



QuesoViejo\_  
[www.wuolah.com/student/QuesoViejo\\_](http://www.wuolah.com/student/QuesoViejo_)

 236784

## **Practicas-Arboles-Binarios-Resueltas.pdf** EJERCICIOS RESUELTOS PRÁCTICAS



**2º Estructuras de Datos**



**Grado en Ingeniería Informática**



**Escuela Técnica Superior de Ingenierías Informática y de  
Telecomunicación  
Universidad de Granada**

# Exámenes, preguntas, apuntes.

12:48

WUOLAH

Join the student revolution.

MULTI

Conéctate dónde y cómo prefieras.

Guarda tus apuntes en un lugar seguro y ordenado, y accede a ellos desde tu pc, móvil o tablet.

Acceder

Registrarse

GET IT ON  
Google Play

Download on the  
App Store

# Estructuras de Datos no Lineales

## Práctica 1

### Problemas de árboles binarios I

#### TRABAJO PREVIO

1. **Antes de asistir a la primera sesión de prácticas es obligatorio tener implementado y probado el TAD *árbol binario* con las tres representaciones estudiadas, vectorial, vector de posiciones relativas y enlazada.**
2. **También es obligatorio antes de asistir a la sesión de prácticas:**
  - 2.1. Imprimir copia de este enunciado.
  - 2.2. Lectura profunda del mismo.
  - 2.3. Reflexión sobre el contenido de la práctica y generación de la lista de dudas asociada a dicha práctica y a los problemas que la componen.
  - 2.4. **Esbozo serio de solución** de los problemas en papel (al menos de los que se hayan entendido).

#### PASOS A SEGUIR

1. Escribir módulos que contengan las implementaciones de los subprogramas demandados en cada problema.
2. Para cada uno de los problemas escribir un programa de prueba, independiente de la representación del TAD elegida, donde se realicen las llamadas a los subprogramas del paso anterior, comprobando el resultado de salida para una batería suficientemente amplia de casos de prueba.

#### ENTRADA Y SALIDA DE ÁRBOLES BINARIOS

Se proporciona la cabecera `abin_E-S.h` que incluye cuatro funciones genéricas para la lectura y escritura de árboles binarios a través de flujos de entrada y salida:

`template <typename T> void rellenarAbin (Abin<T> & A, const T & fin)`

Pre: `A` está vacío.

Post: Rellena el árbol `A` con la estructura y elementos leídos en preorden de la entrada estándar, usando `fin` como elemento especial para introducir nodos nulos.

`template <typename T> void rellenarAbin (istream & is, Abin<T> & A)`

Pre: `A` está vacío.

Post: Extrae los nodos de `A` del flujo de entrada `is`, que contendrá el elemento especial que denota un nodo nulo seguido de los elementos en preorden, incluyendo los correspondientes a nodos nulos.

*template <typename T> void imprimirAbin (const Abin<T>& A)*

Post: Muestra los nodos de A en la salida estándar.

*template <typename T>*

*void imprimirAbin (ostream& os, const Abin<T>& A, const T& fin)*

Post: Inserta en el flujo de salida *os* los nodos de *A* en preorder, precedidos del elemento especial usado para denotar un nodo nulo.

### Ejemplo:

```
#include <iostream>
#include <fstream>
#include "abin.h"
#include "abin_E-S.h"

using namespace std;

typedef char tElto;
const tElto fin = '#'; // fin de lectura

int main ()
{
    Abin<tElto> A, B;

    cout << "*** Lectura del árbol binario A ***\n";
    llenarAbin(A, fin); // desde std::cin

    ofstream fs("abin.dat"); // abrir fichero de salida
    imprimirAbin(fs, A, fin); // en fichero
    fs.close();
    cout << "\n*** Árbol A guardado en fichero abin.dat ***\n";

    cout << "\n*** Lectura de árbol binario B de abin.dat ***\n";
    ifstream fe("abin.dat"); // abrir fichero de entrada
    llenarAbin(fe, B); // desde fichero
    fe.close();

    cout << "\n*** Mostrar árbol binario B ***\n";
    imprimirAbin(B); // en std::cout
}
```

### Salida del programa:

```
*** Lectura del árbol binario A ***
Raiz (Fin = #): a
Hijo izqdo. de a (Fin = #): b
Hijo izqdo. de b (Fin = #): c
Hijo izqdo. de c (Fin = #): d
Hijo izqdo. de d (Fin = #): #
Hijo drcho. de d (Fin = #): #
Hijo drcho. de c (Fin = #): #
Hijo drcho. de b (Fin = #): e
Hijo izqdo. de e (Fin = #): #
Hijo drcho. de e (Fin = #): f
Hijo izqdo. de f (Fin = #): #
```

```

Hijo drcho. de f (Fin = #): #
Hijo drcho. de a (Fin = #): g
Hijo izqdo. de g (Fin = #): h
Hijo izqdo. de h (Fin = #): #
Hijo drcho. de h (Fin = #): #
Hijo drcho. de g (Fin = #): i
Hijo izqdo. de i (Fin = #): j
Hijo izqdo. de j (Fin = #): #
Hijo drcho. de j (Fin = #): #
Hijo drcho. de i (Fin = #): k
Hijo izqdo. de k (Fin = #): l
Hijo izqdo. de l (Fin = #): #
Hijo drcho. de l (Fin = #): #
Hijo drcho. de k (Fin = #): #

```

\*\*\* Árbol A guardado en fichero abin.dat \*\*\*

\*\*\* Lectura de árbol binario B de abin.dat \*\*\*

```

*** Mostrar árbol binario B ***
Raíz del árbol: a
Hijo izqdo de a: b
Hijo izqdo de b: c
Hijo izqdo de c: d
Hijo derecho de b: e
Hijo derecho de e: f
Hijo derecho de a: g
Hijo izqdo de g: h
Hijo derecho de g: i
Hijo izqdo de i: j
Hijo derecho de i: k
Hijo izqdo de k: l

```

Fichero abin.dat:

```

# 
a b c d # # # e # f # # g h # # i j # # k l # # #

```

Las funciones de `abin_E-S.h` serán muy útiles para introducir y mostrar árboles binarios en los programas de prueba de los ejercicios, especialmente si se usan ficheros para ello, pues se podrá repetir la ejecución de las pruebas sin tener que teclear una y otra vez los mismos datos.

## PROBLEMAS

1. Implementa un subprograma que calcule el número de nodos de un árbol binario.
2. Implementa un subprograma que calcule la altura de un árbol binario.
3. Implementa un subprograma que, dados un árbol binario y un nodo del mismo, determine la profundidad de este nodo en dicho árbol.
4. Añade dos nuevas operaciones al TAD *árbol binario*, una que calcule la profundidad de un nodo y otra que calcule la altura de un nodo en un árbol dado. Implementa esta operación para la representación vectorial (índices del padre, hijo izquierdo e hijo derecho).

5. Repite el ejercicio anterior para la representación enlazada de árboles binarios (punteros al padre, hijo izquierdo e hijo derecho).
6. Implementa un subprograma que determine el nivel de desequilibrio de un árbol binario, definido como el máximo desequilibrio de todos sus nodos. El desequilibrio de un nodo se define como la diferencia entre las alturas de los subárboles del mismo.
7. Implementa un subprograma que determine si un árbol binario es o no *pseudocompleto*.  
En este problema entenderemos que un árbol es *pseudocompleto*, si en el penúltimo nivel del mismo cada uno de los nodos tiene dos hijos o ninguno.

# Estructuras de Datos no Lineales

## Práctica 2

### Problemas de árboles binarios II

#### TRABAJO PREVIO

**Es obligatorio antes de asistir a la sesión de prácticas:**

1. Imprimir copia de este enunciado.
2. Lectura profunda del mismo.
3. Reflexión sobre el contenido de la práctica y generación de la lista de dudas asociada a dicha práctica y a los problemas que la componen.
4. **Esbozo serio de solución** de los problemas en papel (al menos de los que se hayan entendido).

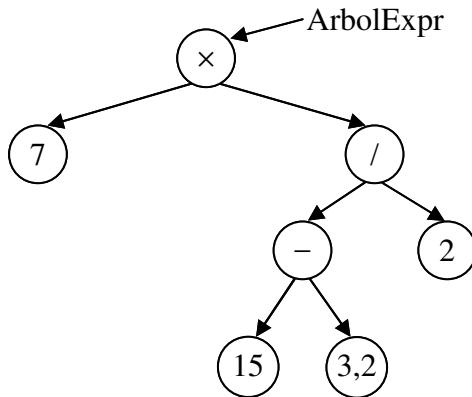
#### PASOS A SEGUIR

1. Escribir módulos que contengan las implementaciones de los subprogramas demandados en cada problema.
2. Para cada uno de los problemas escribir un programa de prueba, independiente de la representación del TAD elegida, donde se realicen las llamadas a los subprogramas del paso anterior, comprobando el resultado de salida para una batería suficientemente amplia de casos de prueba.

#### PROBLEMAS

1. Dos árboles binarios son similares cuando tienen idéntica estructura de ramificación, es decir, ambos son vacíos, o en caso contrario, tienen subárboles izquierdo y derecho similares. Implementa un subprograma que determine si dos árboles binarios son similares.
2. Para un árbol binario  $B$ , podemos construir el árbol binario reflejado  $B^R$  cambiando los subárboles izquierdo y derecho en cada nodo. Implementa un subprograma que devuelva el árbol binario reflejado de uno dado.
3. El TAD *árbol binario* puede albergar expresiones matemáticas mediante un árbol de expresión. Dentro del árbol binario los nodos hojas contendrán los operandos, y el resto de los nodos los operadores.
  - a) Define el tipo de los elementos del árbol para que los nodos puedan almacenar operadores y operandos.
  - b) Implementa una función que tome un árbol binario de expresión (aritmética) y devuelva el resultado de la misma. Por simplificar el problema se puede asumir que el árbol representa una expresión correcta. Los operadores binarios posibles en la expresión aritmética serán suma, resta, multiplicación y división.

Ejemplo: El siguiente árbol binario representa la expresión infija  $7 \times (15 - 3,2) / 2$ .

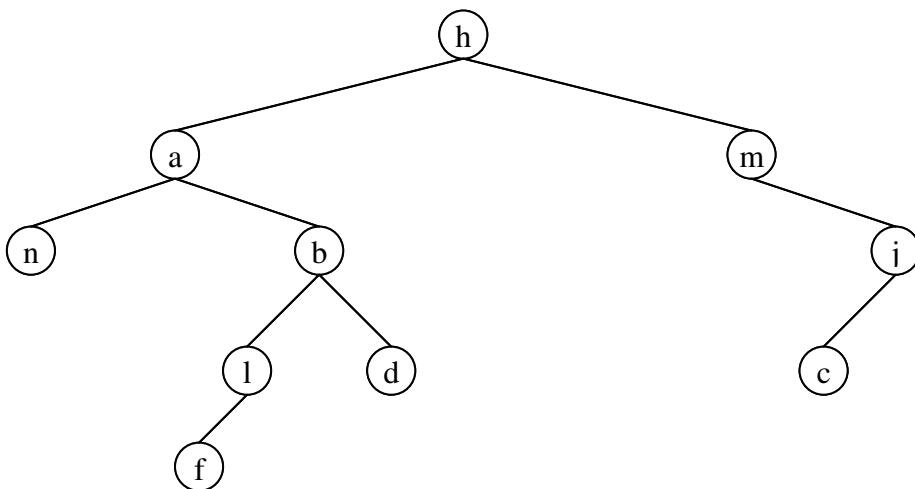


Nota: En el programa de prueba podemos usar las funciones `rellenarAbin()` de `abin_ES.h` para introducir por teclado o desde un fichero el árbol de expresión a evaluar. Sin embargo, en este caso, será necesario sobrecargar los operadores utilizados internamente en dichas funciones, es decir `>>`, `<<` y `!=` para el tipo de los elementos del árbol.

4. Una posible representación del TAD *Árbol Binario* consiste en almacenar los elementos del árbol en un vector cuyo tamaño depende de la altura máxima que pueda llegar a alcanzar el árbol. Cada nodo del árbol se corresponde con una única posición del vector, la cual viene determinada por el recorrido en inorden del árbol. Es decir, en el vector aparecen primero los nodos del subárbol izquierdo en inorden, luego la raíz y a continuación los nodos del subárbol derecho también en inorden. Por ejemplo, el árbol de la figura se representa como el vector

`(-, -, -, n, -, -, -, a, f, l, -, b, -, d, -, h, -, -, -, -, -, -, m, -, c, -, j, -, -, -),`

donde ‘-’ representa una posición vacía.



