6. Estructuras de datos no lineales. Arboles

6.1 Introducción

En este tema abordaremos contenedores en la que el conjunto de datos no se disponen linealmente (uno detrás de otro). En este contexto los datos se podrán disponer de forma jerárquica, o en una estructura donde los elementos se pueden conectar unos con otros sin restricciones como sería en un grafo o red. Con estos tipos de datos podremos abordar problemas donde sea más eficiente la búsqueda, la relaciones de orden entre los datos se expresan con más posibilidades: ordenes totales o parciales. O simplemente podemos representar la realidad de nuestro problema con una mayor abstracción como sería por ejemplo una red de ordenadores con objeto de encontrar el mejor camino para enrutar los paquetes de información.

Presentada las posibilidad de las estructuras de datos no lineales primer lugar abordaresmo las estructuras jerárquicas.

6.2 Estructura de datos jerárquica: árboles

Desde el punto de vista de la teoría de grafos, definimos un **árbol** , como un grafo acíclico donde cada nodo tiene grado de entrada[1] 1 (excepto el nodo raíz que tiene grado de entrada 0) y el grado de salida[2] 0 o mayor que cero (un ejemplo de árbol en la figura 6.1).

Un árbol se compone de *nodos*. Hay tres tipos de nodos:

1. *raíz*: no tiene padre, es el nodo que está en la parte superior de la jerarquía.
2. *hoja*: no tienen hijos, son los nodos que están en la parte inferior de la jerarquía.

[1] número de líneas que entran al nodo

[2] número de líneas que salen de un nodo. Las hojas tienen grado de salida 0

3. *interiores*: el resto de nodos.

Algunas características de los árboles son:

1. Todos los nodos descienden de la raíz.
2. Los descendientes directos se llaman hijos
3. Los nodos del mismo nivel y que descienden del mismo padre son hermanos.
4. Y los padres de los padres de un nodo son los ancestros de éste.

Un ejemplo de árbol sería

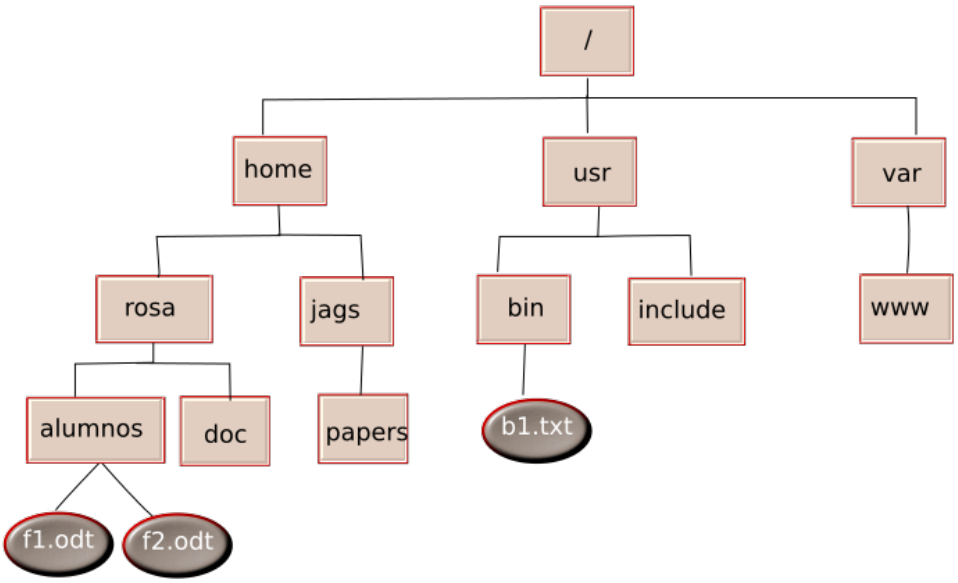


Figura 6.1: Ejemplo de árbol: estructura de directorios

6.2.1 Conceptos

árbol n-ario : se caracteriza porque todos los nodos tienen 0 ó n hijos. Por ejemplo, un árbol 3-ario tiene 0 ó 3 hijos únicamente. Un árbol 2-ario tiene 0 ó 2 hijos, pero uno binario puede tener 0, 1 ó 2 hijos.

camino en un árbol : es una sucesión de nodos $n_1, n_2, \cdots, n_k$ donde el nodo i-ésimo (n_i) es padre del nodo $i+1$ (n_{i+1}). La longitud del camino es igual al número de nodos menos uno. En la figura ejemplo 6.1, un camino podría ser:

$$/ \qquad home \qquad rosa \qquad alumnos \qquad f1.odt$$

Donde *rosa* sería padre de *alumnos* y la longitud del camino sería 4.

ancestro : el nodo n_i es ancestro del nodo n_j si existe un camino desde n_i tal que n_i se coloca en el camino delante de n_j. Por ejemplo, en el camino:

$$n_s \cdots n_j \cdots n_i \cdots n_l$$

n_j es ancestro de n_i porque está antes en el camino.

descendiente : n_i es descendiente de n_j si existe un camino tal que n_i se liste después que n_j. En el ejemplo anterior, n_i es descendiente de n_j pues se lista después.

subárbol : sean n_i y todos los descendientes de n_i en el árbol T_1. En el ejemplo de la figura 6.1, podríamos tener un subárbol que empiece en *rosa* conteniendo los nodos *rosa, alumnos,doc f1.odt y f2.odt*. El propio árbol es un subárbol que cuelga de él mismo.

altura de un nodo : es el camino más largo entre el nodo i y una hoja. Todas las hojas tienen altura cero. La altura de un árbol es la altura del nodo raíz. En la figura 6.1, el nodo *usr* tiene altura $h = 2$ pues el camino más largo hasta llegar a una hora sería *usr-bin-b1.txt*. En cambio la altura de nuestro arbol seria la longitud del camino dado por */-home-rosa-alumnos-f1.odt* que es 4.

profundidad de un nodo : longitud del camino que existe entre el nodo y la raíz. Por ejemplo, los hijos de la raíz tienen profundidad uno.

niveles de un árbol : Gráficamente el nivel se puede definir como todos los nodos que quedan encima de la misma línea horizontal.

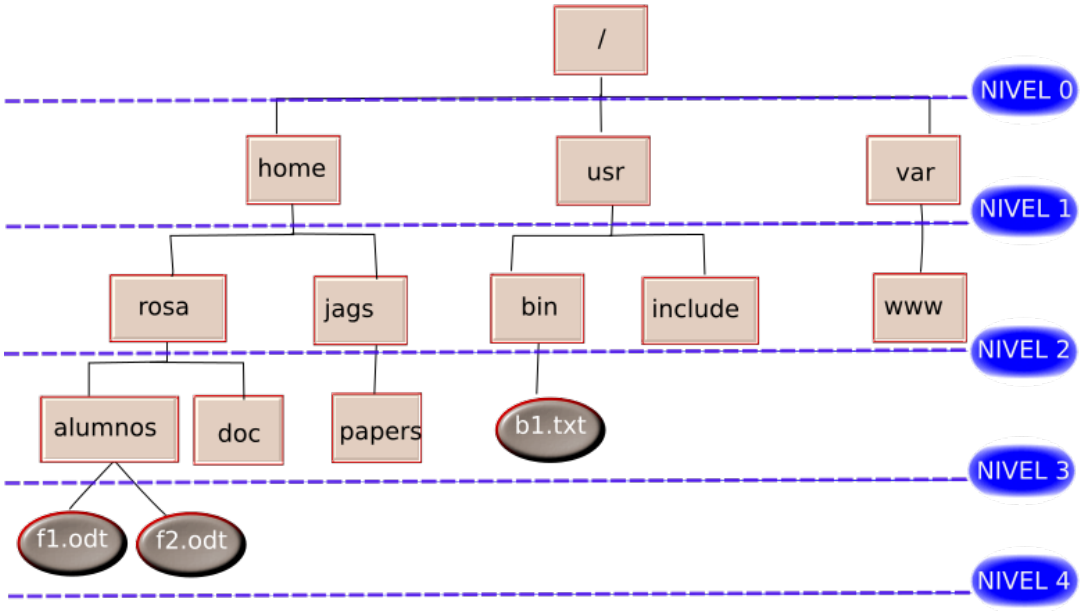


Figura 6.2: Niveles de un árbol

Si un árbol tiene altura h, tenemos $h+1$ niveles. El rango de valores para los niveles va desde 0 hasta h. En el nivel 0 está la raíz, en el nivel 1 están los hijos de la raíz, en el h están las hojas y en el nivel i tenemos todos los nodos de profundidad i. En la figura 6.2 se pueden observar los nodos por niveles.

grado de un nodo (grado de salida): número de hijos que tiene un nodo.

grado de un árbol : máximo de los grados de todos los nodos del árbol.

árbol binario : en un árbol binario, cada nodo puede tener 0, 1 ó 2 hijos. El árbol vacío[3], también se

[3]el árbol que no tiene ningún nodo

considera binario.

árbol 2-ario : cada nodo tiene 0 ó dos hijos. Es equivalente al árbol binario homogéneo.

árbol binario homogéneo : cada nodo tiene 0 ó dos hijos. Es equivalente al árbol 2-ario.

árbol binario completo : es un árbol que tiene todos los niveles completos excepto el último nive l (a partir de ahora lo llamaremos nivel inferior), en cuyo caso, los huecos quedan a la derecha. Por ejemplo:

1. Este árbol sería completo porque sólo tiene un hueco a la derecha en el nivel inferior (el hermano que no tiene f):

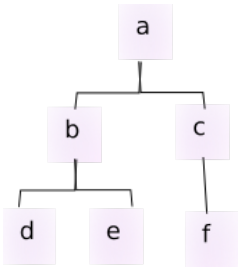


Figura 6.3: Árbol Binario completo

2. Este árbol sería homogéneo y completo porque tiene dos huecos a la derecha en el nivel inferior:

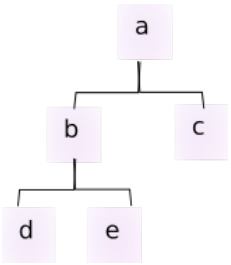


Figura 6.4: Árbol Binario homogeneo y completo

El árbol de la figura 6.3 no sería homogéneo.

