Apellidos:
Nombre:
Convocatoria:
DNI:

Examen TAD/PED diciembre 2004
Modalidad 0

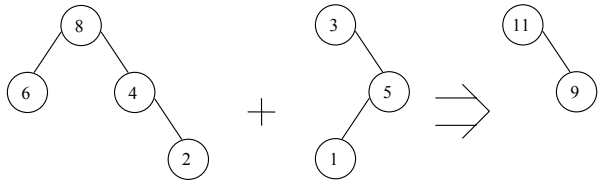
Normas:
- La entrega del test no corre convocatoria.
- Tiempo para efectuar el test: **20 minutos**.
- Una pregunta mal contestada elimina una correcta.
- Las soluciones al examen se dejarán en el campus virtual.
- **Una vez empezado el examen no se puede salir del aula hasta finalizarlo. A continuación comenzará el siguiente ejercicio.**
- **El test vale un 40% de la nota de teoría.**
- En la **hoja de contestaciones** el verdadero se corresponderá con la **A**, y el falso con la **B**.

	V	**F**		
En un Tipo Abstracto de Datos la manipulación de los objetos o valores de un tipo, sólo depende del comportamiento descrito en su especificación y es independiente de su implementación.	❑	❑	V	1.
La complejidad temporal (en su peor caso) de la operación de insertar un elemento en una cola circular enlazada que no admite elementos repetidos es O(n).	❑	❑	V	2.
La complejidad temporal de la operación *apilar* en una pila es independiente de su implementación.	❑	❑	F	3.
Sea el TIPO *arbin* definido en clase. La semántica de la operación *nodos* es la siguiente: *var i,d:arbin; x:item;* *nodos(crear_arbin())=0* *nodos(enraizar(i,x,d))=nodos(i)+nodos(d)*	❑	❑	F	4.
En un árbol AVL, el factor de equilibrio de un nodo se define como el valor absoluto de la altura del subárbol de la derecha menos la altura del subárbol de la izquierda.	❑	❑	F	5.
El mínimo número de elementos que se puede almacenar en un árbol 2-3 de altura *h* coincide con el número de elementos que hay en un árbol binario lleno de altura *h*.	❑	❑	V	6.
Al insertar un único ítem en un árbol 2-3-4 puede ocurrir que haya que realizar dos operaciones: DIVIDERAIZ (p) y DIVIDEHIJODE2 (p, q).	❑	❑	V	7.
La inserción de una etiqueta en un árbol rojo-negro se efectuará creando un hijo de color rojo.	❑	❑	V	8.
Un árbol binario de búsqueda con altura 7 y 127 nodos es un árbol B con m=3	❑	❑	V	9.
En la operación de búsqueda de un elemento en un conjunto ordenado representado mediante un vector de elementos ordenado (representación no enlazada de una lista), se puede utilizar el algoritmo de búsqueda binaria.	❑	❑	V	10.
El factor de carga de la dispersión abierta siempre está entre 0 y 1.	❑	❑	F	11.
En un montículo doble, un elemento "i" del montículo mínimo tiene como máximo un elemento simétrico "j" del montículo máximo.	❑	❑	V	12.
Un árbol binario de búsqueda cumple las propiedades de un árbol Leftist.	❑	❑	F	13.
La siguiente secuencia de nodos de un grafo es un ciclo: 1,2,3,1.	❑	❑	V	14.

Examen TAD/PED diciembre 2004

Normas:
- Tiempo para efectuar el ejercicio: **2 horas**
- En la cabecera de cada hoja **Y EN ESTE ORDEN** hay que poner: ***Apellidos, Nombre***. Cada pregunta se escribirá en folios diferentes.
- Se dispone de 20 minutos para abandonar el examen sin que corra convocatoria.
- Las soluciones al examen se dejarán en el campus virtual.
- Se puede escribir el examen con lápiz, siempre que sea legible
- **Todas las preguntas tienen el mismo valor.** Este examen vale el 60% de la nota de teoría.
- **Publicación de notas de exámenes**: 13 de diciembre. **Fecha de revisión de exámenes:** El lugar, fecha y hora se publicará en el campus virtual.
- **Los alumnos que estén en 5ª o 6ª convocatoria deben indicarlo en la cabecera de todas las hojas**

1. Utilizando exclusivamente las operaciones constructoras generadoras de los árboles binarios, definir la sintaxis y la semántica de la operación *sumar* que crea un árbol binario resultado de sumar las etiquetas de los nodos simétricos de 2 árboles binarios dados cuyas etiquetas son números naturales.
Nota: Si un nodo de un árbol no tiene nodo simétrico en el otro, en el árbol binario resultado no aparecerá dicho nodo.