Los hijos izquierdo y derecho de un nodo  $n$  corresponden, respectivamente, a las posiciones  $n - (N+1)/2^{p+2}$  y  $n + (N+1)/2^{p+2}$ , donde  $p$  es la profundidad de  $n$  y  $N$  es el número máximo de nodos del árbol, es decir, el tamaño del vector. Por tanto, el padre de un nodo  $n$  se puede calcular de la siguiente forma:

$$Padre(n) = \begin{cases} n + (N+1)/2^{p+1} & \text{si } n \text{ es hijo izquierdo} \\ n - (N+1)/2^{p+1} & \text{si } n \text{ es hijo derecho} \end{cases}$$

Un nodo  $n$  es hijo izquierdo de su padre si  $n \bmod \left(\frac{N+1}{2^{p-1}}\right) = \frac{N+1}{2^{p+1}} - 1$

- Define la clase genérica  $Abin<T>$  para esta representación.
- Implementa una función miembro privada que calcule la profundidad de un nodo de un árbol binario representado de la forma descrita.
- Para esta representación implementa, al menos, el constructor de árboles vacíos y las operaciones  $insertarRaizB()$ ,  $insertarHijoIzqdoB()$  y  $padreB()$ , según la especificación del TAD Árbol Binario vista en clase.

# Implementación Dinámica

```

1  #ifndef AbinDin
2  #define AbinDin
3
4  #include <cassert>
5
6  template <typename T>
7  class Abin{
8      struct celda;
9  public:
10     typedef celda* nodo;
11
12     static const nodo NODO_NULO;      //Se declara la constante NODO_NULO que después se
13     inicializará
14
15     Abin(); //Ctor vacío
16     Abin(const Abin<T>& a);
17     Abin<T>& operator=(const Abin<T>& a);
18
19     void insertarRaizB(const T& e);
20     void insertarHijoIzqdoB(nodo n, const T& e);
21     void insertarHijoDrchoB(nodo n, const T& e);
22     void eliminarHijoIzqdoB(nodo n);
23     void eliminarHijoDrchoB(nodo n);
24     void eliminarRaizB();
25
26
27     bool arbolVacioB() const ;
28     const T& elemento(nodo n) const;
29     T& elemento(nodo n);
30     nodo raizB() const;
31     nodo padreB(nodo n) const;
32     nodo hijoIzqdoB(nodo n) const;
33     nodo hijoDrchoB(nodo n) const;
34
35
36     ~Abin();
37
38
39
40 private:
41     struct celda{
42         T elto;
43         nodo padre, hizq, hder;
44         celda(const T& e, nodo p = NODO_NULO): elto(e), padre(p), hizq(NODO_NULO),
45         hder(NODO_NULO) {}
46     };
47     nodo r; //A partir de él accedemos al árbol
48
49     void destruirNodos(nodo& n); //Se carga el nodo n y todos sus descendientes de manera
50     recursiva
51     nodo copiar(nodo n); //Copia el nodo n y todos sus descendientes de manera recursiva
52 };
53
54 template <typename T>
55 const typename Abin<T>::nodo Abin<T>::NODO_NULO(0);
56
57
58 template <typename T>
59 void Abin<T>::destruirNodos(Abin<T>::nodo& n) {
60
61     if(n != NODO_NULO) {
62         destruirNodos(n->hizq);
63         destruirNodos(n->hder);
64         delete n;
65         n = NODO_NULO; //Delete lo deja en estado indefinido. Para identificar que está
66         vacio, tenemos definido NODO_NULO
67     }
68 }
69
70 template <typename T>
71 typename Abin<T>::nodo Abin<T>::copiar(Abin<T>::nodo n) {
72     //Este método devuelve una copia del nodo n, es decir, se copia en otra posición de
73     //memoria.
74
75     nodo m = NODO_NULO;
76     if(n != NODO_NULO) {

```

```

75         m = new celda(n->elto); //m->padre por defecto inicializado a nulo, al igual que los
76         hijos
77         m->hder = copiar(n->hder); //Esto se hace fuera en vez de dentro del if simplemente
78         para que la llamada recursiva pare en
79         //el caso base definido.
80         if(m->hder != NODO_NULO) {
81
82             m->hder->padre = m;
83
84             m->hizq = copiar(n->hizq);
85             if(m->hizq != NODO_NULO) {
86
87                 m->hizq->padre = m;
88
89             }
90
91         }
92
93         return m;
94     }
95
96     template <typename T>
97     inline Abin<T>::Abin() : r(NODO_NULO) {}
98
99     template <typename T>
100    Abin<T>::Abin(const Abin<T>& a) {
101        r = copiar(a.r);
102    }
103
104    template <typename T>
105    Abin<T>& Abin<T>::operator=(const Abin<T>& a) {
106
107        if(this != &a){ //Para evitar autoasignación
108            this->~Abin(); //Se destruye el árbol actual
109            r = copiar(a.r);
110
111        }
112
113        return *this;
114    }
115
116    template <typename T>
117    inline void Abin<T>::insertarRaizB(const T& e) {
118        assert (r == NODO_NULO);
119        r = new celda(e);
120    }
121
122    template <typename T>
123    inline void Abin<T>::insertarHijoIzqdoB(Abin<T>::nodo n, const T& e) {
124        assert(n != NODO_NULO);
125        assert (n->hizq == NODO_NULO);
126
127        n->hizq = new celda(e, n);
128    }
129
130    template <typename T>
131    inline void Abin<T>::insertarHijoDrchoB(Abin<T>::nodo n, const T& e) {
132        assert(n != NODO_NULO);
133        assert (n->hder == NODO_NULO);
134
135        n->hder = new celda(e, n);
136    }
137
138    template <typename T>
139    inline void Abin<T>::eliminarHijoIzqdoB(Abin<T>::nodo n) {
140        assert(n != NODO_NULO)
141        assert( n->hizq != NODO_NULO);
142        assert(n->hizq->hizq == NODO_NULO && n->hizq->hder == NODO_NULO);
143        delete n->hizq;
144        n->hizq = NODO_NULO;
145    }
146
147    template <typename T>
148    inline void Abin<T>::eliminarHijoDrchoB(Abin<T>::nodo n) {
149        assert(n != NODO_NULO)
150        assert( n->hder != NODO_NULO);
151        assert(n->hder->hizq == NODO_NULO && n->hder->hder == NODO_NULO);
152        delete n->hder;

```

```

152     n->hder = NODO_NULO;
153 }
154
155 template <typename T>
156 inline void Abin<T>::eliminarRaizB() {
157     assert(r != NODO_NULO);
158     assert(r->hizq == NODO_NULO && r->hder == NODO_NULO);
159     delete r;
160     r = NODO_NULO;
161 }
162
163 template <typename T>
164 inline bool Abin<T>::arbolVacioB() const{
165     return (r == NODO_NULO);
166 }
167
168 template <typename T>
169 inline const T& Abin<T>::elemento(Abin<T>::nodo n) const{
170     assert(n != NODO_NULO);
171     return n->elto;
172 }
173
174 template <typename T>
175 inline T& Abin<T>::elemento(Abin<T>::nodo n) {
176     assert(n != NODO_NULO);
177     return n->elto;
178 }
179
180 template <typename T>
181 typename Abin<T>::nodo Abin<T>::raizB() const{
182     return r; //Si no existe devuelve NODO_NULO que según la precondition
183 }
184
185 template <typename T>
186 typename Abin<T>::nodo Abin<T>::padreB(Abin<T>::nodo n) const{
187     assert(n != NODO_NULO);
188     return n->padre; //Si no existe es NODO_NULO que era la precondition
189 }
190
191 template <typename T>
192 typename Abin<T>::nodo Abin<T>::hijoIzqdoB(Abin<T>::nodo n) const{
193     assert(n != NODO_NULO);
194     return n->hizq;
195 }
196
197 template <typename T>
198 typename Abin<T>::nodo Abin<T>::hijoDrchoB(Abin<T>::nodo n) const{
199     assert(n != NODO_NULO);
200     return n->hder;
201 }
202
203 template <typename T>
204 Abin<T>::~Abin() {
205     destruirNodos(r); //El trabajo lo hace la otra función
206 }
207
208
209
210
211
212 #endif // AbinDin
213

```

## Implementación Pseudestática

```

1  #ifndef Abinvec
2  #define Abinvec
3  #include <cassert>
4  #include <cstdlib> //Para el tipo size_t
5
6  /**
7   * OJO, nodos es un celda* mientras que nodo es un int a secas
8
9
10 */
11
12
13 template <typename T> class Abin{
14 public:
15     typedef int nodo; //Definimos el tipo nodo como un entero (posición en la matriz)
16
17     static const nodo NODO_NULO; //nodo especial que representa que no hay un nodo ahí.
18
19     explicit Abin(size_t maxNodos); //Constructor      size_t: entero positivo, usado para
designar tamaños
20     Abin(const Abin<T>& a); //Constructor de Copia
21     Abin<T>& operator =(const Abin<T>& a); //Operador de asignación
22
23     void insertarRaizB(const T& e);
24     void insertarHijoIzqdoB(nodo n, const T& e);
25     void insertarHijoDrchoB(nodo n, const T& e);
26     void eliminarHijoIzqdoB(nodo n);
27     void eliminarHijoDrchoB(nodo n);
28     void eliminarRaizB();
29
30     ~Abin();
31     bool arbolVacioB() const;
32
33     const T& elemento(nodo n) const; //versión solo lectura
34     T& elemento(nodo n); //versión por referencia (L/E)
35
36     nodo raizB() const; //Devuelve el nodo raiz
37     nodo padreB(nodo n) const; //Devuelve el nodo padre de n
38     nodo hijoIzqdoB(nodo n) const; //Devuelve el hijo izquierdo de n
39     nodo hijoDrchoB(nodo n) const; //Devuelve el hijo derecho de n
40
41
42 private:
43     struct celda{
44         T elto;
45         nodo padre, hizq, hder;
46     };
47     celda* nodos;
48     int maxNodos; //Número máximo, según la capacidad de la matriz que conforma la
representación interna
49     int numNodos; //Número actual de nodos con valor
50
51 };
52
53 // Definición del nodo nulo: Debe ser aquí, para que al hacer el include, este código quede
en el ámbito global del programa
54 // de manera que se ejecute antes que el main (y por tanto antes que cualquier otra instrucción
55
56 template <typename T>
57 const typename Abin<T>::nodo Abin<T>::NODO_NULO(-1); //Lo inicializo a -1
58 // typename tipo atributo valor
59 //Se pone typename para indicar que nodo(Abin) es un tipo y no un atributo
60
61 //Implementación de los métodos:
62
63
64 //Constructor
65 template <typename T>
66 inline Abin<T>::Abin(size_t maxNodos): nodos(new celda[maxNodos]), maxNodos(maxNodos),
numNodos(0) {
67
68 //Constructor de copia
69 template <typename T>
70 Abin<T>::Abin(const Abin<T>& a): nodos(new celda[a.maxNodos]), maxNodos(a.maxNodos),
numNodos(a.numNodos) {
71
72     //Se copia el vector
73
74     for(nodo i = 0 ; i < numNodos ; i++){

```

```

75         nodos[i] = a.nodos[i]; //Se copian las estructuras celdas enteras, usando el
76         constructor de "celda" por defecto
77     }
78
79 //Operador de asignación
80 template <typename T>
81 Abin<T>& Abin<T>::operator =(const Abin<T>& a){
82     if (this != &a){ //Para evitar autoasignación
83
84         //Se destruye el antiguo vector y se crea uno nuevo si no tienen la misma longitud
85
86         if(maxNodos != a.maxNodos){
87             delete[] nodos;
88             nodos = new celda[a.maxNodos];
89             maxNodos = a.maxNodos;
90         }
91         numNodos = a.numNodos;
92         for(nodo i = 0 ; i < numNodos ; i++){
93             nodos[i] = a.nodos[i];
94         }
95     }
96
97     return *this;
98 }
99
100
101
102
103 template <typename T>
104 void Abin<T>::insertarRaizB(const T& e){
105     numNodos = 1;
106     nodos[0].elto = e;
107     nodos[0].padre = NODO_NULO;
108     nodos[0].hizq = NODO_NULO;
109     nodos[0].hder = NODO_NULO;
110 }
111
112 template <typename T>
113 void Abin<T>::insertarHijoIzqdoB(Abin<T>::nodo n, const T& e){
114     assert(n>=0 && n < numNodos); //En esta implementación es fácil comprobar que n ∈ al
115     assert(nodos[n].hizq == NODO_NULO);
116     assert(numNodos < maxNodos); //Requerida por la implementación
117
118     //Se añade el nuevo nodo al final
119
120     nodos[n].hizq = numNodos;
121     nodos[numNodos].elto = e;
122     nodos[numNodos].padre = n;
123     nodos[numNodos].hizq = NODO_NULO;
124     nodos[numNodos].hder = NODO_NULO;
125     numNodos++;
126 }
127
128 template <typename T>
129 void Abin<T>::insertarHijoDrchoB(Abin<T>::nodo n, const T& e){
130     assert(n>=0 && n < numNodos); //En esta implementación es fácil comprobar que n ∈ al
131     assert(nodos[n].hder == NODO_NULO);
132     assert(numNodos < maxNodos); //Requerida por la implementación
133
134     //Se añade el nuevo nodo al final
135
136     nodos[n].hder = numNodos;
137     nodos[numNodos].elto = e;
138     nodos[numNodos].padre = n;
139     nodos[numNodos].hizq = NODO_NULO;
140     nodos[numNodos].hder = NODO_NULO;
141     numNodos++;
142 }
143
144
145 template <typename T>
146 void Abin<T>::eliminarHijoIzqdoB(Abin<T>::nodo n){
147     assert(n>= 0 && n < numNodos); //Comprobamos que el nodo ∈ al árbol
148     nodo hizqdo = nodos[n].hizq;
149     assert(hizqdo != NODO_NULO); //Comprobamos que hay nodo que eliminar
150