3. Este árbol es homogéneo pero no completo porque tiene los huecos a la izquierda en el nivel inferior:

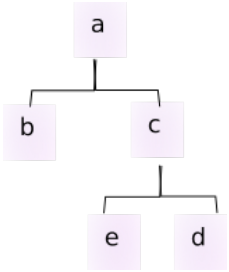
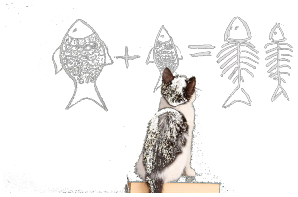


Figura 6.5: Árbol Binario homogeneo pero no es completo



Matemáticas de fondo 6.2.1 El número máximo de nodos de un árbol con h niveles, teniendo en cuenta que un nivel tiene como máximo 2^i nodos es:

$$\sum_{i=0}^{h} 2^i = 2^0 + 2^1 + \cdots + 2^n$$

$$S_n = 2^0 + 2^1 + \cdots + 2^n$$

$$2S_n = 2^1 + \cdots + 2^{n+1}$$

$$2S_n - S_n = 2^{n+1} - 2^0 = 2^{n+1} - 1$$

Teniendo en cuenta que el árbol debe tener todos sus niveles completos.

6.2.2 Recorridos

En un árbol cuando hablamos de recorridos nos referimos al orden en el que visitamos sus nodos. Los recorridos de un árbol se clasifican en:

1. **Profundidad**: Son aquellos en los visitan los nodos desde la raiz hacia las hojas dejándose nodos en un mismo nivel sin visitar hasta más tarde. Para este tipo de recorrido podemos realizarlo de tres formas:
   - Preorden: Al visitar un nodo se procesa en ese momento (bien para imprimir o hacer algo con él).
   - Inorden: Al visitar un nodo se procesará cuando se haya procesado su hijo más a la izquierda.
   - Postorden: Al visitar un nodo se procesará cuando se hayan procesados todos sus hijos.
2. **Anchura** o por niveles. En este recorrido se visitan y procesan en primer lugar todos los nodos del mismo nivel, de izquierda a derecha. Se parte de igual forma desde la raiz y se avanza hacia las hojas. Por ejemplo si vemos el árbol de la figura 6.2 se procesa todos los nodos del nivel 0, luego los del nivel 1, etc.

A continuación para facilitar la explicación de los diferentes recorridos vamos a suponer que el procesamiento que realizamos en cada nodo es listarlo.

Preorden

En el preorden, empezamos listando la raíz y después los subárboles de sus hijos, empezando por el hijo de la izquierda, recursivamente.
Vamos a trabajar sobre el arbol de la figura 6.6:

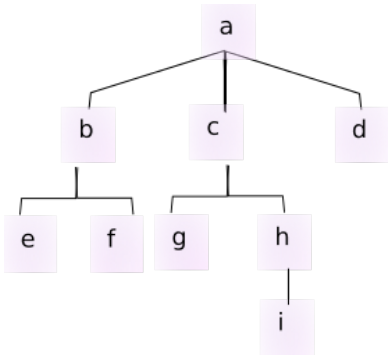


Figura 6.6: Árbol General

El preorden de este ejemplo 6.6, sería:

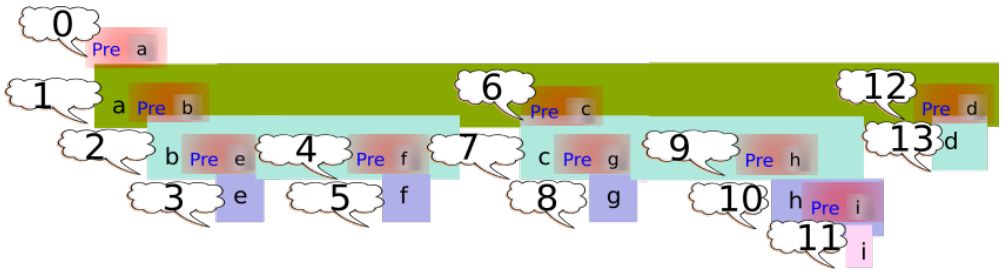


Figura 6.7: Recorrido preorden del árbol de la figura 6.6

Lo primero que se hace es llamar a Pre con el nodo a (Pre(a)) . A continuación los pasos a ejecutar son:

1. Listamos la raíz: a. A continuación vamos al subárbol de b, el hijo más a la izquierda de a y hacemos Pre(b)
2. Listamos la raíz de este subárbol, b, y nos vamos al subárbol del hijo más a la izquierda de b: e. Hacemos Pre(e)
3. Al ser la raíz del subárbol lo listamos, e, pero al no tener hijos volvemos para atrás y listamos el siguiente hijo de b.
4. Llamamos Pre(f)
5. Listamos la raíz del subárbol formado por f y al no tener más hijos volvemos hacia b. Al no tener b más hijos volvemos a a y
6. hacemos Pre(c), el siguiente hijo de a.
7. Listamos la raíz del subárbol formado por c y hacemos Pre(g), el hijo más a la izquierda de c

8. Hacemos el preorden del primer hijo a la izquierda de c, g, listamos la raíz del subárbol formado por g y y como no tiene hijos volvemos para atrás a c y listamos el siguiente hijo de c, h.
9. Hacer Pre(h)
10. Listamos h, hacemos Pre(i), el único hijo de h,
11. Listamos i y como no tiene hijos volvemos para atrás hasta a (llamada 0).
12. Hacemos el preorden del último hijo de a, d.
13. Listamos la raíz del subárbol formado por d y como no tiene hijos volvemos a a. Como a ya no tiene más hijos, hemos terminado el preorden.

El listado en preorden sería:

a b e f c g h i d

Inorden

En el inorden, listamos primero el hijo mas a la izquierda de la raíz, después la raíz y por último, el resto de hijos de la raíz. El inorden del árbol en la figura 6.6 sería:

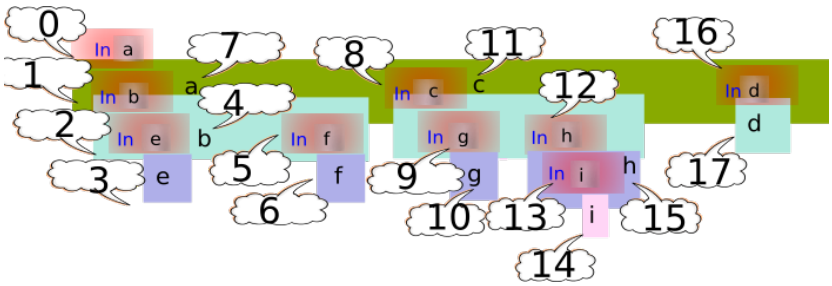


Figura 6.8: Recorrido inorden del árbol de la figura 6.6

Los pasos a seguir trás la primera llamada In(a) son:

1. Hacemos In(b), el primer hijo de a
2. Hacemos In(e) de e, el primer hijo de b
3. Y, como e no tiene hijos, lo listamos.
4. Después, volvemos a b y lo listamos, b,
5. Después, hacemos In(f), el otro hijo de b.
6. Listamos f al no tener hijos.Volvemos para atrás, ya no tenemos más hijos b por lo que volvemos a *a* (llamada 0).
7. Listamos a,
8. Seguimos con In(c), el siguiente hijo de a.
9. Tras hacer In(c), empezamos con su primer hijo por la izquierda, y hacemos In(g),
10. Al no tener g ningún hijo, lo listamos y volvemos a c (llamada 8),
11. Listamos c
12. Seguimos con el otro hijo de c, h, hacemos In(h)
13. Seguimos con el primer hijo de h, i y hacemos In(i).
14. Al no tener i ningún hijo, listamos i y volvemos atrás,

15. Listamos h y al no tener h más hijos volvemos directamente a a, porque c tampoco tiene más hijos.
16. Seguimos con el último hijo de a, d. Hacemos In(d)
17. Y como d no tiene ningún hijo, listamos d y volvemos a *a*. Como a no tiene más hijos, hemos terminado el problema.

El listado en inorden sería:

e b f a g c i h d

Postorden

En el postorden empezamos listando primero todos los hijos de la raíz, empezando por el hijo de la izquierda y, por último, listamos la raíz.
El recorrido en postorden del árbol de la 6.6 sería:

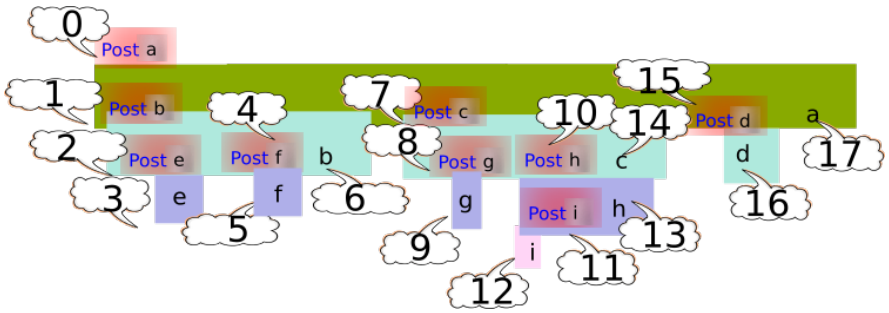


Figura 6.9: Recorrido postorden del árbol de la figura 6.6

Trás realizar la llamada Post(a) Los pasos a seguir serían:

1. Empezamos con el primer hijo a la izquierda de a, b haciendo Post(b).
2. Seguimos con el primer hijo a la izquieda de b que es e y hacemos Post(e),
3. Como e ya no tiene más hijos, lo listamos y volvemos atrás.
4. Seguimos con el otro hijo de b, f haciendo Post(f),
5. Como f no tiene hijos lo listamos y volvemos atrás,
6. Al no tener más hijos b lo listamos y volvemos a a.
7. Después seguimos con c, el siguiente hijo de a. Hacemos un Post(c)
8. Y empezamos con el primer hijo de c, g. Hacemos un Post(g)
9. Como g no tiene hijos, lo listamos.
10. Seguimos con el otro hijo de c, h haciendo un Post(h)
11. Y como h sí tiene un hijo, i, hacemos un Post(i).
12. Como i no tiene hijos lo listamos y volvemos a h,
13. Como h no tiene más hijos, lo listamos y volvemos a c
14. Y como c no tiene más hijos, lo listamos y volvemos a a.
15. Nos vamos al último hijo de a, d, hacemos Post(d)
16. Como d no tiene hijos lo listamos y volvemos a a.
17. Como a no tiene más hijo lo listamos y terminamos el recorrido.

