



6. Estructuras de datos no lineales. Árboles

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 en primer lugar abordaremos 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 entrada¹ 1 (excepto el nodo raíz que tiene grado de entrada 0) y el grado de salida² 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.
3. *interiores*: el resto de nodos.

Algunas características de los árboles son:

1. Todos los nodos descienden de la raíz.

¹número de líneas que entran al nodo

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

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

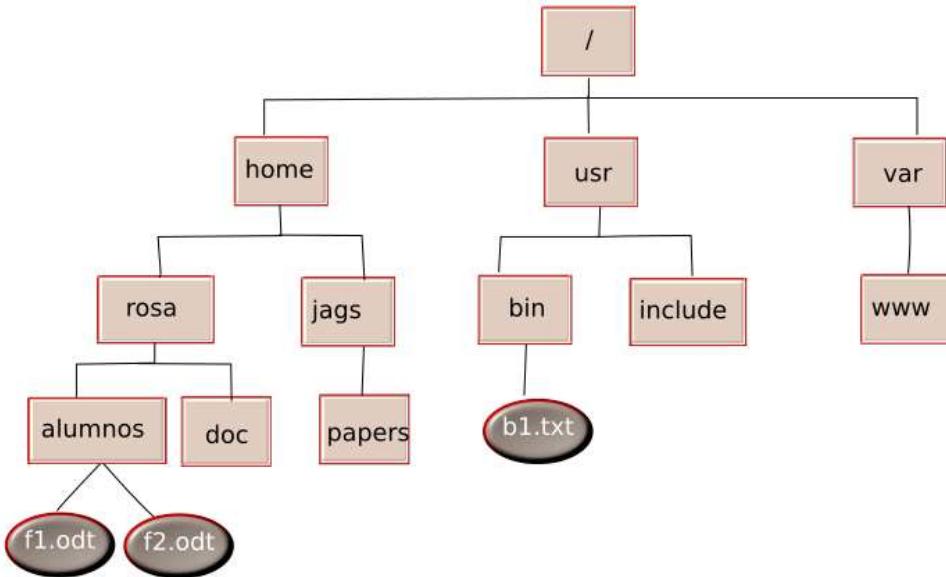


Figura 6.1: Ejemplo de árbol: estructura de directorios

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, \dots, 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:

/ home rosa alumnos 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:

$n_s \dots n_j \dots n_i \dots n_l$

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 sería 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.

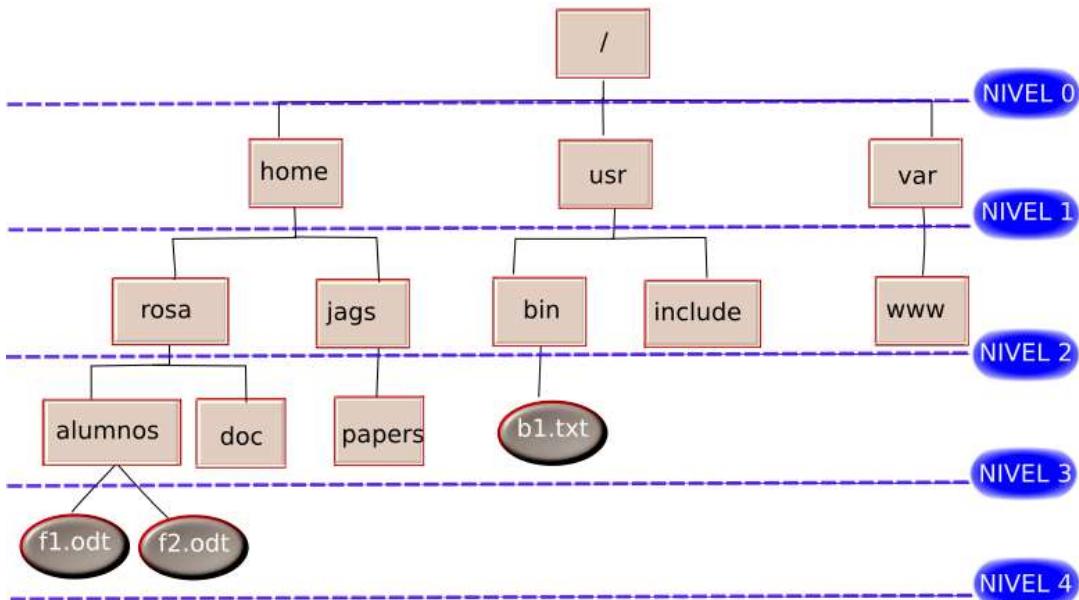


Figura 6.2: Niveles de un árbol

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 vacío³, también se 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 nivel (a partir de ahora lo llamaremos nivel inferior), en cuyo caso, los huecos quedan a la derecha. Por ejemplo:

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

³el árbol que no tiene ningún nodo

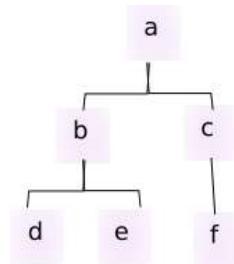


Figura 6.3: Árbol Binario completo

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

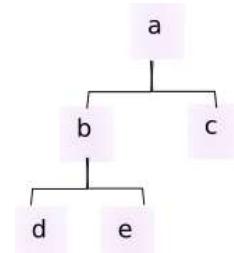


Figura 6.4: Árbol Binario homogéneo y completo

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:

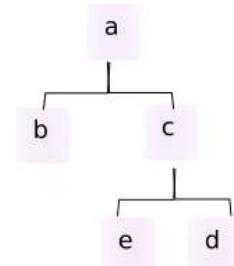


Figura 6.5: Árbol Binario homogéneo pero no es completo

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:

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

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

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

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

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 raíz 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 raíz 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:

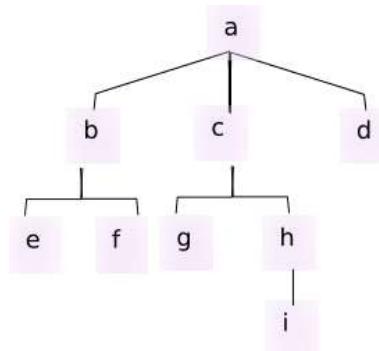


Figura 6.6: Árbol General

El preorden de este ejemplo 6.6, sería:

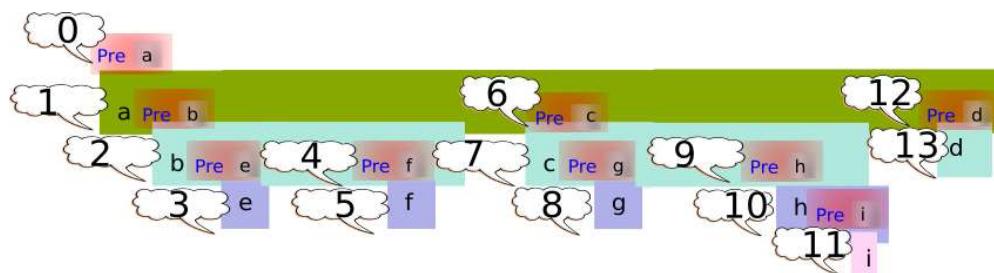


Figura 6.7: Recorrido preorden del árbol de la figura 6.6

Lo primero que se hace es llamar a Pre con el nodo a ($\text{Pre}(a)$) . 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 $\text{Pre}(b)$
2. Listamos la raíz de este subárbol, b, y nos vamos al subárbol del hijo más a la izquierda de b: e. Hacemos $\text{Pre}(e)$
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 $\text{Pre}(f)$
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 $\text{Pre}(c)$, el siguiente hijo de a.
7. Listamos la raíz del subárbol formado por c y hacemos $\text{Pre}(g)$, 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 $\text{Pre}(h)$
10. Listamos h, hacemos $\text{Pre}(i)$, 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:

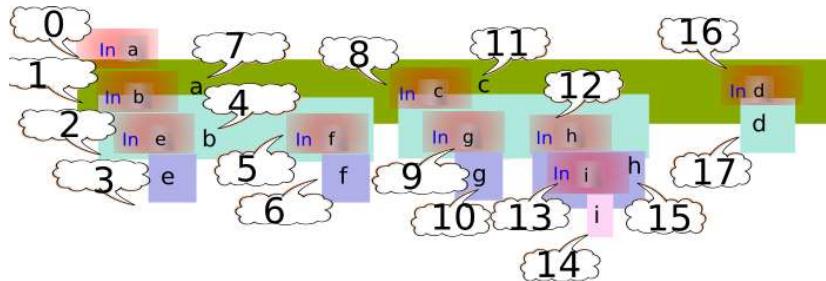


Figura 6.8: Recorrido inorden del árbol de la figura 6.6

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:

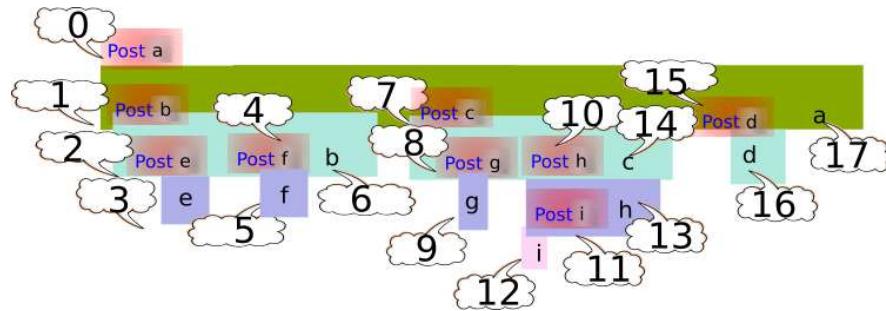


Figura 6.9: Recorrido postorden del árbol de la figura 6.6

Tras 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 izquierda 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. Despues 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 f b g i h c d 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:



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

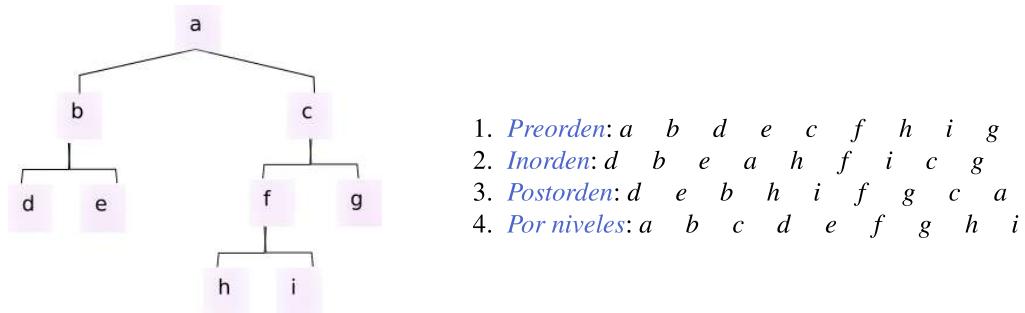


Figura 6.10: Árbol Binario

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.

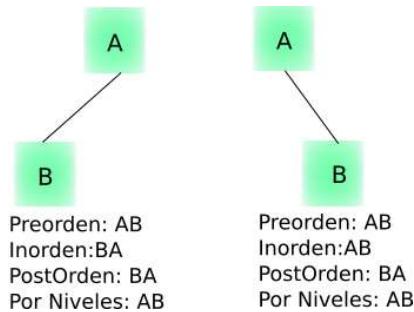


Figura 6.11: Dos árboles binario con sus recorridos

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

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 única si los listados que nos dan son:

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

No podemos 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.



¿
dos listados
puede rec
el árbol?

Ejercicio 6.1

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

□

Por lo general, con sólo uno de los recorridos de un árbol, no puede recuperarse de manera unívoca, 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 derecha⁴ del árbol de la izquierda y ambos tienen el mismo inorden:

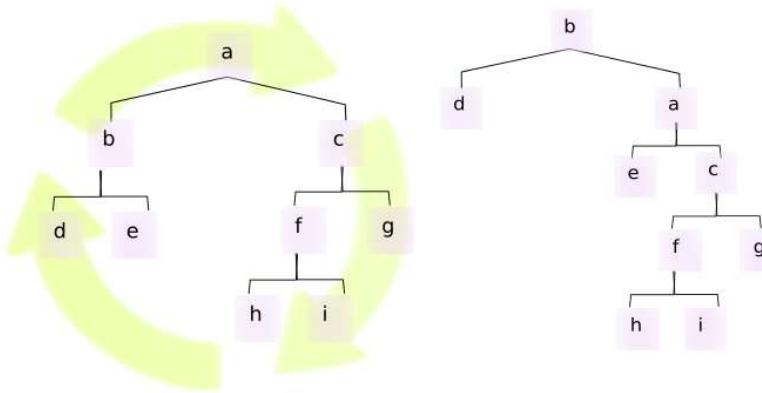


Figura 6.12: Dos árboles binarios con igual recorrido en inorden. El árbol de la derecha se obtiene como una rotación simple a derecha aplicado al árbol de la izquierda.