```

```

151     assert(nodos[hizqdo].hizq == NODO_NULO && nodos[hizqdo].hder == NODO_NULO);
152     //Comprobamos que sea nodo hoja
153
154     if(hizqdo != numNodos-1){ // Si no es el último (hay que mover el ultimo a la posición de n
155
156         nodos[hizqdo] = nodos[numNodos-1]; //Se copia la estructura celda entera de la
157         //última posición a esa
158
159         //Ahora hay que modificar los padres e hijos del nodo que hemos movido de la última
160         //posición a esa
161
162         //Modificando los hijos
163         if(nodos[hizqdo].hder != NODO_NULO){
164             nodos[nodos[hizqdo].hder].padre = hizqdo;
165         }
166
167         if(nodos[hizqdo].hizq != NODO_NULO){
168             nodos[nodos[hizqdo].hizq].padre = hizqdo;
169         }
170
171         //Modificando el padre
172         //Compruebo si era hijo izq o dcho del padre
173         if(nodos[nodos[hizqdo].padre].hizq == numNodos-1){ //Si es numNodos-1 es que era hija.
174             nodos[nodos[hizqdo].padre].hizq = hizqdo;
175         }else{ //Si no era hijo izquierdo, entonces era hijo derecho
176             nodos[nodos[hizqdo].padre].hder = hizqdo;
177         }
178
179     }
180     nodos[hizqdo].padre.hizq = NODO_NULO; //En cualquier caso, se le indica al nodo padre
181     //que ya no tiene hijo izquierdo
182     numNodos--; //Indicamos el nuevo final de los datos
183 }
184
185 template <typename T>
186 void Abin<T>::eliminarHijoDrchoB(Abin<T>::nodo n){
187     assert(n>= 0 && n < numNodos); //Comprobamos que el nodo e al árbol
188     nodo hdrcho = nodos[n].hder;
189     assert(hdrcho != NODO_NULO); //Comprobamos que hay nodo que eliminar
190     assert(nodos[hdrcho].hizq == NODO_NULO && nodos[hdrcho].hder == NODO_NULO);
191     //Comprobamos que sea nodo hoja
192
193     if(hdrcho != numNodos-1){ // Si no es el último (hay que mover el ultimo a la posición de n
194
195
196         nodos[hdrcho] = nodos[numNodos-1]; //Se copia la estructura celda entera de la
197         //última posición a esa
198
199         //Ahora hay que modificar los padres e hijos del nodo que hemos movido de la última
200         //posición a esa
201
202         //Modificando los hijos
203         if(nodos[hdrcho].hder != NODO_NULO){
204             nodos[nodos[hdrcho].hder].padre = hdrcho;
205         }
206
207         if(nodos[hdrcho].hizq != NODO_NULO){
208             nodos[nodos[hdrcho].hizq].padre = hdrcho;
209         }
210
211         //Modificando el padre
212         //Compruebo si era hijo izq o dcho del padre
213         if(nodos[nodos[hdrcho].padre].hizq == numNodos-1){ //Si es numNodos-1 es que era hija.
214             nodos[nodos[hdrcho].padre].hizq = hdrcho;
215         }else{ //Si no era hijo izquierdo, entonces era hijo derecho
216             nodos[nodos[hdrcho].padre].hder = hdrcho;
217         }
218
219     }
220     nodos[hdrcho].padre.hder = NODO_NULO; //En cualquier caso, se le indica al nodo padre
221     //que ya no tiene hijo izquierdo
222     numNodos--; //Indicamos el nuevo final de los datos

```

```

222
223 }
224
225 template <typename T>
226 inline void Abin<T>::eliminarRaizB() {
227     assert(numNodos == 1); //Compruebo que solo queda la raiz
228
229     numNodos = 0; //Indico que se ha borrado el elemento de la posición 0 y que lo que haya
230     ahí es basura
231 }
232
233 template <typename T>
234 inline Abin<T>::~Abin() {
235     delete[] nodos;
236 }
237
238 template <typename T>
239 inline bool Abin<T>::arbolVacioB() const{
240     return (numNodos == 0);
241 }
242
243 template <typename T>
244 inline const T& Abin<T>::elemento(Abin<T>::nodo n) const{
245     assert(n>=0 && n < numNodos);
246     return nodos[n].elto;
247 }
248
249 template <typename T>
250 inline T& Abin<T>::elemento(Abin<T>::nodo n) {
251     assert(n>=0 && n < numNodos);
252     return nodos[n].elto;
253 }
254
255 template <typename T>
256 inline typename Abin<T>::nodo Abin<T>::raizB() const{
257     return (numNodos>0) ? 0 : NODO_NULO;
258 }
259
260 template <typename T>
261 inline typename Abin<T>::nodo Abin<T>::padreB(Abin<T>::nodo n) const{
262     assert(n>=0 && n < numNodos);
263
264     return nodos[n].padre; //Si n es 0, el campo padre será NODO_NULO
265 }
266
267 template <typename T>
268 inline typename Abin<T>::nodo Abin<T>::hijoIzqdoB(Abin<T>::nodo n) const{
269     assert(n>=0 && n < numNodos);
270     return nodos[n].hizq;
271 }
272
273 template <typename T>
274 inline typename Abin<T>::nodo Abin<T>::hijoDrchoB(Abin<T>::nodo n) const{
275     assert(n>=0 && n < numNodos);
276     return nodos[n].hder;
277 }
278
279
280 #endif // Abinvec
281

```

# Implementación Pseudoestática celdas relativas

QuesoViejo\_

```
1  #ifndef ABINVECREL
2  #define ABINVECREL
3
4  #include <cassert>
5  #include <cstdlib>    //Para el size_t
6
7  template <typename T>
8  class Abin{
9
10 public:
11     typedef int nodo;
12     static const nodo NODO_NULO;    //Solo servirá para devolvérselo al usuario. Para
13     comprobar que no existe ese nodo está ELTO_NULO
14
15     explicit Abin(size_t maxNodos, const T& e_nulo = T());    //Constructor por defecto
16     Abin(const Abin<T>& a);
17     Abin<T>& operator=(const Abin<T>& a);
18
19     void insertarRaizB(const T& e);
20     void insertarHijoIzqdoB(nodo n, const T& e);
21     void insertarHijoDrchoB(nodo n, const T& e);
22     void eliminarHijoIzqdoB(nodo n);
23     void eliminarHijoDrchoB(nodo n);
24     void eliminarRaizB();
25
26     bool arbolVacioB()const;
27     const T& elemento(nodo n)const;
28     T& elemento(nodo n);
29     nodo raizB()const;
30     nodo padreB(nodo n)const;
31     nodo hijoDrchoB(nodo n)const;
32     nodo hijoIzqdoB(nodo n)const;
33
34
35     ~Abin();
36
37 private:
38
39     T* nodos;    //vector de nodos
40     int maxNodos;
41     T ELTO_NULO;
42
43 };
44
45 template <typename T>
46 const typename Abin<T>::nodo Abin<T>::NODO_NULO(-1);
47
48 template <typename T>
49 Abin<T>::Abin(size_t maxNodos, const T& e_nulo): maxNodos(maxNodos), ELTO_NULO(e_nulo),
50 nodos(new T[maxNodos]) {
51     for(nodo i = 0; i < maxNodos; i++) { //Se inicializan todos los valores al elemento nulo
52         para indicar que están vacíos
53         nodos[i] = ELTO_NULO;
54     }
55 }
56
57 template <typename T>
58 Abin<T>::Abin(const Abin<T>& a): maxNodos(a.maxNodos), ELTO_NULO(a.ELTO_NULO), nodos(new
59 T[a.maxNodos]) {
60     for(nodo i = 0; i < maxNodos; i++) {
61         nodos[i] = a.nodos[i];
62     }
63 }
64
65 template <typename T>
66 Abin<T>& Abin<T>::operator=(const Abin<T>& a) {
67
68     if(this != &a){ //Para evitar autoasignación
69
70         if(maxNodos != a.maxNodos){
71             delete[] nodos;
72             nodos = new T[a.maxNodos];
73             maxNodos = a.maxNodos;
74         }
75         ELTO_NULO = a.ELTO_NULO;
76         for(nodo i = 0; i < maxNodos; i++) {
77             nodos[i] = a.nodos[i];
78         }
79     }
80
81     return *this;
82 }
```

```

76         }
77     }
78
79     return *this;
80 }
81
82 template <typename T>
83 inline void Abin<T>::insertarRaizB(const T& e) {
84     assert(nodos[0] == ELTO_NULO);
85
86     nodos[0] = e;
87 }
88
89 template <typename T>
90 inline void Abin<T>::insertarHijoIzqdoB(nodo n, const T& e) {
91     assert(n>=0 && n < maxNodos);
92     assert(nodos[n] != ELTO_NULO);
93     assert(2*n+1 < maxNodos);
94     assert(nodos[2*n+1] == ELTO_NULO);
95
96     nodos[2*n+1] = e;
97 }
98
99
100 template <typename T>
101 inline void Abin<T>::insertarHijoDrchoB(nodo n, const T& e) {
102     assert(n>=0 && n < maxNodos);
103     assert(nodos[n] != ELTO_NULO);
104     assert(2*n+2 < maxNodos);
105     assert(nodos[2*n+2] == ELTO_NULO);
106
107     nodos[2*n+2] = e;
108 }
109
110 template <typename T>
111 inline void Abin<T>::eliminarHijoIzqdoB(nodo n) {
112     assert(n>=0 && n < maxNodos);
113     assert(nodos[n] != ELTO_NULO);
114     assert(2*n+1 < maxNodos);
115     assert(nodos[2*n+1] != ELTO_NULO);
116
117     nodos[2*n+1] = ELTO_NULO;
118 }
119
120 template <typename T>
121 inline void Abin<T>::eliminarHijoDrchoB(nodo n) {
122     assert(n>=0 && n < maxNodos);
123     assert(nodos[n] != ELTO_NULO);
124     assert(2*n+2 < maxNodos);
125     assert(nodos[2*n+2] != ELTO_NULO);
126
127     nodos[2*n+2] = ELTO_NULO;
128 }
129
130 template <typename T>
131 inline void Abin<T>::eliminarRaizB() {
132     assert(nodos[0] != ELTO_NULO);
133     assert(nodos[1] == ELTO_NULO && nodos[2] == ELTO_NULO);
134
135     nodos[0] = ELTO_NULO;
136 }
137
138 template <typename T>
139 inline bool Abin<T>::arbolVacioB() const{
140     return (nodos[0] == ELTO_NULO);
141 }
142
143 template <typename T>
144 inline const T& Abin<T>::elemento(Abin<T>::nodo n) const{
145     assert(n>=0 && n < maxNodos);
146     assert(nodos[n] != ELTO_NULO);
147
148     return nodos[n];
149 }
150
151 template <typename T>
152 inline T& Abin<T>::elemento(Abin<T>::nodo n) {
153     assert(n>= 0 && n < maxNodos);
154     assert(nodos[n] != ELTO_NULO);

```

```

155     return nodos[n];
156 }
157
158 template <typename T>
159 inline typename Abin<T>::nodo Abin<T>::raizB() const{
160     return (nodos[0] == ELTO_NULO) ? NODO_NULO : 0;
161 }
162
163 template <typename T>
164 inline typename Abin<T>::nodo Abin<T>::padreB(Abin<T>::nodo n) const{
165     assert(n>=0 && n < maxNodos);
166     assert(nodos[n] != ELTO_NULO);
167     return (n==0) ? NODO_NULO: (n-1)/2;
168 }
169
170 template <typename T>
171 inline typename Abin<T>::nodo Abin<T>::hijoIzqdoB(nodo n) const{
172     assert(n>=0 && n < maxNodos);
173     assert(nodos[n] != ELTO_NULO);
174
175
176     return (nodos[2*n+1]==ELTO_NULO || 2*n+1 >=maxNodos) ? NODO_NULO : 2*n+1;
177 }
178
179
180 template <typename T>
181 inline typename Abin<T>::nodo Abin<T>::hijoDrchoB(nodo n) const{
182     assert(n>=0 && n < maxNodos);
183     assert(nodos[n] != ELTO_NULO);
184
185
186     return (nodos[2*n+2]==ELTO_NULO || 2*n+2 >=maxNodos) ? NODO_NULO : 2*n+2;
187 }
188
189
190 template <typename T>
191 Abin<T>::~Abin() {
192     delete [] nodos;
193 }
194
195
196
197 #endif // ABINVECREL
198
199