El listado en postorden sería:

$$e \quad f \quad b \quad g \quad i \quad h \quad c \quad d \quad a$$

Anchura o por niveles

En el recorrido por niveles listamos los nodos que hay en cada nivel, empezando por el que esté más a la izquierda. Por ejemplo, el recorrido por niveles de la Figura 6.6 sería:

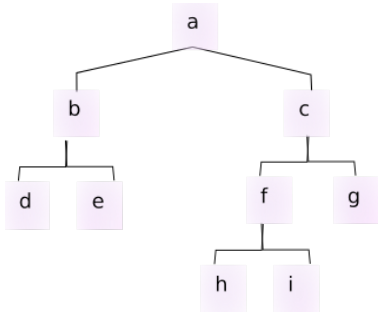
$$\underbrace{a}_{h=0} \quad \underbrace{b \quad c \quad d}_{h=1} \quad \underbrace{e \quad f \quad g \quad h}_{h=2} \quad \underbrace{i}_{h=3}$$

Recorridos en árboles binarios

En esta sección detallaremos como realizar los recorridos cuando el árbol es binario (cada nodo no hoja tiene 0,1 o 2 hijos). Los recorridos en un árbol binario serían:

1. *Preorden*: raíz - Pre(T_{izq}) - Pre(T_{dcha})
2. *Inorden*: In(T_{izq}) - raíz - In(T_{dcha})
3. *Postorden*: Post(T_{izq}) - Post(T_{dcha}) - raíz

Siendo T_{izq} el subárbol izquierda de la raíz y T_{dcha} el subárbol derecha de la raíz. Por ejemplo, los recorridos del siguiente árbol serían:



1. *Preorden*: $a \quad b \quad d \quad e \quad c \quad f \quad h \quad i \quad g$
2. *Inorden*: $d \quad b \quad e \quad a \quad h \quad f \quad i \quad c \quad g$
3. *Postorden*: $d \quad e \quad b \quad h \quad i \quad f \quad g \quad c \quad a$
4. *Por niveles*: $a \quad b \quad c \quad d \quad e \quad f \quad g \quad h \quad i$

Figura 6.10: Arbol Binario

A continuación nos planteamos si es posible recuperar un árbol dada una secuencia que representan un listado del árbol. La respuesta es con un único listado no se puede construir el árbol.

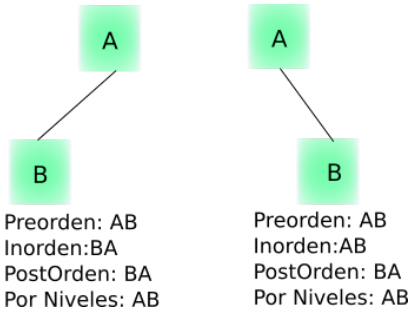


Figura 6.11: Dos árboles binario con sus recorridos

¿Con dos listados se puede recuperar el árbol?

Para ver esto más claro en la Figura 6.11 se puede observar dos árboles diferentes y los listados Preorden, Postorden y Por Niveles coninciden.

Por otro lado podemos plantearnos cuando tenemos dos listados del árbol: *¿puedo recuperar el árbol original?*. Depende de los listados que nos den.

Podemos recuperar el árbol de forma únivoca si los listados que nos dan son:

- *Inorden y Preorden*
- *Inorden y Postorden*
- *Inorden y Por Niveles*

No podremos recuperar si nos dan:

- *Postorden y Preorden* (hay alguna excepción para los árboles binarios pero en general no se puede)
- *Preorden y Por Niveles*
- *Postorden y Por Niveles*

Incluso si nos dieran el Preorden, Postorden y Por niveles no podemos definir el árbol. Esto se puede ver desde la Figura 6.11 en el que el Preorden y Por niveles es AB y Postorden es BA para ambos árboles.

Ejercicio 6.1

Dado un arbol binario completo pensad si podríamos recuperarlo de forma unívoca dado su listado en preorden y postorden. Si la respuesta es afirmativa dar el conjunto de pasos del algoritmo.

□

Por normal general, con sólo uno de los recorridos de un árbol, no puede recuperarse de manera univoca, es decir, dos árboles diferentes pueden tener el mismo recorrido. Por ejemplo en la Figura 6.12, el árbol de la derecha, es la rotación a la derecha[4] del árbol de la izquierda y ambos tienen el mismo inorden:

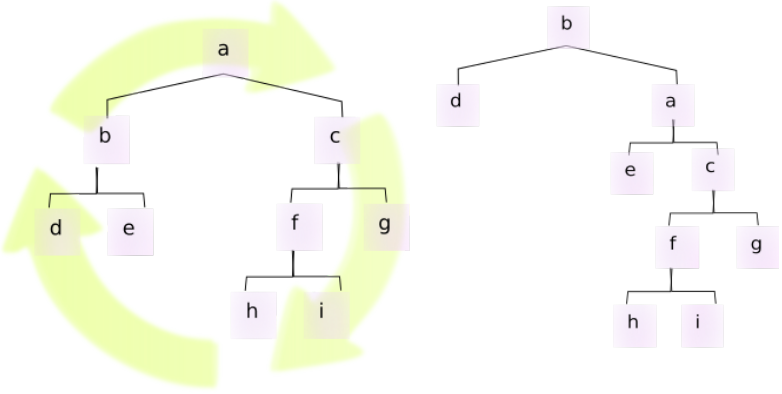


Figura 6.12: Dos árboles binarios con igual recorrido en inorden. El árbol de la derecha se obtiene como una rotación simple a derecha aplicado al árbol de la izquierda.

[4]Desplazamos todos los elementos a un lado (izquierda o derecha), esto se usa para equilibrar el árbol cuando por una rama tiene muchos nodos y por la otra no.

El inorden de ambos árboles es:

$$d \quad b \quad e \quad a \quad h \quad f \quad i \quad c \quad g$$

En la siguiente tabla damos un repaso y resumen de los recorridos, los valores serán verdadero o falso si n se lista antes o después que m:

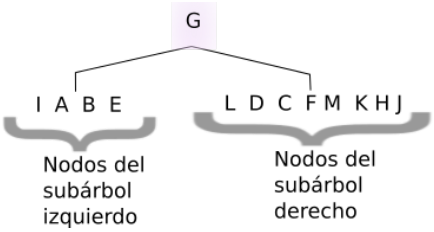
	$Pre(n) < Pre(m)$	$In(n) < In(m)$	$Post(n) < Post(m)$
$n \in h_{izq}(m)$	F	V	V
$n \in h_{dcha}(m)$	F	F	V
$n \in descendiente(m)$	F	V si es descendiente por la izquierda y F si lo es por la derecha	V
$n \in ancestro(m)$	V	igual que antes	F

Ejemplo 6.2.1
Dados el preorden y el inorden, obtén el correspondiente árbol binario:

1. *preorden*: G E A I B M C L D F K J H
2. *inorden*: I A B E G L D C F M K H J

Tenemos que fijarnos en los siguientes detalles:

1. La G corresponde con la raíz del árbol pues es la primera que listamos en el preorden.
2. El subárbol que corresponde al hijo izquierdo de G está listado en el inorden antes que G y el hijo a la derecha, está listado después de G. En este caso generaría un primer boceto de árbol de la siguiente forma:



3. Estos razonamientos se aplican recursivamente a los distintos subárboles del árbol. De esta forma obtendríamos para el ejemplo dado el árbol que se muestra en la Figura 6.13

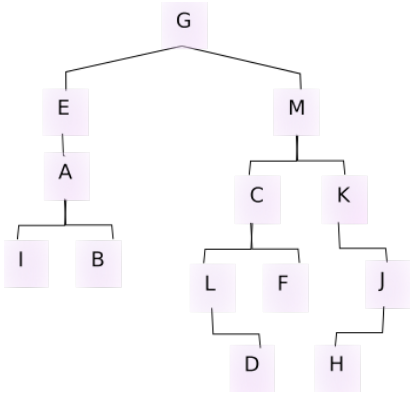


Figura 6.13: Árbol resultante del listado en preorden e inorden

□

6.2.3 Lectura y escritura de un árbol binario en disco

Para guardar un árbol en disco, se realiza un preorden del árbol transformado. Este árbol transformado consiste en añadirle a los nodos que no tienen los dos hijos (si es un árbol binario) un nuevo nodo fiticio, que tiene como etiqueta *x*. Cuando hacemos el listado del arbol si el nodo existe se le antepone a la etiqueta *n* y si es un nodo fiticio simplemente listamos *x*. Por ejemplo, para guardar el siguiente árbol:

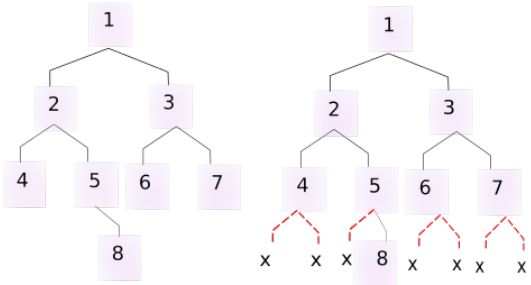


Figura 6.14: Izquierda: Arbol original. Derecha: Árbol transformado para aplicar la lectura/escritura

El preorden que deberíamos escribir en el disco sería: n1n2n4xxn5xn8xxn3n6xxn7xx Este proceso de escritura y lectura lo veremos con más detalle tanto para árboles binarios como para árboles generales.

6.3 Arboles binarios

En esta seccion analizaremos como representar un árbol binario, cuales son las operaciones más relevantes y formas de recorrerlos. Especificación: 1)Son árboles tal que cada nodo tiene 0, 1 o 2 hijos. Cada nodo tiene un nodo padre a excepción del nodo raíz que no tiene padre. 2) El árbol vacío es un árbol binario.

6.3.1 Representación

Una primera aproximación al árbol binario la haremos como un objeto de tipo nodo (lo llamaremos *info_nodo*) en el que tendremos: 1)la información o etiqueta que almacena; y 2) enlaces al padre,hijo izquierda e hijo derecha. Hay que reflexionar simplemente que dando el nodo raíz tenemos la información de todo el árbol. Por lo tanto la primera representación la haremos de la siguiente forma:

```
#include <queue> //para hacer el recorrido por niveles
using namespace std;
template <class T>
struct info_nodo {
      info_nodo *padre, //puntero al padre
      *hijoizq, //puntero al hijo izquierda
      *hijodcha; // puntero al hijo derecha
      T et; // etiqueta del nodo

      // Constructor por defecto del struct
      info_nodo() {
            padre = hijoizq = hijodcha = 0;
      }

      info_nodo(const T &e) {
            et = e;
            padre = hijoizq = hijodcha = 0;
      }
};
```

Como se puede observar hemos incluido constructores: por defecto y por parámetros. En la Figura 6.15 se puede ver la estructura info_nodo. Tambien hemos dibujado un árbol binario (a la izquierda) como se hará normalmente, y a la derecha que valores de los que campos tendrían los nodos del árbol.