2. **PARTE A.** El siguiente fragmento de código corresponde a la definición de la clase *TDir*, que representa a una DIRECCIÓN INTERNET y está básicamente compuesta de un NOMBRE y una DIRECCIÓN IP, siendo ambos datos una cadena de caracteres. El código pertenece a la definición de prototipos (*TDir.h*). Se pide:

a) Indicar en qué líneas ves INCORRECCIONES desde el punto de vista de una CORRECTA PROGRAMACIÓN C++ . Pueden ser ERRORES LÉXICO-SINTÁCTICOS (producen ERROR DE COMPILACIÓN), o bien INCORRECCIONES desde el punto de vista de una correcta programación (no necesariamente producirían ERROR DE COMPILACIÓN).

b) Indicar, para cada error, la palabra "COMPILA" (si NO producen ERROR DE COMPILACIÓN) o "NO COMPILA" (si producen ERROR DE COMPILACIÓN)

Ejemplo de contestación : "**LINEA x**: el error es <*bla bla bla* > . COMPILA"

```
1   #ifndef _TDIR_
2   #define _TDIR_
3
4   #include <iostream>
5   #include <cstring>
6   using namespace std;
7
8   class TDir
9   {
10  public:
11        TDir();
12        TDir(char *Nombre,char *IP);
13        TDir(const TDir );
14        ~TDir();
15
16  private:
17        char* Nombre;
18        char* IP;
19  friend ostream & operator<<(ostream &, Tdir &);
20
21  }.
22
23  # end
```

PARTE B. El siguiente fragmento de código corresponde a la clase TDir , la misma que la de la parte A. El código pertenece a la programación de los métodos (TDir.cc). Se pide:

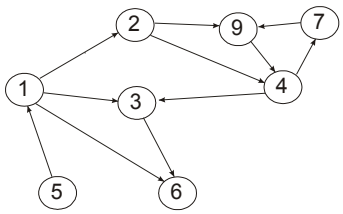
a) Indicar en qué líneas ves INCORRECCIONES desde el punto de vista de una CORRECTA PROGRAMACIÓN C++ . Pueden ser ERRORES LÉXICO-SINTÁCTICOS (producen ERROR DE COMPILACIÓN), o bien INCORRECCIONES desde el punto de vista de una correcta programación de la clase (no necesariamente producirían ERROR DE COMPILACIÓN). **NOTA:** si el error se debe a una OMISIÓN (es decir, FALTA código por poner), se debe señalar la LÍNEA VACÍA donde debería aparecer este código. Para ello, se han dejado líneas vacías en el código propuesto.

b) Indicar, para cada error, la palabra "COMPILA" (si NO producen ERROR DE COMPILACIÓN) o "NO COMPILA" (si producen ERROR DE COMPILACIÓN)

Ejemplo de contestación: "**LINEA x**: el error es <*bla bla bla* > . COMPILA"

```
1
2   #include <iostream>
3
4   #include <cstring>
5
6   TDir::TDir() {
7
8          Nombre=NULL;
9
10         IP=NULL;
11
12  };
13
14  TDir::TDir(const TDir &a){
15
16         Nombre=new char[strlen(a.Nombre)+1];
17
18         strcpy(Nombre,a.Nombre);
19
20         if(a->IP != NULL){
21
22         {
23
24                IP=new char[strlen(a.IP+1)];
25
26                strcpy(IP,a.IP);
27
28         }
29
30  }
31
32  bool TDir::~TDir(){
33
34         if(Nombre != NULL){
35
36                Nombre=NULL;
37
38         }
39
40         if(IP != NULL){
41
42                delete IP;
43
44                IP=NULL;
45
46         }
47
48  }
```

3. Sea el siguiente grafo dirigido:



a) Realiza el recorrido DFS(1), con la adyacencia de salida ordenada de menor a mayor, y obtén el árbol extendido en profundidad.
b) Etiqueta los arcos.
c) ¿Este grafo tiene ciclos? ¿Por qué?
d) ¿Es un grafo fuertemente conexo? Justifica tu respuesta.

4. Calcular las cotas superiores de complejidad temporal en su caso peor, para un montículo simple máximo y un montículo doble, para las siguientes operaciones:

Insertar(elemento), Borrar(elemento), int NúmeroDeElementos(), ListarElementosPorNiveles(), int DevolverMáximo(), elemento DevolverMínimo()

Para ello suponed que la implementación interna de los montículos es como sigue:

class Montículo {
*int * vector;*
int numeroElementos; ... };

NOTA: Hay que explicar cada complejidad con un máximo de 3 líneas. Si la explicación no es correcta, no se valorará positivamente.