```

# Funciones para Entrada y Salida de Árboles

# QuesoViejo\_

```
1 #ifndef ABIN_E_S_H
2 #define ABIN_E_S_H
3
4 #include <cassert>
5 #include <iostream>
6 #include <fstream>
7
8 using std::istream;
9 using std::ostream;
10 using std::cin;
11 using std::cout;
12 using std::endl;
13
14 template <typename T>
15 void rellenarAbin(Abin<T>& A, const T& fin)
16 // Pre: A está vacío.
17 // Post: Rellena el árbol A con la estructura y elementos
18 // leídos en preorden de la entrada estándar, usando
19 // fin como elemento especial para introducir nodos nulos.
20 {
21     T e;
22
23     assert(A.arbolVacioB());
24     cout << "Raiz (Fin = " << fin << "): "; cin >> e;
25     if (e != fin) {
26         A.insertarRaizB(e);
27         rellenarDescendientes(A.raizB(), A, fin);
28     }
29 }
30
31 template <typename T>
32 void rellenarDescendientes(typename Abin<T>::nodo r, Abin<T>& A, const T& fin)
33 // Pre: r es un nodo hoja de A.
34 // Post: Lee de la entrada estándar y en preorden los descendientes de r,
35 // usando fin para introducir nodos nulos.
36 {
37     T ehi, ehd;
38
39     assert(A.hijoIzqdoB(r) == Abin<T>::NODO_NULO &&
40           A.hijoDrchoB(r) == Abin<T>::NODO_NULO);
41     cout << "Hijo izqdo. de " << A.elemento(r) <<
42           " (Fin = " << fin << "): ";
43     cin >> ehi;
44     if (ehi != fin) {
45         A.insertarHijoIzqdoB(r, ehi);
46         rellenarDescendientes(A.hijoIzqdoB(r), A, fin);
47     }
48     cout << "Hijo drcho. de " << A.elemento(r) <<
49           " (Fin = " << fin << "): ";
50     cin >> ehd;
51     if (ehd != fin) {
52         A.insertarHijoDrchoB(r, ehd);
53         rellenarDescendientes(A.hijoDrchoB(r), A, fin);
54     }
55 }
56
57 template <typename T>
58 void rellenarAbin(istream& is, Abin<T>& A)
59 // Pre: A está vacío.
60 // Post: Extrae los nodos de A del flujo de entrada is,
61 // que contendrá el elemento especial que denota
62 // un nodo nulo seguido de los elementos en preorden,
63 // incluyendo los correspondientes a nodos nulos.
64 {
65     T e, fin;
66
67     assert(A.arbolVacioB());
68     if (is >> fin && is >> e && e != fin) {
69         A.insertarRaizB(e);
70         rellenarDescendientes(is, A.raizB(), A, fin);
71     }
72 }
73
74 template <typename T>
75 void rellenarDescendientes(istream& is, typename Abin<T>::nodo r, Abin<T>& A, const T& fin)
76 // Pre: r es un nodo hoja de A.
77 // Post: Lee del flujo de entrada is y en preorden los descendientes de r,
78 // usando fin para los nodos nulos.
79 {
```

```

80     T ehi, ehd;
81
82     assert(A.hijoIzqdoB(r) == Abin<T>::NODO_NULO &&
83            A.hijoDrchoB(r) == Abin<T>::NODO_NULO);
84     if (is >> ehi && ehi != fin) {
85         A.insertarHijoIzqdoB(r, ehi);
86         rellenarDescendientes(is, A.hijoIzqdoB(r), A, fin);
87     }
88     if (is >> ehd && ehd != fin) {
89         A.insertarHijoDrchoB(r, ehd);
90         rellenarDescendientes(is, A.hijoDrchoB(r), A, fin);
91     }
92 }
93
94 template <typename T>
95 void imprimirAbin (const Abin<T>& A)
96 // Post: Muestra los nodos de A en la salida estándar
97 {
98     if (!A.arbolVacioB())
99     {
100         cout << "Raíz del árbol: "
101             << A.elemento(A.raizB()) << endl;
102         imprimirDescendientes(A.raizB(), A);
103     }
104     else
105         cout << "Árbol vacío\n";
106 }
107
108 template <typename T>
109 void imprimirDescendientes (typename Abin<T>::nodo r, const Abin<T>& A)
110 // Post: Muestra los descendientes de r en la salida estándar.
111 {
112     if (A.hijoIzqdoB(r) != Abin<T>::NODO_NULO)
113     {
114         cout << "Hijo izquierdo de " << A.elemento(r) << ":" 
115             << A.elemento(A.hijoIzqdoB(r)) << endl;
116         imprimirDescendientes(A.hijoIzqdoB(r), A);
117     }
118     if (A.hijoDrchoB(r) != Abin<T>::NODO_NULO)
119     {
120         cout << "Hijo derecho de " << A.elemento(r) << ":" 
121             << A.elemento(A.hijoDrchoB(r)) << endl;
122         imprimirDescendientes(A.hijoDrchoB(r), A);
123     }
124 }
125
126 template <typename T>
127 void imprimirAbin (ostream& os, const Abin<T>& A, const T& fin)
128 // Post: Inserta en el flujo de salida os los nodos de A en preorden,
129 // precedidos del elemento especial usado para denotar un nodo nulo.
130 {
131     if (!A.arbolVacioB())
132     {
133         os << fin << endl
134             << A.elemento(A.raizB()) << ' ';
135         imprimirDescendientes(os, A.raizB(), A, fin);
136         os << endl;
137     }
138 }
139
140 template <typename T>
141 void imprimirDescendientes (ostream& os, typename Abin<T>::nodo r, const Abin<T>& A, const
142 T& fin)
143 // Post: Inserta en el flujo de salida os y en preorden
144 // los descendientes de r, usando fin como nodo nulo.
145 {
146     if (A.hijoIzqdoB(r) != Abin<T>::NODO_NULO)
147     {
148         os << A.elemento(A.hijoIzqdoB(r)) << ' ';
149         imprimirDescendientes(os, A.hijoIzqdoB(r), A, fin);
150     }
151     else
152         os << fin << ' ';
153     if (A.hijoDrchoB(r) != Abin<T>::NODO_NULO)
154     {
155         os << A.elemento(A.hijoDrchoB(r)) << ' ';
156         imprimirDescendientes(os, A.hijoDrchoB(r), A, fin);
157     }
158 }
```

```
158         os << fin << ' ' ;
159     }
160
161 #endif // ABIN_E_S_H
162
```



## Práctica 5: Ej 5

```

1  #include "AbinDin.h"
2  #include <iostream>
3  #include <fstream>
4  #include "abin_E-S.h"
5
6
7  /*
8   TIENES QUE HACERTE UN ARCHIVO NUMS.dat (CON EL MISMO BLOC DE NOTAS PUEDES)
9   Y METERLE LOS DATOS. PRIMERO EL TERMINADOR Y LUEGO LOS NODOS.
10
11  EJEMPLO:
12
13  #
14  a b c d # # e # f # # g h i # # j # k # l z # # m # #
15
16
17
18  */
19
20  template <typename T>
21  int numNodos(const Abin<T>& A);
22
23  template <typename T>
24  int numNodos_rec(typename Abin<T>::nodo n, const Abin<char>& A);
25
26  using namespace std;
27
28
29  int main()
30  {
31      Abin<char> A;
32
33      cout << "\n*** Lectura de árbol binario fichero de datos ***\n";
34      ifstream fe("NUMS.dat"); // abrir fichero de entrada
35      llenarAbin(fe, A); // desde fichero
36      fe.close();
37      cout << "\n*** Mostrar árbol binario B ***\n";
38      imprimirAbin(A); // en std::cout
39
40
41      int numeroNodos;
42      numeroNodos = numNodos(A);
43
44      cout << "\n\nEl arbol tiene "<<numeroNodos<<" nodos"<<endl;
45  }
46
47
48
49
50
51  template <typename T>
52  int numNodos(const Abin<T>& A)
53  {
54      return numNodos_rec(A.raizB(), A);
55  }
56
57  template <typename T>
58  int numNodos_rec(typename Abin<T>::nodo n, const Abin<T>& A)
59  {
60
61      if(n == Abin<char>::NODO_NULO)
62      {
63          return 0;
64      }
65      else
66      {
67          return 1 + numNodos_rec(A.hijoIzqdoB(n), A) + numNodos_rec(A.hijoDrchoB(n), A);
68      }
69  }
70
71
72

```

## Práctica 5: Ej 2

## QuesoViejo\_

```
1  #include "AbinDin.h"
2  #include <iostream>
3  #include <fstream>
4  #include "abin_E-S.h"
5
6
7
8  template <typename T>
9  int CalcularAltura(const Abin<T>& A);
10
11 template <typename T>
12 int CalcularAltura_rec(typename Abin<T>::nodo n, const Abin<T>& A);
13
14 int Max(int n1, int n2);
15
16
17 using namespace std;
18
19 int main()
20 {
21     Abin<char> A;
22
23     cout << "\n*** Lectura de árbol binario fichero de datos ***\n";
24     ifstream fe("NUMS.dat"); // abrir fichero de entrada
25     llenarAbin(fe, A); // desde fichero
26     fe.close();
27     cout << "\n*** Mostrar árbol binario B ***\n";
28     imprimirAbin(A); // en std::cout
29
30     int altura;
31     altura = CalcularAltura(A);
32
33     cout << "La altura del arbol es: " << altura << endl;
34
35
36 }
37
38
39 template <typename T>
40 int CalcularAltura(const Abin<T>& A)
41 {
42     return CalcularAltura_rec(A.raizB(), A);
43 }
44
45
46 template <typename T>
47 int CalcularAltura_rec(typename Abin<T>::nodo n, const Abin<T>& A)
48 {
49     if(n == Abin<T>::NODO_NULO)
50     {
51         return -1; //La altura de un NODO_NULO es -1, la altura de un nodo hoja es 0, ...
52     }
53     else
54     {
55         return 1 + Max(CalcularAltura_rec(A.hijoIzqdoB(n), A),
56                         CalcularAltura_rec(A.hijoDrchoB(n), A));
57     }
58 }
59
60 int Max(int n1, int n2)
61 {
62     return (n1 > n2) ? n1 : n2;
63 }
64
```

## Práctica 3: Ej 3

```

1  #include "AbinDin.h"
2  #include <iostream>
3  #include <fstream>
4  #include "abin_E-S.h"
5
6
7  template <typename T>
8  int CalcProfundidadNodo_iter(typename Abin<T>::nodo n, const Abin<T>& A);
9
10 template <typename T>
11 int CalcProfundidadNodo_rec(typename Abin<T>::nodo n, const Abin<T>& A);
12
13
14 using namespace std;
15
16 int main()
17 {
18     Abin<char> A;
19
20     cout << "\n*** Lectura de árbol binario fichero de datos ***\n";
21     ifstream fe("NUMS.dat"); // abre fichero de entrada
22     llenarAbin(fe, A); // desde fichero
23     fe.close();
24     cout << "\n*** Mostrar árbol binario B ***\n";
25     imprimirAbin(A); // en std::cout
26
27     int resultado;
28     resultado = CalcProfundidadNodo_iter(A.hijoIzqdoB(A.raizB()), A);
29     cout<<"[ITER] La profundidad del hijo del nodo raiz es: "<<resultado<<endl;
30
31     resultado = CalcProfundidadNodo_rec(A.raizB(), A);
32     cout<<"[REC] La profundidad del nodo raiz es: "<<resultado<<endl;
33
34
35 }
36
37 template <typename T>
38 int CalcProfundidadNodo_iter(typename Abin<T>::nodo n, const Abin<T>& A)
39 {
40     int alt;
41     alt = -1; //Para que la profundidad del nodo raiz sea 0, la de uno de sus hijos 1...
42
43     while(n != Abin<T>::NODO_NULO)
44     {
45         ++alt;
46         n = A.padreB(n);
47     }
48
49     return alt;
50
51 }
52
53 template <typename T>
54 int CalcProfundidadNodo_rec(typename Abin<T>::nodo n, const Abin<T>& A)
55 {
56
57     if(n == Abin<T>::NODO_NULO)
58     {
59         return -1;
60     }
61     else
62     {
63         return 1+CalcProfundidadNodo_iter(A.padreB(n), A);
64     }
65
66
67
68 }
69