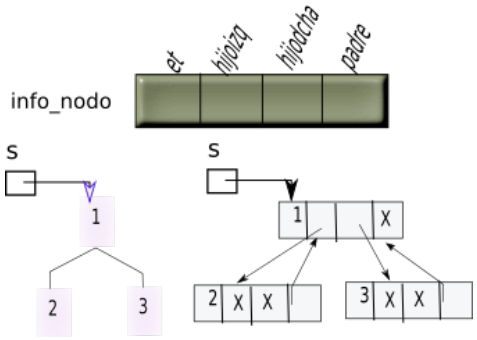


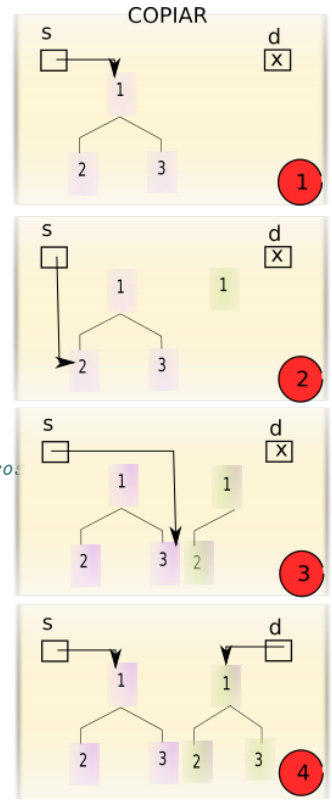
Figura 6.15: Arriba representación de un info_nodo. A la izquierda un árbol binario, y a la derecha como se rellenaría los campos de los nodos que cuelgan de s

Las operaciones a implementar en un árbol binario serían:

1. *Get*: padre, hijoizq, hijodcha, etiqueta
2. *Insertar*: hijoizq, hijodcha
3. *Podar*: hijoizq, hijodcha
4. *Recorridos*: preorden, inorden, postorden, anchura
5. *Leer/escribir*: lee/escribe un árbol binario en un flujo
6. *Copiar*: copia un árbol binario en otro
7. *Borrar*: elimina toda la memoria del árbol binario.
8. *size*: devuelve el número de nodos del árbol binario
9. *Iguales*: establece si dos árboles binarios son iguales.

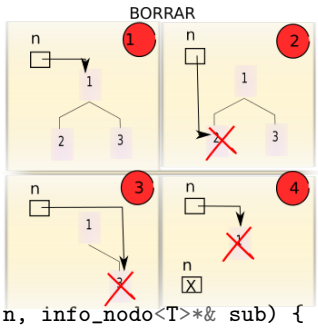
Con esta primera aproximación, un enfoque puramente funcional, vamos a abordar la implementación de las operaciones como funciones. Cada función como mímino tendrá un nodo del árbol.

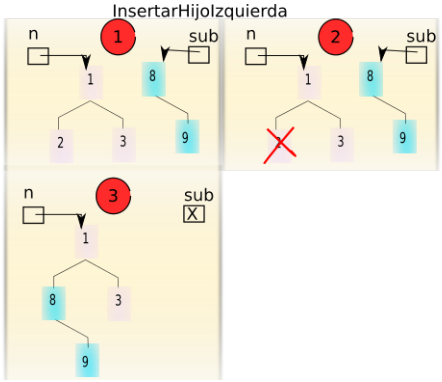
```
21   template <class T>
22   info_nodo<T>* GetPadre (info_nodo<T>* n) {
23         return n->padre;
24   }
25
26   template <class T>
27   info_nodo<T>* GetHijoIzquierda (info_nodo<T>* n) {
28         return n->hijoizq;
29   }
30   template <class T>
31   info_nodo<T>* GetHijoDerecha (info_nodo<T>* n) {
32         return n->hijodcha;
33   }
34  template <class T>
35  void Copiar (info_nodo<T>* s, info_nodo<T>* &d) {
36        if (s == 0)
37              d = 0;
38        else {
39       //invoca al constructor de info nodo con parametros
40              d = new info_nodo<T> (s->et);
41              Copiar (s->hijoizq,d->hijoizq);
42              Copiar (s->hijodcha,d->hijodcha);
43              if (d->hijoizq != 0)
44                    d->hijoizq->padre = d;
45
46              if (d->hijodcha != 0)
47                    d->hijodcha->padre = d;
48        }
49  }
```



En la imagen que acompaña al anterior código se puede ver un ejemplo del proceso de Copiar. Inicialmente se llama con un puntero a info_nodo, s (árbol fuente), que contiene la dirección de la raiz del árbol fuente y d (árbol destino) con valor 0. En este paso se crea un nuevo objeto info_nodo que es apuntado por d y se llama recursivamente a Copiar con $s->hijoizq$ que apunta al nodo con etiqueta 2 y $d->hijoizq$ que es 0 (ver en la figura la viñeta 2). En la ejecución de esta llamada se crea un nuevo nodo para la variable d actual y se copia la etiqueta 2. Se llama con el hijo a la izquierda pero es 0 por

lo tanto simplemente se pone el hijo a la izquierda de 2 (en el destino) a 0. A continuación se llama con el hijo a la derecha de 2 pero tambien es 0. Por lo tanto se vuelve de la recusrividad al nodo 1 y se llama a copiar con el hijo a la derecha. Un vez finalizado todo el proceso como quedan s y d se puede ver en la viñeta 4. Un detalle a observar es que cuando se vuelve de la recursividad de copiar el hijo a la izquierda como copiar el hijo a la derecha se asignan los padres de estos, poniendo en ambos d.






```
52  template <class T>
53  void BorrarInfo (info_nodo<T>* n) {
54        if (n != 0) {
55              BorrarInfo(n->hijoizq);
56              BorrarInfo(n->hijodcha);
57              delete n;
58        }
59  }
60  template <class T>
61  void InsertarHijoIzquierda (info_nodo<T>* n, info_nodo<T>*& sub) {
62        info_nodo<T>* aux = n->hijoizq;
63        if (sub != 0) {
64              n->hijoizq = sub;
65              BorrarInfo(aux);
66              n->hijoizq->padre = n;
67              sub=0;
68        }
69        else {
70              n->hijoizq = 0;
71              BorrarInfo(aux);
72        }
73  }
74  template <class T>
75  void InsertarHijoDerecha (info_nodo<T>* n, info_nodo<T>* &sub) {
76        info_nodo<T>* aux = n->hijodcha;
77        if (sub != 0) {
78              n->hijodcha = sub;
79              BorrarInfo(aux);
80              n->hijodcha->padre = n;
81              sub=0;
82        }
83        else {
84              n->hijodcha = 0;
85              BorrarInfo(aux);
86        }
87  }
```

Con las funciones *InsertarHijoIzquierda* e *InsertarHijoDerecha* son funciones útiles ya que dan posibilidades de ir haciendo crecer el árbol además de modificarlo en el futuro. A continuación sobrecargamos estás dos funciones para que en vez de pasarle un subárbol le pasemos una etiqueta.

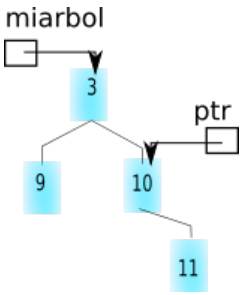
```
88  //Hacemos una sobrecarga de esta funcion
89  //para pasarle una etiqueta en vez del nodo
```

```
90  template <class T>
91  void InsertarHijoIzquierda (info_nodo<T>* n, const T &v) {
92        info_nodo<T>* aux = new info_nodo(v);
93        InsertarHijoIzquierda(n,aux);
94  }
95
96  template <class T>
97  void InsertarHijoDerecha (info_nodo<T>* n, const T &v) {
98        info_nodo<T>* aux = new info_nodo(v);
99        InsertarHijoDerecha(n,aux);
100 }
```

Así con esta sobrecarga permite al usuario de árbol binario hacer crecer su árbol. Por ejemplo un código para crear un arbol sería el siguiente:

```
1  ...
2   //crea un arbol con raiz 3
3  info_nodo<int> *miarbol=new info_nodo(3);
4
5  //ahora le insertamos hijos
6  InsertarHijoIzquierda(miarbol,9);
7  InsertarHijoDerecha(miarbol,10);
8
9  //usamos otro puntero para poner mas descendientes
10 info_nodo<int> *ptr=miarbol->hijodcha;
11 InserrtarHijoDerecha(ptr,11);
```

Trás la ejecución de este código tendríamos que *miarbol* apunta a:



Sigamos con las funciones asociadas a árbol binario. A continuación veremos dos funciones que permite eliminar un subárbol de un nódo en el árbol. De la misma forma que con insertar las posibilidades para eliminar son: podar el subárbol hijo izquierda, implementado con la función *PodarHijoIzquierda*; o eliminar el subárbol hijo derecha, implementado con la función *PodarHijoDerecha*. Con estas dos funciones la memoria usada por los subárboles se elimina. Una segunda posibilidad es construir un árbol con el subarbol que se poda, y por lo tanto no borrar la memoria del subarbol. Para poder tener esta funcionalidad hemos incluido las funciones: *Podar_HijoIzq_getSubtree* y *Podar_HijoDcha_getSubtree*.

```
101 template <class T>
102 void PodarHijoIzquierda (info_nodo<T>* n) {
103       if (n->hijoizq != 0) {
104             BorrarInfo(n->hijoizq);
105             n->hijoizq = 0;
106       }
107 }
108
109 template <class T>
110 void PodarHijoDerecha (info_nodo<T>* n) {
111       if (n->hijodcha != 0) {
112             BorrarInfo(n->hijodcha);
113             n->hijodcha = 0;
114       }
115 }
116
117 // Con esta funcion obtenemos el arbol que hemos podado
118 template <class T>
119 info_nodo<T>* Podar_HijoIzq_getSubtree (info_nodo<T>* n) {
120       info_nodo<T>* aux = n->hijoizq;
121       n->hijoizq = 0;
122       if (aux != 0)
123             aux->padre = 0;
124
125       return aux;
126 }
127
128 template <class T>
129 info_nodo<T>* Podar_HijoDcha_getSubtree (info_nodo<T>* n) {
130       info_nodo<T>* aux = n->hijodcha;
131       n->hijodcha = 0;
132       if (aux != 0)
133             aux->padre = 0;
134
135       return aux;
136 }
```

