuando la lista queda vacía
end;



function FrenteCola (C:tCola): tInfoCola;
begin
  FrenteCola:=C.Frente^.info;
end;



function EsColaVacia (C: tCola): boolean;
begin
  EsColaVacia:=(C.Frente=nil) // También valdría con C.Fin=nil
end;



procedure VaciarCola(var C:tCola);
var
  aux: tPosCola;
begin
  while (C.Frente <> nil) do
    begin
      aux:=C.Frente;
      C.Frente:=C.Frente^.sig;
      dispose(aux);
    end;
  C.Fin:=nil;
end;

end.
```

B)

```
procedure CalculaEscanos(var L:tLista; n:integer);
var
  colas: array[1..10] of tCola;
  i, maxVotos, maxCola, escanos: integer;
  maxPartido: string;
  auxCola: tInfoCola;
  aux: tInfo;
  P: tPos;
begin
  // Creamos las colas
  for i:=1 to n do
    colas[i]:=ColaVacia;

  // Rellenamos las colas
  P:=Primero(L);
  repeat
    aux:=DevolverContenido(L,P);
    auxCola.partido:=aux.partido;
    for i:=1 to n do
      begin
        auxCola.votos:=aux.votos div i;
        InsertarCola(colas[i], auxCola);
      end;
    P:=Siguiente(L,P);
  until P=NULO;

  // Asignamos los escaños
  escanos := n;
  repeat
    maxVotos:=0;

    // Buscamos la cola con el mayor valor
    for i:=1 to n do
      if not EsColaVacia(colas[i]) then // Es posible que una cola se agote
        begin
          auxCola:=FrenteCola(colas[i]);
          if auxCola.votos > maxVotos then
            begin
              maxVotos:=auxCola.votos;
              maxPartido:=auxCola.partido;
              maxCola:=i;
            end;
        end;

    // Extraemos de la cola y asignamos el escaño
    ELiminarCola(colas[maxCola]);
    P:=LocalizarElemento(L, maxPartido);
    ModificarEscanos(L, P, DevolverEscanos (L,P) + 1);
    escanos := escanos - 1;
  until escanos=0;
end;
```

EJERCICIO 3 (3 PUNTOS)

3A)

SOLUCION:

añadir caso base

```
function sumaPosImpares(a : tArray; ini,fin:integer):integer;
begin
  if ini>fin then
     sumaPosImpares := 0
  else if (ini mod 2) <> 0 then
         sumaPosImpares := a[ini] + sumaPosImpares (a,ini+2,fin)
       else
         sumaPosImpares := sumaPosImpares(a,ini+1,fin);
end; { sumaPosImpares }
```

3B)

SOLUCIÓN:

```
function hojaMenosProf(A:tArbolBin):integer;
begin
  if esArbolVacio(ramaIzda(A)) AND esArbolVacio(ramaDcha(A)) then
     hojaMenosProf := 1 (o 0 si se considera el nivel raiz como 0)
  else if esArbolVacio(ramaIzda(A)) then
         hojaMenosProf := hojaMenosProf(ramaDcha(A)) + 1
       else if esArbolVacio(ramaDcha(A)) then
              hojaMenosProf := hojaMenosProf(ramaIzda(A)) + 1
            else hojaMenosProf := 1 + min(hojaMenosProf(ramaIzda(A))
                                          hojaMenosProf(ramaDcha(A)));
end;
```

EJERCICIO DE PRÁCTICAS (DURACIÓN 30 MINUTOS)

A realizar sólo por aquellos alumnos de primera matrícula que no hayan asistido a clase de prácticas.

Implementa las siguientes funciones del TAD lista ordenada de partidos descrito en el ejercicio 2, supón que la lista ordenada esta implementada como una lista dinámica con enlaces simples:

LocalizarPartido (tLista, string) → tPos
Objetivo: Localiza en la lista la posición del partido cuyo nombre coincide con el string pasado por parámetro
Salidas: La posición del valor en la lista o nulo si no existe
PreCD: La lista no está vacía

function InsertarOrdenado (tLista, tInfo) → tLista, boolean
Objetivo: Inserta de forma ordenada (por nombre de partido) un registro tInfo (nombre, votos y escaños) en una lista
Entradas: Una lista ordenada y un registro tInfo
Salidas: La lista con la información insertada de forma ordenada y un valor boolean que es cierto si se ha podido insertar correctamente y falso en caso contrario
PreCD: tInfo contiene un nombre válido de partido

Soluciones Ejercicio de Prácticas

```
function LocalizarPartido (L: tLista; partido: string): tPos;
begin
  while (L^.info.partido<partido) and (L^.sig<>NULO) do // Buscamos hasta que info >= L
    L:=L^.sig;                                          // o hasta que sig sea NULO
  if L^.info.partido=partido   // Comprobamos por qué nos hemos salido del bucle
    then LocalizarElemento:=L
    else LocalizarElemento:=NULO
end;
```

```
function InsertarOrdenado (var L: tLista; Valor: tInfo):boolean;
var
  aux, ant, Nuevo:tPos;
begin
  new(Nuevo);
  if (Nuevo=NULO) then
    InsertarOrdenado:=false
  else
    begin
      InsertarOrdenado:=true;
      Nuevo^.info:=Valor;
      // Comprobamos si hay que insertar en la primera posición
      if (L=NULO) or (Valor.partido<=L^.info.partido) then // Funciona con cortocircuito
        begin
          Nuevo^.sig:=L;
          L:=Nuevo
        end
      else    // recorremos la lista con dos punteros. aux marca una posición y
        begin // ant marca la posicion anterior
          ant:=L;
          aux:=L;
          while (Valor.partido>aux^.info.partido) and (aux^.sig<>nil) do
            begin // nos salimos al encontrar un valor >= o al llegar al final
              ant:=aux;
              aux:=aux^.sig
            end;
          if (Valor.partido<=aux^.info.partido) then // comprobamos por que salimos
            begin // insertamos por el medio
              ant^.sig:=Nuevo;
              Nuevo^.sig:=aux;
            end
         else
            begin // hay que insertar al final
              aux^.sig:=Nuevo;
              Nuevo^.sig:=NULO;
            end
        end
    end;
end;
```