El inorden de ambos árboles es:

d b e a h f i c 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
$n \in \text{descendiente}(m)$	F	V si es descendiente por la izquierda y F si lo es por la derecha	V
$n \in \text{ancestro}(m)$	V	igual que antes	F

Ejemplo 6.2.1

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

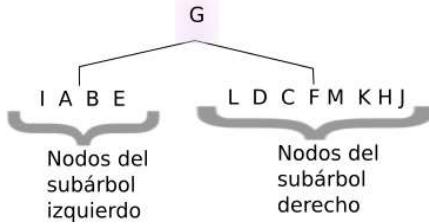
1. *preorden*: G E A I B M C L D F K J H

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

Tenemos que fijarnos en los siguientes detalles:

⁴Desplazamos 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.

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

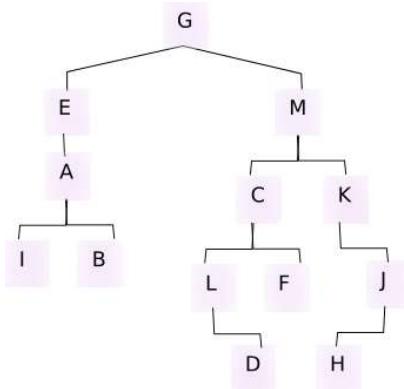


Figura 6.13: Árbol resultante del listado en preorden e inorden

□

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:

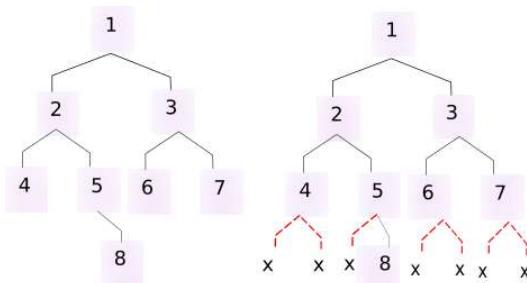


Figura 6.14: Izquierda: Arbol original. Derecha: Árbol transformado para aplicar la lectura/escritura

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 Árboles binarios

En esta sección 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:

```

1 #include <queue> //para hacer el recorrido por niveles
2 using namespace std;
3 template <class T>
4 struct info_nodo {
5     info_nodo *padre, //puntero al padre
6     *hijoizq, //puntero al hijo izquierda
7     *hijodcha; // puntero al hijo derecha
8     T et; // etiqueta del nodo
9
10    // Constructor por defecto del struct
11    info_nodo() {
12        padre = hijoizq = hijodcha = 0;
13    }
14
15    info_nodo(const T &e) {
16        et = e;
17        padre = hijoizq = hijodcha = 0;
18    }
19};
```

Como se puede observar hemos incluido constructores: por defecto y por parámetros. En la Figura 6.15 se puede ver la estructura *info_nodo*. También 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.

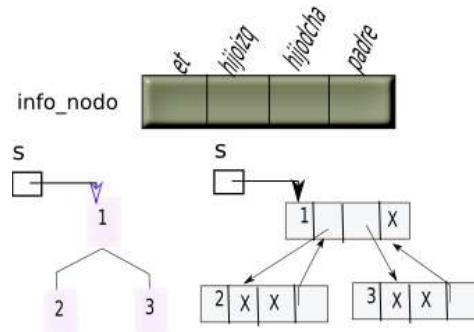


Figura 6.15: Arriba representación de un info_nodo. A la izquierda un árbol binario, y a la derecha como se rellenan los campos de los nodos que cuelgan de s

Las operaciones a implementar en un árbol binario serían:

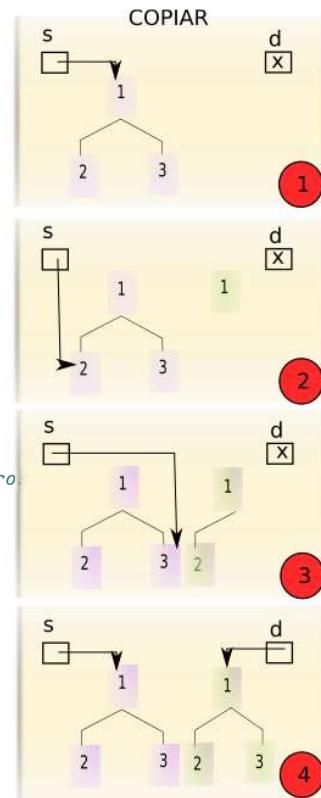
1. *Get*: parent, hijoizq, hijodcha, etiqueta
2. *Insertar*: hijoizq, hijodcha
3. *Podar*: hijoizq, hijodcha
4. *Recorridos*: preorden, inorden, postorden, anchura
5. *Leer/escribir*: lee/escribe un árbol binario en un flujo
6. *Copiar*: copia un árbol binario en otro
7. *Borrar*: elimina toda la memoria 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 mínimo 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 parametro
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   }

```

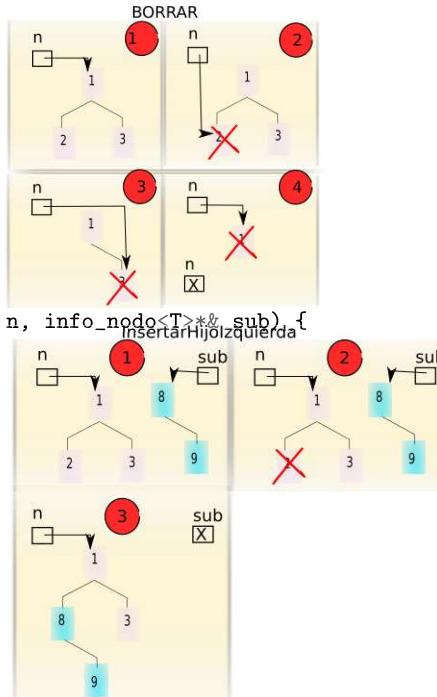


En la imagen que acompaña al anterior código se puede ver un ejemplo del proceso de Copiar. Inicialmente se llama con un puntero a *info_nodo*, *s* (árbol fuente), que contiene la dirección de la raíz del árbol fuente y *d* (árbol destino) con valor 0. En este paso se crea un nuevo objeto *info_nodo* 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 también es 0. Por lo tanto se vuelve de la recursividad al nodo 1 y se llama a copiar con el hijo a la derecha. Una 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 estas dos funciones para que en vez de pasarle un subárbol le pasemos una etiqueta.

```

88 //Hacemos una sobrecarga de esta función
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 }

```

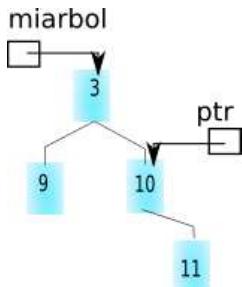
Así con esta sobrecarga permite al usuario de árbol binario hacer crecer su árbol. Por ejemplo un código para crear un arbol sería el siguiente:

```

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 InsertarHijoDerecha(ptr,11);

```

Tras 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 nodo 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 subárbol que se poda, y por lo tanto no borrar la memoria del subárbol. 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 misma 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 raíz n tiene 1 nodo por la raíz, más el número de nodos que tenga el subárbol izquierdo (esto se implementa con una llamada recursiva) y otros tantos nodos por el subárbol derecho (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 etiquetas⁵. La diferencia de estas tres funciones es donde se pone la salida de la etiqueta del nodo n . Así 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*/
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
```

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

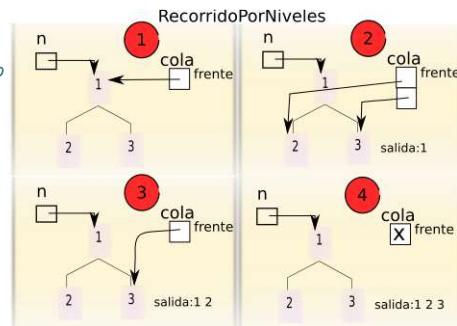
```

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