A continuación se describen las siguientes funciones:

- *iguales*: que establecer si dos árboles son iguales. Para ver que son iguales tiene que tener la misma estructura de nodos, y los nodos a su vez tienen que tener las mismas etiquetas. Ver que tienen la mis ma estructura se implementa con las dos primeras comparaciones: 1)Si ambos nodos son nulos son iguales ; 2) No son iguales en el caso de que uno sea nulo y el otro no.
- *numero_nodos*: contabiliza el número de nodos que tiene un árbol. La idea que se implementa es que un árbol con raiz n tiene 1 nodo por la raiz, más el número de nodos que tenga el subarbol

izquierda (esto se implementa con una llamada recursiva) y otros tantos nodos por el subárbol derecha (segunda llamada recursiva).

```
137 // funcion que define si dos arboles son iguales
138 template <class T>
139 bool iguales(info_nodo<T>* n1, info_nodo<T>* n2) {
140       if (n1==0 && n2==0) //ambos arboles estan vacios
141             return true;
142
143       else if (n1==0 || n2==0)
144             return false; //uno de los arboles es vacio
145
146       else { //ninguno es vacio
147             if (n1->et == n2->et)
148                   return iguales(n1->hijoizq, n2->hijoizq) &&
149                         iguales(n1->hijodcha, n2->hijodcha);
150
151             else
152                   return false;
153       }
154 }
155
156 template <class T>
157 int numero_nodos (info_nodo<T>* n) {
158       if (n==0)
159             return 0;
160       else
161             return numero_nodos(n->hijoizq) +
162                   numero_nodos(n->hijodcha) + 1;
163 // devolvemos el numero de nodos de los dos subarboles hijos
164 // de la raiz y le sumamos 1 para contar tambien la raiz en
165 // el numero de nodos
166 }
```

A continuación implementaremos las funciones que nos permiten realizar los recorridos. Los recorridos en profundidad tienen un esquema muy parecido, en estas funciones para imprimir las etiquetas [5]. La diferencia de estas tres funciones es donde se pone la salida de la etiqueta del nodo *n*. Asi si es preorden es lo primero que se hace, si es inorden, se realiza entre las dos llamadas recursivas y si es postorden se realiza una vez concluidas las llamadas recursivas.

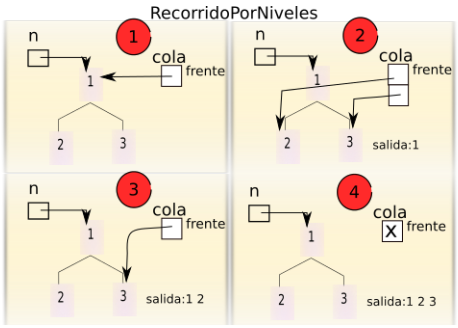
```
166 template <class T>
167 void RecorridoPreorden (ostream & os, const info_nodo<T> *n) {
168 /* En este caso, nuestro caso base seria tener un arbol vacio
169 pero en ese caso base no se haria nada*/
```

[5] podríamos hacer el recorrido y en cada nodo hacer cualquier otra operación diferente a la impresión

```
170   if (n!=0) {
171         os << n->et << ' '; //listamos la raiz
172           //hacemos preorden del hijo izquierdo
173         RecorridoPreorden (os, n->hijoizq);
174         RecorridoPreorden (os, n->hijodcha); //y luego del hijo derecho
175   }
176 }
177
178 template <class T>
179 void RecorridoInorden (ostream & os, const info_nodo<T>* n) {
180   if (n!=0) {
181   //Hacemos inorden del hijo izquierdo
182         RecorridoInorden (os, n->hijoizq);
183         os << n->et << ' '; //listamos la raiz
184         //y hacemos inorden del hijo derecho
185         RecorridoInorden (os, n->hijodcha);
186   }
187 }
188
189 template <class T>
190 void RecorridoPostorden (ostream & os, const info_nodo<T>* n) {
191   if (n!=0) {
192         //Hacemos postorden del hijo izquierdo
193         RecorridoPostorden (os, n->hijoizq);
194         RecorridoPostorden (os, n->hijodcha); //luego del hijo derecho
195         os << n->et << ' '; //y por ultimo listamos la raiz
196   }
197 }
```

Otro de los recorridos que vamos a implementar es el recorrido en Anchura o Por Niveles. Para llevar a cabo este recorrido usaremos una cola que nos guarde los nodos aún no procesados y en el orden en el que hay que procesarlos. En este recorrido los nodos se deben procesar por niveles y en cada nivel de izquierda a derecha. De esta forma en primer lugar insertamos en la cola el nodo raíz. Y a continuación pasamos a un bucle mientras que la cola no esté vacía. Sacamos el nodo en el frente y los imprimimos. A continuación ponemos al final de la cola sus hijos. Cuando la cola esté vacía habremos listado todos los nodos por niveles.

```
198 template <class T>
199 void RecorridoPorNiveles (ostream & os, const info_nodo<T>* n) {
200  if (n!=0) {
201   queue<const info_nodo<T>*> cola;
202
203   cola.push(n); //guardamos el primer nodo
204
205   while (!cola.empty()) {
206   //ultimo nodo que hemos guardado
207     const info_nodo<T>* p = cola.front();
208     //lo listamos
209     os << p->et << ' ';
210     /*si tiene hijo a
211     la izquierda lo guardamos*/
212     if (p->hijoizq != 0)
213           cola.push(p->hijoizq);
214     //igual con el hijo de la derecha
215     if (p->hijodcha != 0)
216           cola.push(p->hijodcha);
217    //borramos el elemento que ya hemos listado
218     cola.pop();
219     // esto tambien se podria haber hecho antes de los if
220   }
221  }
222 }
```



Para las funciones de leer y escribir, vamos a guardar un árbol en disco como ya hemos explicado en la sección 6.14:

```
223 template <class T>
224 void Escribe (ostream & os, const info_nodo<T>* n) {
225       if (n==0)
226        //cuando el nodo es hijo de una hoja, se pone una x
227        os << 'x';
228       else {
229       //cuando no, se pone su etiqueta detras de una n
230             os << 'n' << n->et;
231             Escribe(os, n->hijoizq);
232             Escribe(os, n->hijodcha);
233       }
234 }
235
236 template <class T>
237 void Lee (istream & is, info_nodo<T>* &n) {
238       char c;
239       c = is.get();
```

```
240         if (is) {
241               if (c=='x') n=0; //nodo vacio
242               else {
243                     T e;
244                     // si T es un tipo definido por nosotros,
245                     // debemos definir su operador de entrada
246                     is >> e;
247                     n = new info_nodo<T>(e);
248                     Lee (is,n->hijoizq);
249                     Lee (is,n->hijodcha);
250                     // ahora enlazamos a los hijos con su padre
251                     if (n->hijoizq != 0) n->hijoizq->padre = n;
252                     if (n->hijodcha != 0) n->hijodcha->padre = n;
253               }
254         }
255   }
```

Ejemplo 6.3.1
La siguiente función, refleja un árbol. Antes de implementarla vamos a poner un ejemplo para entender lo que sería reflejar un árbol. El árbol de la derecha es el reflejo del árbol de la izquierda:

Como se puede ver en el ejemplo, vamos cambiando recursivamente el subárbol de la izquierda por el de la derecha. La implementación en C++ sería:

```
261   template <class T>
262   void Reflejo (info_nodo<T>* n) {
263         if (n!=0) {
264               swap (n->hijoizq,n->hijodcha);
265               Reflejo(n->hijoizq);
266               Reflejo(n->hijodcha);
267         }
268   }
```

□

6.3.2 Clase ArbolBinario

La siguiente aproximación a la representación del T.D.A. ArbolBinario y dar una nueva capa de abstracción. Para ello encapsularemos todo lo que hemos visto anteriormente en el entorno

classArbolBinario. De la siguiente forma:

```
template <class T>
class ArbolBinario{
    private:
    struct info_nodo{
              T et;
              info_nodo * padre;
              info_nodo * hizq;
              info_nodo * hder;
              info_nodo(){ padre=hizq=hder=0; }
              info_nodo(const T & e){ et = e; padre=hizq=hder=0;}
  };
    //Funciones asociadas a info_nodo
    void Copiar(info_nodo * &dest,const info_nodo*const &source);

    void BorrarInfo(info_nodo *&d);

    unsigned int numero_nodos(const info_nodo*d)const ;

    bool iguales(const info_nodo*s1,const info_nodo*s2)const ;

    void InsertarHijoIzquierda(info_nodo * n,info_nodo * sub);

    void InsertarHijoIzquierda(info_nodo * n,const T & e);

    void InsertarHijoDerecha(info_nodo * n, info_nodo * sub);

    void InsertarHijoDerecha(info_nodo * n,const T & e);

    void PodarHijoIzquierda(info_nodo * n);

    void PodarHijoDerecha(info_nodo * n);

    info_nodo *PodarHijoIzq_GetSubtree(info_nodo * n);

    info_nodo *PodarHijoDer_GetSubtree(info_nodo * n);

    void RecorridoPreorden(ostream & os, const info_nodo *n)const ;

    void RecorridoPostorden(ostream & os,const info_nodo *n)const ;

    void RecorridoInorden(ostream & os,const info_nodo *n)const ;

    void RecorridoNiveles(ostream &os,const info_nodo *n)const ;

    void Lee(istream & is, info_nodo *&n);

    void Escribe(ostream & os,const  info_nodo *n)const;

...
//ahora como se representa un ArbolBinario
};
```

Con estas funciones privadas ahora la representación:

```
template <class T>
class ArbolBinario{
    private:
    struct info_nodo{
              T et;
              info_nodo * padre;
              info_nodo * hizq;
              info_nodo * hder;
              info_nodo(){ padre=hizq=hder=0; }
              info_nodo(const T & e){ et = e; padre=hizq=hder=0;}
  };
    //Funciones asociadas a info_nodo
    ...
    info_nodo *raiz; //raiz del arbol binario

...
};
```