```

# Práctica 5: Ej 4 (fichero.h)

# QuesoViejo\_

```
1  #ifndef Abinvec
2  #define Abinvec
3  #include <cassert>
4  #include <cstdlib> //Para el tipo size_t
5
6  /**
7   * OJO, nodos es un celda* mientras que nodo es un int a secas
8
9
10 */
11
12
13 template <typename T> class Abin{
14 public:
15     typedef int nodo; //Definimos el tipo nodo como un entero (posición en la matriz)
16
17     static const nodo NODO_NULO; //nodo especial que representa que no hay un nodo ahí.
18
19     explicit Abin(size_t maxNodos); //Constructor      size_t: entero positivo, usado para
designar tamaños
20     Abin(const Abin<T>& a); //Constructor de Copia
21     Abin<T>& operator =(const Abin<T>& a); //Operador de asignación
22
23     void insertarRaizB(const T& e);
24     void insertarHijoIzqdoB(nodo n, const T& e);
25     void insertarHijoDrchoB(nodo n, const T& e);
26     void eliminarHijoIzqdoB(nodo n);
27     void eliminarHijoDrchoB(nodo n);
28     void eliminarRaizB();
29
30     ~Abin();
31     bool arbolVacioB() const;
32
33     const T& elemento(nodo n) const; //versión solo lectura
34     T& elemento(nodo n); //versión por referencia (L/E)
35
36     nodo raizB() const; //Devuelve el nodo raiz
37     nodo padreB(nodo n) const; //Devuelve el nodo padre de n
38     nodo hijoIzqdoB(nodo n) const; //Devuelve el hijo izquierdo de n
39     nodo hijoDrchoB(nodo n) const; //Devuelve el hijo derecho de n
40
41
42     /*-----NUEVAS-----*/
43     int CalcProfundidadNodoB(nodo n);
44     int CalcAlturaNodoB(nodo n);
45     /*-----FIN NUEVAS-----*/
46
47
48
49
50
51
52
53
54 private:
55     struct celda{
56         T elto;
57         nodo padre, hizq, hder;
58     };
59     celda* nodos;
60     int maxNodos; //Número máximo, según la capacidad de la matriz que conforma la
representación interna
61     int numNodos; //Número actual de nodos con valor
62
63     int Max(int n1, int n2);
64
65 };
66
67 // Definición del nodo nulo: Debe ser aquí, para que al hacer el include, este código quede
en el ámbito global del programa
68 // de manera que se ejecute antes que el main (y por tanto antes que cualquier otra instrucción
69
70 template <typename T>
71 const typename Abin<T>::nodo Abin<T>::NODO_NULO(-1); //Lo inicializo a -1
72 // typename tipo atributo valor
73 //Se pone typename para indicar que nodo(< T >) es un tipo y no un atributo
74
75 //Implementación de los métodos:
```

```

77
78 //Constructor
79 template <typename T>
80 inline Abin<T>::Abin(size_t maxNodos): nodos(new celda[maxNodos]), maxNodos(maxNodos),
81 numNodos(0) {}
82
83 //Constructor de copia
84 template <typename T>
85 Abin<T>::Abin(const Abin<T>& a): nodos(new celda[a.maxNodos]), maxNodos(a.maxNodos),
86 numNodos(a.numNodos) {
87
88     //Se copia el vector
89
90     for(nodo i = 0 ; i < numNodos ; i++){
91         nodos[i] = a.nodos[i]; //Se copian las estructuras celdas enteras, usando el
92         constructor de "celda" por defecto
93     }
94 }
95
96 /*-----NUEVAS OPERACIONES EJ 4
97 -----*/
98
99 template <typename T>
100 int Abin<T>::CalcProfundidadNodoB(typename Abin<T>::nodo n)
101 {
102     int prof = -1;
103
104     while(n != NODO_NULO)
105     {
106         ++prof;
107         n = nodos[n].padre;
108     }
109
110     return prof;
111 }
112
113 template <typename T>
114 int Abin<T>::CalcAlturaNodoB(typename Abin<T>::nodo n)
115 {
116     if(n == NODO_NULO)
117     {
118         return -1;
119     }
120     else
121     {
122         return 1 + Max(CalcAlturaNodoB(nodos[n].hder),CalcAlturaNodoB(nodos[n].hizq));
123     }
124
125
126 template <typename T>
127 int Abin<T>::Max(int n1, int n2)
128 {
129     return (n1 > n2) ? n1 : n2;
130 }
131
132
133 /*-----FIN NUEVAS OPERACIONES EJ 4
134 -----*/
135
136
137
138 //Operador de asignación
139 template <typename T>
140 Abin<T>& Abin<T>::operator =(const Abin<T>& a) {
141     if (this != &a){ //Para evitar autoasignación
142
143         //Se destruye el antiguo vector y se crea uno nuevo si no tienen la misma longitud
144
145         if(maxNodos != a.maxNodos) {
146             delete[] nodos;
147             nodos = new celda[a.maxNodos];
148             maxNodos = a.maxNodos;
149         }
150         numNodos = a.numNodos;
151
152         for(nodo i = 0 ; i < numNodos ; i++){
153             nodos[i] = a.nodos[i]; //Se copian las estructuras celdas enteras, usando el
154             constructor de "celda" por defecto
155         }
156     }
157
158     return *this;
159 }

```

```

151             for(nodo i = 0 ; i < numNodos ; i++){
152                 nodos[i] = a.nodos[i];
153             }
154         }
155     }
156     return *this;
157 }
158
159
160
161
162     template <typename T>
163     void Abin<T>::insertarRaizB(const T& e) {
164         numNodos = 1;
165         nodos[0].elto = e;
166         nodos[0].padre = NODO_NULO;
167         nodos[0].hizq = NODO_NULO;
168         nodos[0].hder = NODO_NULO;
169     }
170
171     template <typename T>
172     void Abin<T>::insertarHijoIzqdoB(Abin<T>::nodo n, const T& e) {
173         assert(n>=0 && n < numNodos); //En esta implementación es fácil comprobar que n ∈ al
árbol
174         assert(nodos[n].hizq == NODO_NULO);
175         assert(numNodos < maxNodos); //Requerida por la implementación
176
177         //Se añade el nuevo nodo al final
178
179         nodos[n].hizq = numNodos;
180         nodos[numNodos].elto = e;
181         nodos[numNodos].padre = n;
182         nodos[numNodos].hizq = NODO_NULO;
183         nodos[numNodos].hder = NODO_NULO;
184         numNodos++;
185     }
186
187
188     template <typename T>
189     void Abin<T>::insertarHijoDrchoB(Abin<T>::nodo n, const T& e) {
190         assert(n>=0 && n < numNodos); //En esta implementación es fácil comprobar que n ∈ al
árbol
191         assert(nodos[n].hder == NODO_NULO);
192         assert(numNodos < maxNodos); //Requerida por la implementación
193
194         //Se añade el nuevo nodo al final
195
196         nodos[n].hder = numNodos;
197         nodos[numNodos].elto = e;
198         nodos[numNodos].padre = n;
199         nodos[numNodos].hizq = NODO_NULO;
200         nodos[numNodos].hder = NODO_NULO;
201         numNodos++;
202     }
203
204
205     template <typename T>
206     void Abin<T>::eliminarHijoIzqdoB(Abin<T>::nodo n) {
207         assert(n>= 0 && n < numNodos); //Comprobamos que el nodo ∈ al árbol
208         nodo hizqdo = nodos[n].hizq;
209         assert(hizqdo != NODO_NULO); //Comprobamos que hay nodo que eliminar
210         assert(nodos[hizqdo].hizq == NODO_NULO && nodos[hizqdo].hder == NODO_NULO);
//Comprobamos que sea nodo hoja
211
212         if(hizqdo != numNodos-1){ // Si no es el último (hay que mover el último a la posición de n
213
214             nodos[hizqdo] = nodos[numNodos-1]; //Se copia la estructura celda entera de la
última posición a esa
215
216             //Ahora hay que modificar los padres e hijos del nodo que hemos movido de la última
posición a esa
217
218             //Modificando los hijos
219             if(nodos[hizqdo].hder != NODO_NULO){
220                 nodos[nodos[hizqdo].hder].padre = hizqdo;
221             }
222
223
224

```

```

225
226     if(nodos[hizqdo].hizq != NODO_NULO){
227         nodos[nodos[hizqdo].hizq].padre = hizqdo;
228     }
229
230     //Modificando el padre
231     //Compruebo si era hijo izq o dcho del padre
232     if(nodos[nodos[hizqdo].padre].hizq == numNodos-1){ //Si es numNodos-1 es que era hizq.
233         nodos[nodos[hizqdo].padre].hizq = hizqdo;
234     }else{ //Si no era hijo izqdo, entonces era hijo derecho
235         nodos[nodos[hizqdo].padre].hder = hizqdo;
236     }
237
238 }
239     nodos[hizqdo].padre.hizq = NODO_NULO; //En cualquier caso, se le indica al nodo padre
240     que ya no tiene hijo izquierdo
241     numNodos--; //Indicamos el nuevo final de los datos
242 }
243
244 template <typename T>
245 void Abin<T>::eliminarHijoDrchoB(Abin<T>::nodo n){
246     assert(n>= 0 && n < numNodos); //Comprobamos que el nodo es al árbol
247     nodo hdrcho = nodos[n].hder;
248     assert(hdrcho != NODO_NULO); //Comprobamos que hay nodo que eliminar
249     assert(nodos[hdrcho].hizq == NODO_NULO && nodos[hdrcho].hder == NODO_NULO);
250     //Comprobamos que sea nodo hoja
251
252     if(hdrcho != numNodos-1){ // Si no es el último (hay que mover el ultimo a la posición de n
253
254         nodos[hdrcho] = nodos[numNodos-1]; //Se copia la estructura celda entera de la
255         última posición a esa
256
257         //Ahora hay que modificar los padres e hijos del nodo que hemos movido de la última
258         //posición a esa
259
260         //Modificando los hijos
261         if(nodos[hdrcho].hder != NODO_NULO){
262             nodos[nodos[hdrcho].hder].padre = hdrcho;
263         }
264
265         if(nodos[hdrcho].hizq != NODO_NULO){
266             nodos[nodos[hdrcho].hizq].padre = hdrcho;
267         }
268
269         //Modificando el padre
270         //Compruebo si era hijo izq o dcho del padre
271         if(nodos[nodos[hdrcho].padre].hizq == numNodos-1){ //Si es numNodos-1 es que era hizq.
272             nodos[nodos[hdrcho].padre].hizq = hdrcho;
273         }else{ //Si no era hijo izqdo, entonces era hijo derecho
274             nodos[nodos[hdrcho].padre].hder = hdrcho;
275         }
276
277     }
278     nodos[hdrcho].padre.hder = NODO_NULO; //En cualquier caso, se le indica al nodo padre
279     que ya no tiene hijo izquierdo
280     numNodos--; //Indicamos el nuevo final de los datos
281 }
282
283
284 template <typename T>
285 inline void Abin<T>::eliminarRaizB(){
286     assert(numNodos == 1); //Compruebo que solo queda la raiz
287
288     numNodos = 0; //Indico que se ha borrado el elemento de la posición 0 y que lo que haya
289     ahí es basura
290 }
291
292 template <typename T>
293 inline Abin<T>::~Abin(){
294     delete[] nodos;
295 }
296
297 template <typename T>
298 inline bool Abin<T>::arbolVacioB() const{

```

```

298     return (numNodos == 0);
299 }
300
301 template <typename T>
302 inline const T& Abin<T>::elemento(Abin<T>::nodo n) const{
303     assert(n>=0 && n < numNodos);
304     return nodos[n].elto;
305 }
306
307 template <typename T>
308 inline T& Abin<T>::elemento(Abin<T>::nodo n) {
309     assert(n>=0 && n < numNodos);
310     return nodos[n].elto;
311 }
312
313 template <typename T>
314 inline typename Abin<T>::nodo Abin<T>::raizB() const{
315     return (numNodos>0) ? 0 : NODO_NULO;
316 }
317
318 template <typename T>
319 inline typename Abin<T>::nodo Abin<T>::padreB(Abin<T>::nodo n) const{
320     assert(n>=0 && n < numNodos);
321
322     return nodos[n].padre; //Si n es 0, el campo padre será NODO_NULO
323 }
324
325 template <typename T>
326 inline typename Abin<T>::nodo Abin<T>::hijoIzqdoB(Abin<T>::nodo n) const{
327     assert(n>=0 && n < numNodos);
328     return nodos[n].hizq;
329 }
330
331 template <typename T>
332 inline typename Abin<T>::nodo Abin<T>::hijoDrchoB(Abin<T>::nodo n) const{
333     assert(n>=0 && n < numNodos);
334     return nodos[n].hder;
335 }
336
337
338
339 #endif // Abinvec
340