Examen TAD/PED diciembre 2004. Soluciones

1)

sumar: arbin, arbin → arbin

var i,l,d,r:arbin; x,y:natural;

sumar(crear_arbin(),crear_arbin()) = crear_arbin()
sumar(crear_arbin(),d) = crear_arbin()
sumar(i,crear_arbin()) = crear_arbin()
sumar(enraizar(i,x,d),enraizar(l,y,r)) = enraizar(sumar(i,l),x+y ,sumar(d,r))

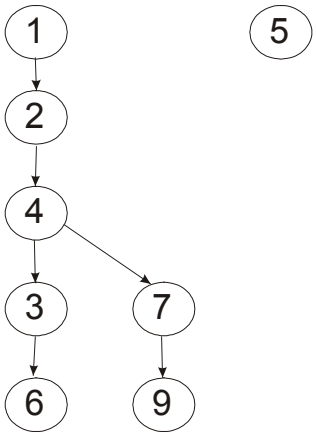
2)

```
1   #ifndef _TDIR_
2   #define _TDIR_
3
4   #include <iostream>
5   #include <cstring>
6   using namespace str;
7
8   class TDir
9   {
10  public:
11          TDir();
12          TDir(char *Nombre,char *IP);
13          TDir(const TDir );        // FALTA "&"   COMPILA
14          ~TDir();
15
16  private:
17          char* Nombre;
18          char* IP;
19  friend ostream & operator<<(ostream &, Tdir &); // mayúscula : "TDir") NO COMPILA
20
21  }.    // cambiar "." por ";"   NO COMPILA
22
23  # end // cambiar "end" por "endif" NO COMPILA
```

```
1                                       //FALTA #include "TDir.h" NO COMPILA
2   #include <iostream>
3
4   #include <cstring>
5
6   TDir::TDir() {
7
8           Nombre=NULL;
9
10          IP=NULL;
11
12  };
13
14  TDir::TDir(const TDir &a){
15                                      // FALTA if(a.Nombre!=NULL){ COMPILA
16          Nombre=new char[strlen(a.Nombre)+1];
17
18          strcpy(Nombre,a.Nombre);
19
20          if(a->IP != NULL){ // cambiar "->" por "."NO COMPILA
21
22          {
23
24                  IP=new char[strlen(a.IP+1)];//es "new char[strlen(a.IP)+1]" NO COMPILA
25
26                  strcpy(IP,a.IP);
27
28          }
29
30  }
31
32  bool TDir::~TDir(){         // quitar "bool"   NO COMPILA
33
34          if(Nombre != NULL){
35                                  // FALTA "delete Nombre" COMPILA
36                  Nombre=NULL;
37
38          }
39
40          if(IP != NULL){
41
42                  delete IP;
43
44                  IP=NULL;
45
46          }
47
48  }
```

3)

a)



b)

(1,2) = Arbol.
(2,4) = Arbol.
(4,3) = Arbol
(4,7) = Arbol
(3,6) = Arbol
(7,9) = Arbol
(1,3) = Avance.
(1,6) = Avance.
(2,9) = Avance.
(5,1) = Cruce.
(9,4) = Retroceso.

c)

Si tiene ciclos porque existe un arco de retroceso. (9,4)

d)

No es un grafo fuertemente conexo.
Para que no sea un grafo fuertemente conexo basta con que existan mas de una componente fuertemente conexa. La componente fuertemente conexa formada por el nodo 6 ya es una componente fuertemente conexa y como no están incluidos todos los nodos del grafo, quiere decir que hay más de una.

4)

	Montículo Simple Máximo	**Montículo Doble**
Insertar(elemento)	$O(\log_2 n)$ habría que realizar un recorrido ascendente: la altura de un árbol binario completo	$O(\log_2 n)$ habría que realizar recorridos ascendentes: la altura de un árbol binario completo
Borrar(elemento)	$O(\log_2 n)$ habría que realizar un recorrido descendente: la altura de un árbol binario completo	$O(\log_2 n)$ habría que realizar recorridos descendentes: la altura de un árbol binario completo
NúmeroDeElementos	O(1) se accede al campo *numeroElementos*	O(1) se accede al campo *numeroElementos*
ListarElementosPorNiveles	O(n) habría que recorrer todo el vector hasta alcanzar los *n* elementos	O(n) habría que recorrer todo el vector hasta alcanzar los *n* elementos
DevolverMáximo	O(1) se accedería en la raíz del montículo que está en la primera posición del *vector*	O(1) se accedería en la raíz del montículo máximo que está en la segunda posición del *vector*
DevolverMínimo	O(n) siguiendo el algoritmo de recorrer todos los elementos del vector almacenando en todo momento el mínimo elemento encontrado	O(1) se accedería en la raíz del montículo mínimo que está en la primera posición del *vector*