Antes de pasar a ver los métodos de ArbolBinario hace falta especificar que es un nodo dentro del árbol. Un nodo dentro del árbol se debe ver como un objeto que apunta dentro del árbol. En este sentido nodo se puede considerar como un iterador. Y por lo tanto dentro de la clase ArbolBinario se va a implementar la clase nodo que permita con objetos de este tipo apuntar a la información contenida en el árbol.

```
template <class T>
class ArbolBinario{
  private:
         ....
  public:

  class nodo{
    private:
        info_nodo *p;
        nodo (info_nodo * i):p(i){}
    public:

        nodo ():p(0){}

        const T& operator*()const {
                  assert(p!=0);
                  return p->et;
        }
       T& operator*() {
                  assert(p!=0);
                  return p->et;
        }

        bool operator==(const nodo &n){
                  return p==n.p;
        }

        bool operator!=(const nodo &n){
                  return p!=n.p;
        }

```

```
32          //devuelve un nodo apuntando al padre
33          nodo padre(){
34                    if (p->padre!=0)
35                    return nodo(p->padre);
36                    else return nodo();
37          }
38          //devuelve un nodo apuntando al hijo izquierdo
39          nodo hi(){
40                    if (p->hizq!=0)
41                    return nodo(p->hizq);
42                    else return nodo();
43          }
44          //devuelve un nodo apuntando al hijo derecho
45          nodo hd(){
46                    if (p->hder!=0)
47                    return nodo(p->hder);
48                    else return nodo();
49          }
50
51          bool nulo(){
52                    return p==0;
53          }
54          friend class ArbolBinario;
55
56
57    };
58  };
```

La clase *nodo* tiene un atributo privado que es un puntero a info_nodo. Como se puede ver en el anterior código *nodo* tiene todas las funciones típicas de un iterador a excepción del operador $++$ y operador $--$. Estas dos operaciones no se ha incluido en *nodo* ya que para un árbol binario recorrer sus nodos se puede hacer en: preorden, inorden, postorden y por niveles. Por lo tanto no sabemos como avanzar cuando se invoque el operador $++$ sobre un nodo. Esta ambigüedad la resolveremos en las siguiente secciones.
Para hacer uso de un nodo tendremos que usar la sintaxis:

$$\boxed{typename\ \ ArbolBinario < T >:: nodo}$$

Es necesario anteponer *typename*, en primer lugar porque ArbolBinario es una clase template. Y además al hacer referencia a nodo, si no ponemos la palabra clave *typename* el compilador, el tipo nodo, lo interpreta como un objeto miembro y no como un tipo.
Ahora ya habiéndo definido nodo veremos como sería la interfaz para ArbolBinario.

```
1   template <class T>
2   class ArbolBinario{
3     private:
4           ....
5     public:
6     info_nodo * raiz;
7     ....
8     class nodo{
9     ...
10    }
```

```
11
12      ArbolBinario(const T &e);
13      ArbolBinario(typename ArbolBinario<T>::nodo n);
14      /**
15        @brief Constructor por copia
16      */
17      ArbolBinario(const ArbolBinario<T> & ab);
18
19      /**
20        @brief Destructor
21        */
22      ~ArbolBinario(){ clear();}
23      /**
24        @brief Operador de asignacion
25        @param ab: arbol binario del que se copia
26      */
27      ArbolBinario<T> & operator=(const ArbolBinario<T> & ab);
28
29      /**
30        @brief Obtiene un nodo apuntando a la raiz del arbol
31        */
32
33      typename ArbolBinario<T>::nodo getRaiz()const;
34      /**
35        @brief Inserta un subarbol como hijo izquierdo del nodo.
36        Este suabarbol solamente tiene un nodo
37        @param n: posicion del nodo donde insertar el subarbol como hijo izquierdo
38        @param e: etiqueta de la raiz del subarbol que se inserta
39      */
40      typename ArbolBinario<T>::nodo Insertar_Hi( typename ArbolBinario<T>::nodo n,
41                                                  const T &e);
42      /**
43        @brief Inserta un subarbol como hijo izquierdo del nodo.
44        @param n: posicion del nodo donde insertar el subarbol como hijo izquierdo
45        @param tree:subarbol que se inserta. ES MODIFICADO
46      */
47
48      typename ArbolBinario<T>::nodo Insertar_Hi( typename ArbolBinario<T>::nodo n ,
49                                                  ArbolBinario<T> & tree);
50
51      /**
52        @brief Inserta un subarbol como hijo derecho del nodo.
53        Este suabarbol solamente tiene un nodo
54        @param n: posicion del nodo donde insertar el subarbol como hijo derecho
55        @param e: etiqueta de la raiz del subarbol que se inserta
56      */
57      typename ArbolBinario<T>::nodo Insertar_Hd( typename ArbolBinario<T>::nodo n,
58                                                  const T &e);
59
60      /**
61        @brief Inserta un subarbol como hijo derecho del nodo.
62        @param n: posicion del nodo donde insertar el subarbol como hijo derecho
63        @param tree:  subarbol que se inserta. ES MODIFICADO
```

```
64      */
65      typename ArbolBinario<T>::nodo Insertar_Hd( typename ArbolBinario<T>::nodo n,
66                                                  ArbolBinario<T> & tree);
67
68
69      /**
70        @brief Poda el hijo izquierdo del nodo dado
71        @pos: posicion del nodo
72      */
73      void Podar_Hi(typename ArbolBinario<T>::nodo pos);
74
75      /**
76        @brief Poda el hijo derecho del nodo dado
77        @pos: posicion del nodo
78      */
79
80      void Podar_Hd(typename ArbolBinario<T>::nodo pos);
81
82      /**
83        @brief Poda el hijo derecho o izquierda del nodo del nodo dado
84        @pos: posicion del nodo
85        @return un arbol nuevo con esta rama eliminada
86      */
87      ArbolBinario<T>  PodarHi_GetSubtree(typename ArbolBinario<T>::nodo pos);
88      ArbolBinario<T>  PodarHd_GetSubtree(typename ArbolBinario<T>::nodo pos);
89
90      /**
91        @brief Se sustituye el subarbol por otro subarbol de otro arbol
92        @param pos_this: posicion de la raiz del subarbol a ser copiado.
93        El que hubiese previo se elimina.
94        @param a: arbol fuente.
95        @param pos_a: posicion de la raiz del suarbol de \a  a que va a ser copiado.
96      */
97      void Sustituye_Subarbol(typename ArbolBinario<T>::nodo pos_this,
98                              const ArbolBinario<T> &a,
99                              typename ArbolBinario<T>::nodo pos_a);
100
101     /**
102       @brief Borra todo arbol, dejandolo como un arbol vacio
103     */
104     void clear();
105
106     /**
107       @brief Arbol vacio
108       @return Devuelve si el arbol es vacio (true), y falso en caso contrario
109     */
110     bool empty()const ;
111
112     /**
113       @brief Numero de nodos de un arbol
114       @return Devuelve  el numero de nodos que tiene el arbol
115
116     */
```

```
117     unsigned int size()const ;
118
119     /**
120       @brief Igualdad entre dos arboles
121       @param a: arbo binario con el que se compara
122       @return true si los dos arboles son iguales false en caso contrario
123     */
124     bool operator==(const ArbolBinario<T> &a)const;
125
126     /**
127       @brief Desigualdad entre dos arboles
128       @param a: arbo binario con el que se compara
129       @return true si los dos arboles son distintos false en caso contrario
130     */
131     bool operator!=(const ArbolBinario<T> &a)const;
132
133
134     /**
135       @brief Recorrido en Preorden
136       @param os: flujo sobre el que se da el recorrido del arbol en preorden
137       */
138     void RecorridoPreOrden(ostream &os)const ;
139     /**
140       @brief Recorrido en Inorden
141       @param os: flujo sobre el que se da el recorrido del arbol en Inorden
142       */
143     void RecorridoInOrden(ostream &os)const ;
144     /**
145       @brief Recorrido en Postorden
146       @param os: flujo sobre el que se da el recorrido del arbol en Postorden
147       */
148
149     void RecorridoPostOrden(ostream &os)const ;
150     /**
151       @brief Recorrido por niveles
152       @param os: flujo sobre el que se da el recorrido del arbol por niveles
153       */
154     void RecorridoNiveles(ostream &os)const ;
155
156 }
```