```

# Práctica 5: Ej 4

# QuesoViejo\_

```
1 #include "Ej4_AbinVec.h" → El .h anterior
2 #include <fstream>
3 #include <iostream>
4 #include "abin_E-S.h"
5
6 using namespace std;
7
8 int main()
9 {
10     Abin<char> A(40);
11
12     cout << "\n*** Lectura de árbol binario fichero de datos ***\n";
13     ifstream fe("NUMS.dat"); // abrir fichero de entrada
14     llenarAbin(fe, A); // desde fichero
15     fe.close();
16     cout << "\n*** Mostrar árbol binario B ***\n";
17     imprimirAbin(A); // en std::cout
18
19     cout<<endl;
20     int res;
21     res = A.CalcAlturaNodoB(A.raizB());
22     cout<<"La altura del nodo raiz es: "<<res<<endl;
23
24     typename Abin<char>::nodo p;
25     p = A.raizB();
26
27     res = A.CalcProfundidadNodoB(p);
28     cout<<"La profundidad del nodo raiz es: "<<res<<endl;
29
30
31
32
33
34
35
36 }
37
```

# Práctica 5: Ej 5 (Fichero.h)

# QuesoViejo\_

```
1  #ifndef AbinDin
2  #define AbinDin
3
4  #include <cassert>
5
6  template <typename T>
7  class Abin{
8      struct celda;
9  public:
10     typedef celda* nodo;
11
12     static const nodo NODO_NULO; //Se declara la constante NODO_NULO que después se
13     inicializará
14
15     Abin(); //Ctor vacío
16     Abin(const Abin<T>& a);
17     Abin<T>& operator=(const Abin<T>& a);
18
19     void insertarRaizB(const T& e);
20     void insertarHijoIzqdoB(nodo n, const T& e);
21     void insertarHijoDrchoB(nodo n, const T& e);
22     void eliminarHijoIzqdoB(nodo n);
23     void eliminarHijoDrchoB(nodo n);
24     void eliminarRaizB();
25
26
27     bool arbolVacioB() const;
28     const T& elemento(nodo n) const;
29     T& elemento(nodo n);
30     nodo raizB() const;
31     nodo padreB(nodo n) const;
32     nodo hijoIzqdoB(nodo n) const;
33     nodo hijoDrchoB(nodo n) const;
34
35     int CalcAlturaNodoB(nodo n);
36     int CalcProfundidadNodoB(nodo n);
37
38     ~Abin();
39
40
41 private:
42     struct celda{
43         T elto;
44         nodo padre, hizq, hder;
45         celda(const T& e, nodo p = NODO_NULO): elto(e), padre(p), hizq(NODO_NULO),
46         hder(NODO_NULO) {}
47     };
48     nodo r; //A partir de él accedemos al árbol
49
50     void destruirNodos(nodo& n); //Se carga el nodo n y todos sus descendientes de manera
51     //recursiva
52     nodo copiar(nodo n); //Copia el nodo n y todos sus descendientes de manera recursiva
53     int Max(int n1, int n2);
54 };
55
56
57 template <typename T>
58 const typename Abin<T>::nodo Abin<T>::NODO_NULO(0);
59
60
61 template <typename T>
62 void Abin<T>::destruirNodos(Abin<T>::nodo& n) {
63
64     if(n != NODO_NULO) {
65         destruirNodos(n->hizq);
66         destruirNodos(n->hder);
67         delete n;
68         n = NODO_NULO; //Delete lo deja en estado indefinido. Para identificar que está
69         //vacío, tenemos definido NODO_NULO
70     }
71 }
72
73 template <typename T>
74 typename Abin<T>::nodo Abin<T>::copiar(Abin<T>::nodo n) {
75     //Este método devuelve una copia del nodo n, es decir, se copia en otra posición de
76     //memoria.
```

```

75
76     nodo m = NODO_NULO;
77     if(n != NODO_NULO) {
78         m = new celda(n->elto); //m->padre por defecto inicializado a nulo, al igual que los
79         //hijos
80
81         m->hder = copiar(n->hder); //Esto se hace fuera en vez de dentro del if simplemente
82         //para que la llamada recursiva pare en
83         //el caso base definido.
84         if(m->hder != NODO_NULO) {
85
86             m->hder->padre = m;
87
88             m->hizq = copiar(n->hizq);
89             if(m->hizq != NODO_NULO) {
90
91                 m->hizq->padre = m;
92
93             }
94
95         return m;
96     }
97 }
98
99 template <typename T>
100 inline Abin<T>::Abin() : r(NODO_NULO) {}
101
102 /*-----NUEVOS
103 MÉTODOS-----*/
104 template <typename T>
105 int Abin<T>::CalcAlturaNodoB(typename Abin<T>::nodo n)
106 {
107     if(n == NODO_NULO)
108     {
109         return -1;
110     }
111     else
112     {
113         return 1+Max(CalcAlturaNodoB(n->hizq),CalcAlturaNodoB(n->hder));
114     }
115 }
116
117 template <typename T>
118 int Abin<T>::CalcProfundidadNodoB(typename Abin<T>::nodo n)
119 {
120     if(n == NODO_NULO)
121     {
122         return -1;
123     }
124     else
125     {
126         return 1+CalcProfundidadNodoB(n->padre);
127     }
128 }
129
130
131
132 template <typename T>
133 int Abin<T>::Max(int n1, int n2)
134 {
135     return (n1 > n2) ? n1 : n2;
136 }
137 /*-----FIN NUEVOS
138 MÉTODOS-----*/
139
140 template <typename T>
141 Abin<T>::Abin(const Abin<T> & a) {
142     r = copiar(a.r);
143 }
144
145 template <typename T>
146 Abin<T>& Abin<T>::operator=(const Abin<T> & a) {
147
148     if(this != &a){ //Para evitar autoasignación
149         this->~Abin(); //Se destruye el árbol actual
150         this = copiar(a.r);
151     }
152 }

```

```

150     }
151
152     return *this;
153 }
154
155 template <typename T>
156 inline void Abin<T>::insertarRaizB(const T& e) {
157     assert (r == NODO_NULO);
158     r = new celda(e);
159 }
160
161 template <typename T>
162 inline void Abin<T>::insertarHijoIzqdoB(Abin<T>::nodo n, const T& e) {
163     assert(n != NODO_NULO);
164     assert (n->hizq == NODO_NULO);
165
166     n->hizq = new celda(e, n);
167 }
168
169 template <typename T>
170 inline void Abin<T>::insertarHijoDrchoB(Abin<T>::nodo n, const T& e) {
171     assert(n != NODO_NULO);
172     assert (n->hder == NODO_NULO);
173
174     n->hder = new celda(e, n);
175 }
176
177 template <typename T>
178 inline void Abin<T>::eliminarHijoIzqdoB(Abin<T>::nodo n) {
179     assert(n != NODO_NULO)
180     assert (n->hizq != NODO_NULO);
181     assert(n->hizq->hizq == NODO_NULO && n->hizq->hder == NODO_NULO);
182     delete n->hizq;
183     n->hizq = NODO_NULO;
184 }
185
186 template <typename T>
187 inline void Abin<T>::eliminarHijoDrchoB(Abin<T>::nodo n) {
188     assert(n != NODO_NULO)
189     assert (n->hder != NODO_NULO);
190     assert(n->hder->hizq == NODO_NULO && n->hder->hder == NODO_NULO);
191     delete n->hder;
192     n->hder = NODO_NULO;
193 }
194
195 template <typename T>
196 inline void Abin<T>::eliminarRaizB() {
197     assert(r != NODO_NULO);
198     assert(r->hizq == NODO_NULO && r->hder == NODO_NULO);
199     delete r;
200     r = NODO_NULO;
201 }
202
203 template <typename T>
204 inline bool Abin<T>::arbolVacioB() const{
205     return (r == NODO_NULO);
206 }
207
208 template <typename T>
209 inline const T& Abin<T>::elemento(Abin<T>::nodo n) const{
210     assert(n != NODO_NULO);
211     return n->elto;
212 }
213
214 template <typename T>
215 inline T& Abin<T>::elemento(Abin<T>::nodo n) {
216     assert(n != NODO_NULO);
217     return n->elto;
218 }
219
220 template <typename T>
221 typename Abin<T>::nodo Abin<T>::raizB() const{
222     return r; //Si no existe devuelve NODO_NULO que según la precondition
223 }
224
225 template <typename T>
226 typename Abin<T>::nodo Abin<T>::padreB(Abin<T>::nodo n) const{
227     assert(n != NODO_NULO);
228     return n->padre; //Si no existe es NODO_NULO que era la precondition

```

```
229     }
230
231     template <typename T>
232     typename Abin<T>::nodo Abin<T>::hijoIzqdoB(Abin<T>::nodo n) const{
233         assert(n != NODO_NULO);
234         return n->hizq;
235     }
236
237     template <typename T>
238     typename Abin<T>::nodo Abin<T>::hijoDrchoB(Abin<T>::nodo n) const{
239         assert(n != NODO_NULO);
240         return n->hder;
241     }
242
243     template <typename T>
244     Abin<T>::~Abin() {
245         destruirNodos(r); //El trabajo lo hace la otra función
246     }
247
248
249
250
251
252 #endif // AbinDin
253
```

# Práctica 5: Ej 5

# QuesoViejo\_

```
1  #include "Ej5_AbinDin.h" -> El .h anterior
2  #include <fstream>
3  #include <iostream>
4  #include "abin_E-S.h"
5
6  using namespace std;
7
8  int main()
9  {
10    Abin<char> A;
11
12    cout << "\n*** Lectura de árbol binario fichero de datos ***\n";
13    ifstream fe("NUMS.dat"); // abrir fichero de entrada
14    rellenarAbin(fe, A); // desde fichero
15    fe.close();
16    cout << "\n*** Mostrar árbol binario B ***\n";
17    imprimirAbin(A); // en std::cout
18
19    cout<<endl;
20    int res;
21    res = A.CalcAlturaNodoB(A.raizB());
22    cout<<"La altura del nodo raiz es: "<<res<<endl;
23
24    typename Abin<char>::nodo p;
25    p = A.raizB();
26
27    res = A.CalcProfundidadNodoB(A.hijoDrchoB(A.hijoDrchoB(A.hijoDrchoB(p)) ));
28    cout<<"La profundidad del hijo del hijo del nodo raiz es: "<<res<<endl;
29
30
31
32
33
34
35
36  }
37
```

## Práctica 3: Ej6

## QuesoViejo\_

```
1  #include "AbinDin.h"
2  #include <iostream>
3  #include <fstream>
4  #include "abin_E-S.h"
5
6
7  template <typename T>
8  int CalcDesequilibrio (const Abin<T>& A);
9
10 template <typename T>
11 int CalcDesequilibrio_rec (typename Abin<T>::nodo n, const Abin<T>& A);
12
13 template <typename T>
14 int CalcAlturaNodo_rec (typename Abin<T>::nodo n, const Abin<T>& A);
15
16 //template <typename T>
17
18 int Abs (int n);
19 int Max (int n1, int n2, int n3);
20 int Max (int n1, int n2);
21
22
23
24
25 using namespace std;
26
27 int main()
28 {
29
30     Abin<char> A;
31
32     cout << "\n*** Lectura de árbol binario fichero de datos ***\n";
33     ifstream fe("NUMS.dat"); // abrir fichero de entrada
34     llenarAbin(fe, A); // desde fichero
35     fe.close();
36     cout << "\n*** Mostrar árbol binario B ***\n";
37     imprimirAbin(A); // en std::cout
38     cout<<endl;
39
40     int resultado;
41     resultado = CalcDesequilibrio (A);
42
43     cout<<"Desequilibrio arbol A: "<<resultado<<endl;
44
45 }
46
47
48
49 template <typename T>
50 int CalcDesequilibrio (const Abin<T>& A)
51 {
52     return CalcDesequilibrio_rec (A.raizB(), A);
53 }
54
55
56 template <typename T>
57 int CalcDesequilibrio_rec (typename Abin<T>::nodo n, const Abin<T>& A)
58 {
59     if (n == Abin<T>::NODO_NULO)
60     {
61         return 0; // La altura de los subárboles de un nodo nulo es la misma, por lo que su
diferencia es 0
62         //La altura de un arbol de un solo nodo es 0 también
63     }
64     else
65     {
66         int Deseq = Abs (CalcAlturaNodo_rec (A.hijoIzqdoB(n), A) -
CalcAlturaNodo_rec (A.hijoDrchoB(n), A));
67         //Se aprovecha que NODO_NULO es -1 de forma que si tengo un subárbol []
aunque la altura del nodo que existe es 0, la del otro es -1 y queda desequilibrio 1
68         //
69
70         return Max (Deseq, CalcDesequilibrio_rec (A.hijoIzqdoB(n), A),
CalcDesequilibrio_rec (A.hijoDrchoB(n), A));
71         //Se devuelve hacia arriba el máximo del desequilibrio de este subárbol, el del
subárbol izq y del derecho
72     }
73 }
74 }
```

```

75
76 template <typename T>
77 int CalcAlturaNodo_rec (typename Abin<T>::nodo n, const Abin<T>& A)
78 {
79     if(n == Abin<T>::NODO_NULO)
80     {
81         return -1;
82     }
83     else
84     {
85         return 1+Max(CalcAlturaNodo_rec(A.hijoIzqdoB(n),A),
CalcAlturaNodo_rec(A.hijoDrchoB(n),A));
86     }
87 }
88 }
89
90 int Max (int n1, int n2)
91 {
92
93     return (n1 > n2) ? n1 : n2;
94 }
95
96 }
97
98 int Max (int n1, int n2, int n3)
99 {
100     int res;
101
102     res = (n1 > n2) ? n1 : n2;
103     res = (res > n3) ? res : n3;
104
105     return res;
106 }
107 }
108
109 int Abs (int n)
110 {
111     return (n >= 0) ? n : -n;
112 }
113
114