Las implementaciones de estos métodos de la clase ArbolBinario en su mayoría se implementan usando las funciones privadas que ya discutimos para punteros de *info_nodo*.

```
1  // Constructor para construir un arbol a partir de una etiqueta
2  template <class T>
3  ArbolBinario<T>::ArbolBinario (const T &e) {
4      raiz = new info_nodo(e);
5  }
6
7  // Constructor para construir un arbol a partir de un nodo
8  template <class T>
9  ArbolBinario<T>::ArbolBinario (typename ArbolBinario<T>::nodo n) {
10     raiz = n.p; // esto se puede hacer porque ArbolBinario es amiga de nodo
```

```
11  }
12
13  //Constructor para construir un arbol a partir de otro arbol (de copia)
14  template <class T>
15  ArbolBinario<T>::ArbolBinario (const ArbolBinario<T> &ab) {
16      if (ab.raiz==0)
17        raiz = 0;
18      else
19        Copiar(raiz,ab.raiz);
20        // esta llamada a copiar es al metodo privado de la clase,
21        // no a la funcion copiar.
22  }
23
24  template <class T>
25  ArbolBinario<T> & ArbolBinario<T>::operator= (const ArbolBinario<T> &ab) {
26        if (*this != &ab) {
27              BorrarInfo(raiz);
28              Copiar(raiz,ab.raiz);
29        }
30
31        return *this;
32  }
33
34  // Esta es la funcion a la que llama el destructor
35  template <class T>
36  void ArbolBinario<T>::clear() {
37        BorrarInfo(raiz);
38  }
39
40  template <class T>
41  bool ArbolBinario<T>::empty()const {
42        return raiz==0;
43  }
44
45  template <class T>
46  unsigned int ArbolBinario<T>::size() const {
47        return numero_nodos(raiz);
48  }
49
50  template <class T>
51  bool ArbolBinario<T>::operator==(const ArbolBinario<T> &ab) {
52        return iguales(raiz,ab.raiz);
53  }
54
55  template <class T>
56  bool ArbolBinario<T>::operator!=(const ArbolBinario<T> &ab) {
57        return !(*this == ab);
58        //otra opcion seria: return !iguales(raiz,ab.raiz);
59  }
60
61  template <class T>
62  void ArbolBinario<T>::RecorridoPreorden (ostream &os)const {
63        RecorridoPreorden(os,raiz);
```

```
64  }
65
66  template <class T>
67  void ArbolBinario<T>::RecorridoInorden (ostream &os)const {
68        RecorridoInorden(os,raiz);
69  }
70
71  template <class T>
72  void ArbolBinario<T>::RecorridoPostorden (ostream &os)const {
73        RecorridoPostorden(os,raiz);
74  }
75
76  template <class T>
77  void ArbolBinario<T>::RecorridoPorNiveles (ostream &os)const {
78      RecorridoPorNiveles(os,raiz);
79  }
80
81  // En los operadores de E/S podemos darle otro tipo distinto de T,
82  // pues no pertenecen a la clase que estamos implementando
83  template <class U>
84  istream & operator>> (istream &is, ArbolBinario<U> &ab) {
85      ab.Lee (is,ab.raiz);
86      return is;
87  }
88
89  template <class U>
90  ostream & operator<< (ostream &os, ArbolBinario<U> &ab) {
91      ab.Escribe(os, ab.raiz);
92      return os;
93  }
94
95  // Implementacion de la clase nodo:
96  template <class T>
97  typename ArbolBinario<T>::nodo ArbolBinario<T>::getRaiz()const {
98      if (raiz != 0)
99            return typename ArbolBinario<T>::nodo (raiz);
100           // Devuelve un objeto de tipo nodo que apunta a la raiz del arbol
101
102     else
103           return typename ArbolBinario<T>::nodo(); //arbol vacio
104 }
105
106 template <class T>
107 typename ArbolBinario<T>::nodo ArbolBinario<T>::Insertar_Hi(
108                         typename ArbolBinario<T>::nodo n, const T &e) {
109     /*Esta funcion elimina el hijo izquierdo de n e inserta una nueva rama
110     con el nodo de etiqueta e. Devuelve un nodo apuntando al nuevo hijo a
111     la izquierda, el nodo de etiqueta e*/
112     InsertarHijoIzquierda(n.p,e);
113     return typename ArbolBinario<T>::nodo (n->hijoizq);
114 }
115
116 template <class T>
```

```
117 typename ArbolBinario<T>::nodo ArbolBinario<T>::Insertar_Hi (
118                         typename ArbolBinario<T>::nodo n, ArbolBinario<T> &tree) {
119     InsertarHijoIzquierda(n.p,tree.raiz);
120     tree.raiz=0; // el arbol ya forma parte de *this, no tiene
121                  // raiz sino que es hijo de n
122     return typename ArbolBinario<T>::nodo(n.p->hijoizq);
123 }
124
125 template <class T>
126 typename ArbolBinario<T>::nodo ArbolBinario<T>::Insertar_Hd(
127                         typename ArbolBinario<T>::nodo n, const T &e) {
128     /*Esta funcion elimina el hijo derecho de n e inserta una nueva rama
129     con el nodo de etiqueta e. Devuelve un nodo apuntando al nuevo hijo a
130     la derecha, el nodo de etiqueta e*/
131     InsertarHijoDerecha(n.p,e);
132     return typename ArbolBinario<T>::nodo (n->hijodcha);
133 }
134
135 template <class T>
136 typename ArbolBinario<T>::nodo ArbolBinario<T>::Insertar_Hd (
137                         typename ArbolBinario<T>::nodo n, ArbolBinario<T> &tree) {
138     InsertarHijoDerecha(n.p,tree.raiz);
139     tree.raiz=0; // el arbol ya forma parte de *this, no tiene
140                  // raiz sino que es hijo de n
141     return typename ArbolBinario<T>::nodo(n.p->hijodcha);
142 }
143
144 template <class T>
145 void ArbolBinario<T>::Podar_Hi (typename ArbolBinario<T>::nodo pos) {
146       PodarHijoIzquierda(pos.p);
147 }
148
149 template <class T>
150 void ArbolBinario<T>::Podar_Hd (typename ArbolBinario<T>::nodo pos) {
151       PodarHijoDerecha(pos.d);
152 }
153
154 // Esta funcion devuelve el hijo que hemos podado
155 ArbolBinario<T> ArbolBinario<T>::PodarHi_GetSubtree (
156                                 typename ArbolBinario<T>::nodo pos) {
157     typename ArbolBinario<T>::info_nodo * aux = Podar_HijoIzq_getSubtree(pos.p);
158     if (aux != 0)
159           aux->padre = 0;
160
161     typename ArbolBinario<T>::nodo naux(aux);
162     ArbolBinario<T> anuevo(naux);
163     return anuevo;
164 }
```

No podemos definir los operadores ++ y – en nodo porque no sabemos cómo recorrer el árbol. Hacer ++ implica saber cómo estamos recorriendo el árbol. Si queremos definir un árbol binario con todas las posibilidades para recorrerlo, debemos sobrecargar tres iteradores distintos que implementen cada uno de los recorridos que hay: iterator_preorden,iterator_inorden e

iterator_postorden

Empezamos con el preorden_iterator. Suponiendo que tiene todos los métodos implementados para la clase nodo (constructores, operador *, de igualdad, etc) el operador de incremento sería:

```
typename ArbolBinario<T>::preorden_iterator &
            ArbolBinario<T>::preorden_iterator::operator++() {

      if (p==0) //si el arbol es vacio no hay nodos que listar
            return *this;

      if (p->hijoizq != 0) //si tenemos hijo izquierdo
            p=p->hijoizq;  //el siguiente es su hijo izquierdo

      else {                        //En caso de que no tenga hijo izquierda
            if (p->hijodcha != 0)  //el siguiente es el hijo derecho
                  p=p->hijodcha;

            else {  //Cuando llegamos a una hoja:
                while (p->padre != 0 && //mientras p no sea la raiz
                      p->padre->hijodcha == 0 || //y no tenga hijo a la derecha
                      p->padre->hijodcha == p) // o el nodo no sea el hijo
                                               //a la derecha
                        p=p->padre; //subimos al padre

                  if (p->padre==0)//si hemos llegado a la raiz
                                  //ya no hay siguiente
                        p=0; //terminamos de listar si no hay hijo derecho

                  else
                        p=p->padre->hijodcha; //cuando salimos del bucle,
                                              //el siguiente es el hermano de p
            }
      }

      return *this;
}
```

Así, con tres iteradores distintos, hay tres funciones begin y tres funciones end (sin contabilizar las versiones constantes). *begin* y *end* para el iterator_preorden sería:

```
template <class T>
typename ArbolBinario<T>::preorden_iterator ArbolBinario<T>::
                                    begin_preorden()const {
      typename ArbolBinario<T>::preorden_iterator nuevo (raiz);
```

```
204         return nuevo;
205 }
206
207 template <class T>
208 typename ArbolBinario<T>::preorden_iterator ArbolBinario<T>::
209                                             end_preorden()const {
210         typename ArbolBinario<T>::preorden_iterator nuevo(0);
211         return nuevo;
212 }
```

De la misma forma el operator ++ del iterator_inorden sería:

```
211 typename ArbolBinario<T>::inorden_iterator &
212             ArbolBinario<T>::inorden_iterator::operator++() {
213
214         if (p==0) //si el arbol es vacio no hay nodos que listar
215             return *this;
216
217         if (p->hijodcha != 0) //si tenemos hijo derecha
218             p=p->hijodcha  //el siguiente es su hijo derecha
219
220         else {//En caso de que no tenga hijo izquierda
221             while (p->padre != 0 && //mientra p no sea la raiz
222              p->padre->hijodcha == p) // o yo sea el hijo a la derecha
223                   p=p->padre; //subimos al padre
224
225         }
226
227         return *this;
228 }
```

De la misma forma que con el preorden_iterador necesitamos dos funciones para iniciar una inorden_iterator (begin) y saber donde termina (end).

```
229 template <class T>
230 typename ArbolBinario<T>::inorder_iterator ArbolBinario<T>::
231                                           begin_inorden()const {
232         typename ArbolBinario<T>::inorden_iterator nuevo (raiz);
233         ++nuevo;//buscamos la hoja mas a la izquierda
234         return nuevo;
235 }
236
237 template <class T>
238 typename ArbolBinario<T>::inorden_iterator ArbolBinario<T>::
239                                             end_inorden()const {
240         typename ArbolBinario<T>::inorden_iterator nuevo(0);
```

```
241         return nuevo;
242 }
```

Y por último para el postorden_iterator tendríamos:

```
243 typename ArbolBinario<T>::postorden_iterator &
244             ArbolBinario<T>::postorden_iterator::operator++() {
245 
246         if (p==0) //si el arbol es vacio no hay nodos que listar
247               return *this;
248         if (p->padre==0)//estoy en la raiz
249           p=0;
250         else{
251          if (p->padre->hijoizq==p){   //el nodo es  el hijo a la izquierda
252           if (p->padre->hijodcha!=0){ //si tiene hermano a la derecha
253           //buscamos el siguiente por la derecha
254            p=p->padre->hijodcha;
255            do{
256              //avanzamos por la izquierda hasta que sea hoja
257              //o con hijo a la derecha
258               while (p->hizq!=0) p=p->hizq;
259               if (p->hijodcha!=0) p=p->hijodcha;
260            }while (p->hijoizq!=0 || p->hijodcha!=0);
261 
262 
263          }
264          else{ //no hay hijo a la derecha
265          p= p->padre;
266          }
267         }
268         else{// el nodo no es el hijo a la izquierda
269              //entonces el nodo es el hijo a la derecha
270          p= p->padre;
271         }
272 
273         return *this;
274 }
```

```
275 template <class T>
276 typename ArbolBinario<T>::postorden_iterator ArbolBinario<T>::
277                                             begin_postorden()const {
278         typename ArbolBinario<T>::postorden_iterator nuevo (raiz);
279         return nuevo;
280 }
281 
```

```
template <class T>
typename ArbolBinario<T>::postorden_iterator ArbolBinario<T>::
                                          end_postorden()const {
      typename ArbolBinario<T>::postorden_iterator nuevo(0);
      return nuevo;
}
```

Ejemplo 6.3.2
Crear una función que liste los nodos de un árbol binario en preorden.
Para llevar a cabo la implementación vamos a usar un preorden_iterator.

```
 typename <class T>
 void ListarPreorden(ArbolBinario<T> &a){
   typename ArbolBinario<T>::predorden_iterator it;
   for (it=a.begin(); it!=a.end();++it)
     cout<<*it<<endl;
 }
```

□

Ejercicio 6.2
Usando objetos de tipo inorden_iterator y postorden_iterator crear dos funciones para listar en inorden y postorden un árbol binario

□

Ejemplo 6.3.3
Implementar el operador – de la clase preorden_iterator de árbol binario.
Para poder implementar el operador – tenemos que conocer en dicho iterador donde está la raiz del arbol, ya que cuando el iterador es 0 el operador – debe ponerse en el último nodo que se lista hacia adelante (usando el operador ++). Así la representación de la clase preorden_iterator sería:

```
  class preorden_iterator{
     info_nodo * p;
     info_nodo * laraiz; //el nodo raiz del arbo que se lista
     ....
     }
     //ahora tambien tenemos que inicializar la laraiz en begin
     preorden_iterator begin_preorden(){
      preorden_iterator it;
      it.p =raiz;
      it.laraiz = raiz;
      return it;
     }
```

```
13
14      preorden_iterator end_preorden(){
15       preorden_iterator it;
16       it.p =0;
17       it.laraiz = raiz;
18       return it;
19      }
```

```
1  typename ArbolBinario<T>::preorden_iterator &
2              ArbolBinario<T>::preorden_iterator::operator--() {
3
4       if (p==0){ //si es el end el anterior es el ultimo
5                  //nodo en preorden
6             p=laraiz;
7             if (p!=0){
8              do{
9                while (p->hijodcha!=0) p=p->hijodcha;
10               if (p->hijoizq!=0) p=p->hijoizq;
11             }while (p->hijoizq!=0 || p->hijodcha!=0);
12            return *this;
13      }
14      else{
15       if (p->padre->hijodcha==p){ //el nodo es el hijo a la derecha
16        if (p->padre->hijoizq!=0){
17         p=p->padre->hijoizq;
18         do{
19           //avanzamos por la derecha hasta que sea hoja
20           //o con hijo a la izquierda
21            while (p->hijodcha!=0) p=p->hidcha;
22            if (p->hijoizq!=0) p=p->hijoizq;
23         }while (p->hijoizq!=0 || p->hijodcha!=0);
24        }
25        else {
26         p=p->padre; //subimos al padre
27        }
28       }
29       else{
30         p=p->padre;
31       }
32      }
33      return *this;
34 }
```

□

Ejemplo 6.3.4
Usando el iterador preorden_iterator y postorden_iterator implementar una función para deducir si dos árboles binarios son uno el reflejado del otro
Para implementar esta función usaremos el operador ++ de iterador postorden_iterator y el operador – de un iterador preorden_iterator.

```
1   typename <class T>
2   bool Reflejados (ArbolBinario<T> &a1, ArbolBinario<T> &a2){
3       typename ArbolBinario<T>::iterator_preorden itpre=a2.end();
4       //retrocedemos al ultimo
5       --itpre;
6       typename ArbolBinario<T>::iterator_preorden itpost=a1.begin();
7
8      while (itpre!=a2.end() && itpost!=a1.end() && *itpre==*itpost){
9       --itpre; ++itpost
10     }
11     //si los dos no han llegado al final
12     if (itpre!=a2.end() && itpost!=a1.end()) return false;
13     else
14     return true;
15  }
```

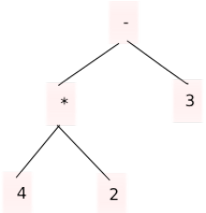
□

Ejercicio 6.3
Sobrecargar los operadores – para la clase postorden_iterator e inorden_iterator.

□

6.3.3 Expresiones Algebraicas

Ya vimos en el capítulo de estructuras lineales las expresiones algebráicas como una aplicación de uso de la pila, para pasar una expresión algebraica a notación Polaca o notación inorden. En esta sección veremos que se puede usar un árbol binario para almacenar una expresión algebraica (compuesta de operadores: +, -, /, *) en notación inorden (con la que estamos acostumbrados a trabajar p.e 4*2-3) o a notación prefijo o polaca (p.e - * 4 2 3), o notación postfijo (p.e 4 2 * 3 -).



Para ello vamos a implementar el TDA Expresión que contiene una expresión en notación inorden, usando para su representación un ArbolBinario. A este TDA le añadiremos métodos

para poder evaluar la expresión dando un resultado escalar, obtener la expresión Polaca o prefijo equivalente y obtener la expresión postfijo. De esta forma el TDA Expresion sería el siguiente:

```
class Expresion{
private:
    ArbolBinario<string> datos;

public:
  Expresion(){}
  /**
   * @brief Inicia una expresion con la cadena de entrada
   * @note si la cadena no tiene valores correctos la expresion se inicia a vacio
   */
  Expresion(const string &e);

  /**
   * * @brief Evalua la expresion deovolviendo el resultado
   */
  float Evalua()const;

  /**
   * * @brief Obtiene la notacion en prefijo
   *
   */
  string Expresion_Prefijo()const;


  /**
   * * @brief Obtiene la notacion en postfijo
   *
   */
  string Expresion_Postfijo()const;

};
```

Antes de ver la implementación de estas funciones nos hará falta algunas funciones que nos diga si dada una expresión lo que viene a continuación es un operador o un operando y obtenerlo en caso afirmativo. Además tendremos una función QuitarBlancos que nos elimina todos los espacios que haya al principio de una expresión.

```
#include "expresion.h"
#include <sstream>

/*************************************************************/
void QuitarBlancos(string &expresion){
```

```
6     while (expresion.size()>0 && expresion[0]==' '){
7        expresion= expresion.substr(1,string::npos);
8     }
9   }
10  /*************************************************************/
11   bool Operador(string &expresion, char &operador){
12
13        QuitarBlancos(expresion);
14        if (expresion.size()>0){
15          if (expresion[0]=='+' || expresion[0]=='-' ||
16            expresion[0]=='*' || expresion[0]=='/'){
17              operador = expresion[0];
18              expresion= expresion.substr(1,string::npos);
19              return true;
20          }
21        }
22        return false;
23   }
24  /*************************************************************/
25  template <class T>
26  void GetOperando(string & expresion,T &operando){
27        QuitarBlancos(expresion);
28        if (expresion.size()>0){
29            stringstream ss;
30            string aux;
31            ss.str(expresion);
32            ss>>aux; //hasta el primer separador
33            //le quitamos a expresion lo leido en aux
34            expresion=expresion.substr(aux.size(),string::npos);
35            //convertimos de string a int
36
37            ss.clear();
38            ss.str(aux);
39            ss>>operando;
40        }
41  }
42  /*************************************************************/
43  bool isOperator(string expresion){
44            if (expresion[0]=='+' || expresion[0]=='-' ||
45            expresion[0]=='*' || expresion[0]=='/'){
46            return true;
47          }
48          else return false;
```

```
49 }
```

Ahora si veamosla implementación de los métodos:

```
1  Expresion::Expresion(const string &e){
2    string expresion = e;
3
4    QuitarBlancos(expresion);
5    //inicializamos el arbol con los tres primeros elementos
6    string op1;
7    GetOperando(expresion, op1);//el operando izquierdo
8    QuitarBlancos(expresion);
9
10   char operacion;
11   if (Operador(expresion,operacion)){//la operacion
12     string op2;
13
14     QuitarBlancos(expresion);
15     GetOperando(expresion, op2);//el operando derecho
16     //inicializamos el arbol
17     string oper; oper.push_back(operacion);
18     datos=ArbolBinario<string>(oper);
19     datos.Insertar_Hi(datos.getRaiz(),op1);
20     datos.Insertar_Hd(datos.getRaiz(),op2);
21
22     //ahora vamos leyendo de dos en dos:operador operando derecho
23     while (expresion.size()>0){
24       QuitarBlancos(expresion);
25       string op;
26       if (Operador(expresion,operacion)){
27         QuitarBlancos(expresion);
28         GetOperando(expresion, op2);
29         string oper; oper.push_back(operacion);
30         ArbolBinario<string> aux(oper);
31         aux.Insertar_Hd(aux.getRaiz(),op2);
32         aux.Insertar_Hi(aux.getRaiz(),datos);
33         datos=aux;
34
35       }
36       else {
37         datos=ArbolBinario<string>();
38         return ;
39       }
40
41     }
```

```
42
43    }
44    else return;
45  }
46  /*************************************************************/
47  float Expresion::Evalua()const{
48    float res=0.0;
49    ArbolBinario<string>::inorden_iterador in=datos.begininorden();
50    float left_op,right_op;
51    string op;
52    while (in!=datos.endinorden()){
53     if (isOperator(*in)){
54       //leemos el siguiente en inorden
55       op=*in;
56       ++in;
57       string aux =*in;
58       GetOperando(aux,right_op);
59       switch (op[0]){
60         case '+':
61              res = left_op+right_op;
62              break;
63         case '-':
64              res = left_op-right_op;
65              break;
66         case '*':
67              res = left_op*right_op;
68              break;
69         case '/':
70              res = left_op/right_op;
71              break;
72
73       }
74       left_op=res;
75
76       ++in;
77     }
78
79     else{
80        string aux =*in;
81        GetOperando(aux,left_op);
82        ++in;
83     }
84
```

```
85    }
86    return res;
87  }
88  /*************************************************************/
89
90  string Expresion::Expresion_Prefijo()const{
91    ArbolBinario<string>::preorden_iterador pre=datos.beginpreorden();
92    string salida="";
93    for(;pre!=datos.endpreorden();++pre){
94       salida=salida+*pre+ " ";
95    }
96    return salida;
97
98  }
99  /*************************************************************/
100 string Expresion::Expresion_Postfijo()const{
101   ArbolBinario<string>::postorden_iterador post=datos.beginpostorden();
102   string salida="";
103   for(;post!=datos.endpostorden();++post){
104      salida=salida +*post+ " ";
105   }
106   return salida;
107
108 }
```

Con respecto a las funciones Expresion_Prefijo y Expresion_Posfijo, simplemente hace falta usar un iterador y recorrer el árbol en el sentido de dicho iterador.

6.4 Arboles generales

Para representar un árbol general, cada nodo contendrá su etiqueta y punteros al padre, al hijo a la izquierda y al hermano a la derecha:

```
1  template <class T>
2  struct info_nodo {
3        T et;
4        info_nodo<T> * padre, * hijoizq, * hermanodcha;
5        info_nodo() {
6              padre = hijoizq = hermanodcha = 0;
7        }
8        infonodo(const T & e) {
9              et = e;
10             padre = hijoizq = hermanodcha = 0;
11       }
12 }
```