```

## Práctica 3: Ej 7

```

1  #include "AbinDin.h"
2  #include <iostream>
3  #include <fstream>
4  #include "abin_E-S.h"
5
6
7
8
9
10 template <typename T>
11 bool CalcPseudoCompletoArbol (const Abin<T>& A);
12
13
14
15 template <typename T>
16 bool CalcPseudoCompleto_rec (typename Abin<T>::nodo n, const Abin<T>& A, int prof, int
penultimo);
17
18 template <typename T>
19 int CalcAlturaNodo_rec (typename Abin<T>::nodo n, const Abin<T>& A);
20
21 int Max (int n1, int n2);
22
23
24 using namespace std;
25
26 int main()
27 {
28
29
30     Abin<char> A;
31
32     cout << "\n*** Lectura de árbol binario fichero de datos ***\n";
33     ifstream fe("NUMS.dat"); // abrir fichero de entrada
34     llenarAbin(fe, A); // desde fichero
35     fe.close();
36     cout << "\n*** Mostrar árbol binario B ***\n";
37     imprimirAbin(A); // en std::cout
38     cout << endl;
39
40     bool pseudoc;
41     pseudoc = CalcPseudoCompletoArbol (A);
42
43     cout << " pseudoc " << pseudoc << " La altura del nodo raiz es:
" << CalcAlturaNodo_rec (A.raizB(), A) << endl;
44     if (pseudoc)
45     {
46         cout << "El arbol es pseudocompleto" << endl;
47     }
48     else
49     {
50         cout << "El arbol no es pseudocompleto" << endl;
51     }
52
53 }
54
55
56
57
58
59
60 template <typename T>
61 bool CalcPseudoCompletoArbol (const Abin<T>& A)
62 {
63     int altura;
64     altura = CalcAlturaNodo_rec (A.raizB(), A);
65     return CalcPseudoCompleto_rec (A.raizB(), A, 0, altura-1);
66
67 }
68
69 template <typename T>
70 bool CalcPseudoCompleto_rec (typename Abin<T>::nodo n, const Abin<T>& A, int prof, int
penultimo)
71 {
72     if (n == Abin<T>::NODO_NULO)
73     {
74         return true;
75     }
76     else

```

```

77     {
78         bool resultado;
79         if(prof == penultimo)
80         {
81
82             if( (A.hijoIzqdoB(n) == Abin<T>::NODO_NULO && A.hijoDrchoB(n) ==
83 Abin<T>::NODO_NULO) ||
84                 (A.hijoIzqdoB(n) != Abin<T>::NODO_NULO && A.hijoDrchoB(n) !=
85 Abin<T>::NODO_NULO) )
86             {
87                 resultado = true;
88             }
89             else
90             {
91                 resultado = false;
92             }
93         else
94         {
95             resultado = true;
96         }
97
98         return( ( resultado && CalcPseudoCompleto_rec(A.hijoIzqdoB(n),A,prof+1,penultimo) )
99             && CalcPseudoCompleto_rec(A.hijoDrchoB(n),A,prof+1,penultimo) );
100            //En el momento que un "resultado" sea false, todo será false e irá subiéndolos hacia
101            //arriba del árbol
102        }
103
104
105
106 template <typename T>
107 int CalcAlturaNodo_rec(typename Abin<T>::nodo n, const Abin<T>& A)
108 {
109     if(n == Abin<T>::NODO_NULO)
110     {
111         return -1;
112     }
113     else
114     {
115         return 1+Max(CalcAlturaNodo_rec(A.hijoIzqdoB(n),A),
116                     CalcAlturaNodo_rec(A.hijoDrchoB(n),A));
117     }
118 }
119
120
121 int Max(int n1, int n2)
122 {
123     return (n1 > n2) ? n1 : n2;
124 }
125

```

## Práctica 2: Ej 5

## QuesoViejo\_

```
1  #include "AbinDin.h"
2  #include <iostream>
3  #include <fstream>
4  #include "abin_E-s.h"
5
6
7  template <typename T>
8  bool Similares(const Abin<T>& A, const Abin<T>& B);
9
10 template <typename T>
11 bool SimilaresNodo_rec(typename Abin<T>::nodo n1, typename Abin<T>::nodo n2, const Abin<T>&
A, const Abin<T>& B);
12
13 using namespace std;
14
15 int main()
16 {
17     Abin<char> A;
18     Abin<char> B;
19
20     cout << "\n*** Lectura de árbol binario fichero de datos ***\n";
21     ifstream fe("NUMS.dat"); // abrir fichero de entrada
22     llenarAbin(fe, A); // desde fichero
23     fe.close();
24     cout << "\n*** Mostrar árbol binario A ***\n";
25     imprimirAbin(A); // en std::cout
26
27     cout << "\n*** Lectura de árbol binario fichero de datos ***\n";
28     ifstream fe2("DAT.dat"); // abrir fichero de entrada
29     llenarAbin(fe2, B); // desde fichero
30     fe2.close();
31     cout << "\n*** Mostrar árbol binario B ***\n";
32     imprimirAbin(B); // en std::cout
33
34     bool resultado;
35     resultado = Similares(A, B);
36
37     cout<<endl<<endl;
38     if(resultado)
39     {
40         cout<<"Los arboles son similares"<<endl;
41     }
42     else
43     {
44         cout<<"Los arboles no son similares"<<endl;
45     }
46
47 }
48
49
50 template <typename T>
51 bool Similares(const Abin<T>& A, const Abin<T>& B)
52 {
53     return SimilaresNodo_rec(A.raizB(), B.raizB(), A, B);
54
55
56 }
57
58 template <typename T>
59 bool SimilaresNodo_rec(typename Abin<T>::nodo n1, typename Abin<T>::nodo n2, const Abin<T>&
A, const Abin<T>& B)
60 {
61     if(n1 == Abin<T>::NODO_NULO && n2 == Abin<T>::NODO_NULO)
62     {
63         return true;
64     }
65     else if( (n1 == Abin<T>::NODO_NULO && n2 != Abin<T>::NODO_NULO ) || (n1 != Abin<T>::NODO_NULO && n2 == Abin<T>::NODO_NULO) )
66     {
67         //Podría comprobar que solo uno sea NODO_NULO porque ya se ha hecho la primera comprobación
68         return false;
69     }
70     else
71     {
72         return ( SimilaresNodo_rec(A.hijoIzqdoB(n1), B.hijoIzqdoB(n2), A, B) && SimilaresNodo_rec(A.hijoDrchoB(n1), B.hijoDrchoB(n2), A, B) );
73     }
74 }
```

75  
76  
77

## Práctica 2: Ej 2

```

1  #include "AbinDin.h"
2  #include <iostream>
3  #include <fstream>
4  #include "abin_E-S.h"
5
6
7
8
9  template <typename T>
10 Abin<T> ArbolReflejado (const Abin<T>& A);
11
12 template <typename T>
13 void ReflejoNodo_rec (typename Abin<T>::nodo na, const Abin<T>& A, typename Abin<T>::nodo nb,
14 Abin<T>& B);
15 using namespace std;
16
17
18 int main()
19 {
20
21     Abin<char> A;
22     Abin<char> B;
23     B.insertarRaizB ('a');
24
25     cout << "\n*** Lectura de árbol binario fichero de datos ***\n";
26     ifstream fe ("NUMS.dat"); // abrir fichero de entrada
27     llenarAbin (fe, A); // desde fichero
28     fe.close();
29     cout << "\n*** Mostrar árbol binario A ***\n";
30     imprimirAbin (A); // en std::cout
31
32     B = ArbolReflejado (A);
33     cout << "Nuevo arbol:" << endl;
34     imprimirAbin (B);
35
36
37
38 }
39
40 template <typename T>
41 Abin<T> ArbolReflejado (const Abin<T>& A)
42 {
43     Abin<T> B; //Crea árbol vacío
44     if (A.raizB () != Abin<T>::NODO_NULO)
45     {
46
47
48     B.insertarRaizB (A.elemento (A.raizB ()));
49     ReflejoNodo_rec (A.raizB (), A, B.raizB (), B);
50
51     }
52
53     return B;
54
55 }
56
57 template <typename T>
58 void ReflejoNodo_rec (typename Abin<T>::nodo na, const Abin<T>& A, typename Abin<T>::nodo nb,
59 Abin<T>& B)
60 {
61
62     if (A.hijoIzqdoB (na) != Abin<T>::NODO_NULO) //Debido a que la precondition de la
63     //operación elemento es que no sea NODO_NULO, //No puedo esperar a que na sea
64     //NODO_NULO, sino que compruebo los hijos del mismo
65     {
66         B.insertarHijoDrchoB (nb, A.elemento (A.hijoIzqdoB (na)));
67         ReflejoNodo_rec (A.hijoIzqdoB (na), A, B.hijoDrchoB (nb), B);
68     }
69
70     if (A.hijoDrchoB (na) != Abin<T>::NODO_NULO)
71     {
72         B.insertarHijoIzqdoB (nb, A.elemento (A.hijoDrchoB (na)));
73         ReflejoNodo_rec (A.hijoDrchoB (na), A, B.hijoIzqdoB (nb), B);
74     }
75 }
```

76  
77      }  
78

# Práctica 2: Ej 3

# QuesoViejo\_

```
1 #include "AbinDin.h"
2 #include <iostream>
3 #include <fstream>
4 #include "abin_E-S.h"
5 #include <cstdlib>
6 #include <string>
7
8
9
10 /*NOTA: LO MISMO LA FUNCIÓN STOI NO TE TIRA. ASEGURATE DE TENER UN COMPILADOR ACTUAL. YO HE
11 COMPILADO EN UBUNTU
12 USANDO UN MAKEFILE
13
14 YO TE DIGO QUE FUNCIONA, QUE TU COMPILADOR LO COMPILE O NO ES OTRA COSA
15
16 MAKEFILE UTILIZADO (GENERA UN EJECUTABLE LLAMADO "program"):
17
18 STD = c++11
19 CXX = c++
20 CXXFLAGS = -g -Wall -std=$(STD) -pedantic
21
22
23 program: Ejercicio3.o
24     $(CXX) -o program Ejercicio3.o $(CXXFLAGS)
25
26 Ejercicio3.o: Ejercicio3.cpp AbinDin.h abin_E-S.h
27     $(CXX) -c Ejercicio3.cpp $(CXXFLAGS)
28
29
30 */
31
32 using namespace std;
33
34
35 int EvaluarExpresion(const Abin<string>& A);
36 int Evaluar_rec(typename Abin<string>::nodo n, const Abin<string>& A);
37
38
39
40
41 int main()
42 {
43
44     Abin<string> A;
45
46     string fin = "#";
47
48     cout << "*** Lectura del árbol binario A ***\n";
49     llenarAbin(A, fin); // desde std::cin
50     imprimirAbin(A); // en pantalla
51
52
53     int resultado;
54     resultado = EvaluarExpresion(A);
55
56     cout<<"El resultado es: "<<resultado<<endl;
57
58
59 }
60
61
62
63
64 int EvaluarExpresion(const Abin<string>& A)
65 {
66
67     return Evaluar_rec(A.raizB(), A);
68
69 }
70
71 int Evaluar_rec(typename Abin<string>::nodo n, const Abin<string>& A)
72 {
73     string actual;
74     actual = A.elemento(n);
75     //string suma = "+", resta = "-", mult = "*", division = "/";
76
77     if(actual == "+") { return Evaluar_rec(A.hijoIzqdoB(n), A) +
Evaluar_rec(A.hijoDrchoB(n), A); }
```

```
78     else if(actual == "-") { return Evaluar_rec(A.hijoIzqdoB(n),A) -  
79     else if(actual == "*") { return Evaluar_rec(A.hijoIzqdoB(n),A) *  
80     else if(actual == "/") { return Evaluar_rec(A.hijoIzqdoB(n),A) /  
81     else{ return stoi(actual); } //Para pasar de char al valor numérico correspondiente  
82  
83  
84 }  
85  
86  
87
```

# Práctica 2: Ej 4 (Síghero.h)

# QuesoViejo\_

```
1  #ifndef ARBOLGRANDEBIENGRANDOTE
2
3  #define ARBOLGRANDEBIENGRANDOTE
4
5  #include "ColaDin.h"
6
7
8  #include <cassert>
9  #include <cmath>
10 #include <cstdlib> //Para el size_t
11 template <typename T> class Abin{
12
13 public:
14     typedef int nodo;
15     static const nodo NODO_NULO;
16
17
18
19     explicit Abin(size_t maxNodos, const T& e_nulo = T());
20     Abin(const Abin<T>& A);
21     Abin<T>& operator =(const Abin<T>& A);
22
23     void insertarRaizB( const T& e);
24     void insertarHijoIzqdoB(nodo n, const T& e);
25     void insertarHijoDrchoB(nodo n, const T& e);
26     void eliminarHijoIzqdoB(nodo n);
27     void eliminarHijoDrchoB(nodo n);
28
29
30
31     bool arbolVacioB() const;
32     void eliminarRaizB();
33     const T& elemento(nodo n) const;
34     T& elemento(nodo n);
35
36     nodo raizB() const;
37     nodo padreB(nodo n) const;
38     nodo hijoIzqdoB(nodo n) const;
39     nodo hijoDrchoB(nodo n) const;
40
41     int Profundidad(nodo n) const;
42
43
44     ~Abin();
45
46
47
48
49
50 private:
51
52
53     int maxNodos;
54     T ELTO_NULO;
55     T* nodos; //Vector de elementos T | No tengo que almacenar padre, hiza... porque vienen
determinados por la posición
56
57
58
59 };
60
61 template <typename T>
62 const typename Abin<T>::nodo Abin<T>::NODO_NULO(-1);
63
64
65 template <typename T>
66 Abin<T>::Abin(size_t maxNodos, const T& e_nulo) : maxNodos(maxNodos), ELTO_NULO(e_nulo),
nodos(new T[maxNodos])
67 {
68
69     for(nodo i = 0 ; i < maxNodos ; i++)
70     {
71         nodos[i] = ELTO_NULO;
72     }
73 }
74
75 template <typename T>
76 Abin<T>::Abin(const Abin<T>& A) : maxNodos(A.maxNodos), ELTO_NULO(A.ELTO_NULO), nodos(new
T[A.maxNodos])
```



```

149         return prof;
150     }
151     else
152     {
153         if(n > centro)
154         {
155             inf = centro+1; //Acoto el intervalo
156         }
157         else
158         {
159             sup = centro-1; //Acoto el intervalo
160         }
161     }
162 }
163 }
164
165
166
167 }
168
169 /*-----*/
-----*/
170
171 template <typename T>
172 inline void Abin<T>::insertarRaizB(const T& e)
173 {
174     assert(nodos[maxNodos/2] == ELTO_NULO);
175     nodos[maxNodos/2] = e;
176     //std::cout<<"Raiz insertada en: "<<maxNodos/2<<std::endl;
177
178 }
179
180
181
182
183 /**Estas funciones con llamadas a profundidad en los assert no son inline.
184 Rodaría no incluir los assert, pero prefiero hacerlo así que ya bastante me lío con esta
185 implementación del TAD **/
186 template <typename T>
187 void Abin<T>::insertarHijoIzqdoB(typename Abin<T>::nodo n, const T& e)
188 {
189     assert(n >= 0 && n < maxNodos);
190     assert(n-(maxNodos+1)/std::pow(2, Profundidad(n)+2) > 0); //Solo se comprueba con el hijo
191     izquierdo porque este valor será menor que n
192     luego este valor es menor que maxNodos
193     assert(nodos[n] != Abin<T>::ELTO_NULO);
194     assert(nodos[ (int) (n-(maxNodos+1)/std::pow(2, Profundidad(n)+2) ) ] ==
195     Abin<T>::ELTO_NULO); //Calculado según la formula del enunciado,
196     es decir, de la única manera correcta
197
198     // std::cout<<"Insertado "<<e<<" en la posición: "<<(int) (
199     n-(maxNodos+1)/std::pow(2, Profundidad(n)+2) )<<std::endl;
200     nodos[ (int) (n-(maxNodos+1)/std::pow(2, Profundidad(n)+2) ) ] = e;
201 }
202
203 template <typename T>
204 void Abin<T>::insertarHijoDrchoB(typename Abin<T>::nodo n, const T& e)
205 {
206     assert(n >= 0 && n < maxNodos);
207     assert(n + (maxNodos+1)/std::pow(2, Profundidad(n)+2) < maxNodos);
208
209     assert(nodos[n] != ELTO_NULO);
210     assert(nodos[ (int) ( n + (maxNodos+1)/std::pow(2, Profundidad(n)+2) ) ] == ELTO_NULO);
211
212     //std::cout<<"Insertado "<<e<<" en la posición: "<<(int) (
213     n+(maxNodos+1)/std::pow(2, Profundidad(n)+2) )<<std::endl;
214
215     nodos[ (int) ( n + (maxNodos+1)/std::pow(2, Profundidad(n)+2) ) ] = e;
216 }
217
218 template <typename T>
219 void Abin<T>::eliminarHijoIzqdoB(Abin<T>::nodo n)
220 {
221     assert(n >= 0 && n < maxNodos);
222     assert(n - (maxNodos+1)/std::pow(2, Profundidad(n)+2) >= 0);

```

```

220
221 //assert(nodos[n] != ELTO_NULO); NO hago esta comprobación porque es inútil
222 assert(nodos[ (int) (n - (maxNodos+1) / std::pow(2, Profundidad(n)+2) ) ] != ELTO_NULO);
223
224 nodos[ (int) (n - (maxNodos+1) / std::pow(2, Profundidad(n)+2) ) ] = ELTO_NULO;
225
226
227 }
228
229 template <typename T>
230 void Abin <T>::eliminarHijoDrchoB(Abin <T>::nodo n)
231 {
232     assert(n >= 0 && n < maxNodos);
233     assert(n + (maxNodos+1) / std::pow(2, Profundidad(n)+2) >= 0);
234
235 //assert(nodos[n] != ELTO_NULO); NO hago esta comprobación porque es inútil
236 assert(nodos[ (int) (n + (maxNodos+1) / std::pow(2, Profundidad(n)+2) ) ] != ELTO_NULO);
237
238 nodos[ (int) (n + (maxNodos+1) / std::pow(2, Profundidad(n)+2) ) ] = ELTO_NULO;
239
240 }
241
242 template <typename T>
243 bool Abin <T>::arbolVacioB() const
244 {
245
246     return (nodos[maxNodos/2] == ELTO_NULO);
247
248 }
249
250 template <typename T>
251 void Abin <T>::eliminarRaizB()
252 {
253     nodo posRaiz = maxNodos/2; //Se hace para mayor claridad y legibilidad del código :-)
254     assert(nodos[posRaiz] != ELTO_NULO);
255     assert(nodos[ (int) (posRaiz - (maxNodos + 1) / std::pow(2, 2) ) ] == ELTO_NULO);
256     assert(nodos[ (int) (posRaiz + (maxNodos + 1) / std::pow(2, 2) ) ] == ELTO_NULO);
257
258     nodos[posRaiz] = ELTO_NULO;
259
260 }
261
262 template <typename T>
263 const T& Abin <T>::elemento(Abin <T>::nodo n) const
264 {
265
266     return nodos[n];
267 }
268
269 template <typename T>
270 T& Abin <T>::elemento(Abin <T>::nodo n)
271 {
272     return nodos[n];
273 }
274
275
276
277 template <typename T>
278 typename Abin <T>::nodo Abin <T>::raizB() const
279 {
280     return maxNodos/2;
281 }
282 template <typename T>
283 typename Abin <T>::nodo Abin <T>::padreB(typename Abin <T>::nodo n) const
284 {
285     assert(nodos[n] != ELTO_NULO);
286     nodo p;
287     p = Profundidad(n);
288
289 //Primero calculo si es hijo izquierdo o derecho
290     int exp = ( (maxNodos+1) / std::pow(2, p-1)); //Si no pongo esto así en vez de abajo se
291 //pone tontorrón
292     //std::cout<<"Valor izquierda expresion :"<< n % exp<<std::endl;
293
294     if ( n % exp == (int) (( (maxNodos+1) / std::pow(2, p+1) )-1) )
295     {
296         //n es hijo izquierdo
297         //std::cout<<"El nodo "<<n<<" es hijo izquierdo de su padre"<<std::endl;
298         return (nodo)(n + (maxNodos+1) / std::pow(2, p+1));
299     }
300 }
```

```

298     }
299     else
300     {
301
302         //std::cout<<"EL nodo "<<n<<" es hijo izquierdo de su padre"<<std::endl;
303
304         return (int)(n - (maxNodos+1)/std::pow(2, p+1));
305     }
306
307 }
308
309 template <typename T>
310 typename Abin<T>::nodo Abin<T>::hijoIzqdoB(typename Abin<T>::nodo n) const
311 {
312
313     //std::cout<<"Valor n en hijoIzqdoB: "<<n<<std::endl;
314     assert(n >= 0 && n < maxNodos);
315
316     assert(nodos[n] != ELTO_NULO);
317
318     nodo p;
319     p = Profundidad(n);
320     //std::cout<<"Profundidad de "<<n<<" es: "<<p<<std::endl;
321
322
323     int res;
324     res = (int)(n - (maxNodos+1)/std::pow(2, p+2));
325     //std::cout<<"      valor en nodos[hijoIzqdo] es: "<<nodos[res]<<std::endl;
326     //std::cout<<"      ELTO_NULO ES: "<<ELTO_NULO<<std::endl;
327
328     // std::cout<<"      IMP      valor comparacion: "<<std::endl;
329     if(res < 0 || res >= maxNodos || nodos[res] == ELTO_NULO)
330     {
331         //std::cout<<"Devuelve " <<NODO_NULO<<std::endl;
332
333         return NODO_NULO;
334
335     }
336     else
337     {
338         //std::cout<<"Devuelve NODO_NULO : "<<res<<std::endl;
339
340         return res;
341
342     }
343 }
344
345 template <typename T>
346 typename Abin<T>::nodo Abin<T>::hijoDrchoB(typename Abin<T>::nodo n) const
347 {
348
349     //std::cout<<"Valor n en hijoDrchoB: "<<n<<std::endl;
350     assert(n >= 0 && n < maxNodos);
351     assert(nodos[n] != ELTO_NULO);
352
353     nodo p;
354     p = Profundidad(n);
355     //std::cout<<"Profundidad de "<<n<<" es: "<<p<<std::endl;
356
357     int res;
358     res = (int)(n + (maxNodos+1)/std::pow(2, p+2));
359
360     if(res < 0 || res >= maxNodos || nodos[res] == ELTO_NULO )
361     {
362         //std::cout<<"Devuelve " <<res<<std::endl;
363
364         return NODO_NULO;
365
366     }
367     else
368     {
369         //std::cout<<"Devuelve NODO_NULO: "<<NODO_NULO<<std::endl;
370
371         return res;
372
373     }
374 }
375
376 template <typename T>

```

```
377     Abin<T>::~Abin()
378     {
379         delete[] nodos;
380     }
381
382
383
384
385
386 #endif // ARBOLGRANDEBIENGRANDOTE
388
```

## Práctica 2: Ej 4

→ Fichero .h de antes

```

1  #include "AbinEj4.h"
2  #include <fstream>
3  #include <iostream>
4  #include "abin_E-S.h"
5
6
7
8
9  using namespace std;
10
11
12 int main()
13 {
14
15     cout<<"holo"<<endl;
16     Abin<char> A(31, '-');
17
18
19
20
21     cout << "\n*** Lectura de árbol binario fichero de datos ***\n";
22     ifstream fe("DAT4.dat"); // abre fichero de entrada
23     llenarAbin(fe, A); // desde fichero
24     fe.close();
25     cout << "\n*** Mostrar árbol binario A ***\n";
26     imprimirAbin(A); // en std::cout
27
28
29     typename Abin<char>::nodo p = A.raizB();
30
31     cout<<"Profundidad raiz: "<<A.Profundidad(p)<<endl;
32     p = A.hijoIzqdoB(p);
33     cout<<"Profundidad hijo izqdo raiz: "<<A.Profundidad(p)<<endl;
34     cout<<"Valor p: "<<p<<endl;
35     p = A.padreB(p);
36     cout<<"Padre del hijo izqdo de raiz: "<<p<<endl;
37     p = A.hijoDrchoB(p);
38     cout<<"Profundidad hijo drcho raiz: "<<A.Profundidad(p)<<endl;
39     p = A.padreB(p);
40 }
41
42
43
44
45
46
47

```