```

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

```
1 template <class T>
2 class ArbolBinario{
3     private:
4         struct info_nodo{
5             T et;
6             info_nodo * padre;
7             info_nodo * hizq;
8             info_nodo * hder;
9             info_nodo(){ padre=hizq=hder=0; }
10            info_nodo(const T & e){ et = e; padre=hizq=hder=0; }
11        };
12        //Funciones asociadas a info_nodo
13        void Copiar(info_nodo * &dest,const info_nodo*const &source);
14
15        void BorrarInfo(info_nodo *&d);
16
17        unsigned int numero_nodos(const info_nodo*d)const ;
18
19        bool iguales(const info_nodo*s1,const info_nodo*s2)const ;
20
21        void InsertarHijoIzquierda(info_nodo * n,info_nodo * sub);
22
23        void InsertarHijoIzquierda(info_nodo * n,const T & e);
24
25        void InsertarHijoDerecha(info_nodo * n, info_nodo * sub);
26
27        void InsertarHijoDerecha(info_nodo * n,const T & e);
28
29        void PodarHijoIzquierda(info_nodo * n);
30
31        void PodarHijoDerecha(info_nodo * n);
32
33        info_nodo *PodarHijoIzq_GetSubtree(info_nodo * n);
34
35        info_nodo *PodarHijoDer_GetSubtree(info_nodo * n);
36
37        void RecorridoPreorden(ostream & os, const info_nodo *n)const ;
38
39        void RecorridoPostorden(ostream & os,const info_nodo *n)const ;
40
41        void RecorridoInorden(ostream & os,const info_nodo *n)const ;
42
43        void RecorridoNiveles(ostream &os,const info_nodo *n)const ;
44
45        void Lee(istream & is, info_nodo *&n);
46
47        void Escribe(ostream & os,const info_nodo *n)const;
48
49    ...
```

```

50 //ahora como se representa un ArbolBinario
51 };

```

Con estas funciones privadas ahora la representación:

```

1 template <class T>
2 class ArbolBinario{
3     private:
4         struct info_nodo{
5             T et;
6             info_nodo * padre;
7             info_nodo * hizq;
8             info_nodo * hder;
9             info_nodo(): padre=hizq=hder=0; }
10            info_nodo(const T & e){ et = e; padre=hizq=hder=0; }
11        };
12     //Funciones asociadas a info_nodo
13     ...
14     info_nodo *raiz; //raiz del arbol binario
15
16 ...
17 };

```

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.

```

1 template <class T>
2 class ArbolBinario{
3     private:
4         ...
5     public:
6
7     class nodo{
8         private:
9             info_nodo *p;
10            nodo (info_nodo * i):p(i){}
11        public:
12
13            nodo () :p(0){}
14
15            const T& operator*()const {
16                assert(p!=0);
17                return p->et;
18            }
19            T& operator*() {
20                assert(p!=0);
21                return p->et;
22            }
23
24            bool operator==(const nodo &n){
25                return p==n.p;
26            }
27
28            bool operator!=(const nodo &n){

```

```

29             return p!=n.p;
30     }
31
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 siguientes secciones.

Para hacer uso de un nodo tendremos que usar la sintaxis:

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 habiendo 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     ArbolBinario(const T &e);
12     ArbolBinario(typename ArbolBinario<T>::nodo n);
13     /**
14      @brief Constructor por copia
15      */
16     ArbolBinario(const ArbolBinario<T> & ab);
17
18     /**
19      @brief Destructor
20      */
21     ~ArbolBinario(){ clear(); }
22
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     /**
36      @brief Inserta un subarbol como hijo izquierdo del nodo.
37      Este suárbol solamente tiene un nodo
38      @param n: posicion del nodo donde insertar el subarbol como hijo izquierdo
39      @param e: etiqueta de la raiz del subarbol que se inserta
40
41     typename ArbolBinario<T>::nodo Insertar_Hi( typename ArbolBinario<T>::nodo n,
42                                         const T &e);
43
44     /**
45      @brief Inserta un subarbol como hijo izquierdo del nodo.
46      @param n: posicion del nodo donde insertar el subarbol como hijo izquierdo
47      @param tree:subarbol que se inserta. ES MODIFICADO
48
49     typename ArbolBinario<T>::nodo Insertar_Hi( typename ArbolBinario<T>::nodo n ,
50                                         ArbolBinario<T> & tree);
51
52     /**
53      @brief Inserta un subarbol como hijo derecho del nodo.
54      Este suárbol solamente tiene un nodo
55      @param n: posicion del nodo donde insertar el subarbol como hijo derecho
56      @param e: etiqueta de la raiz del subarbol que se inserta
57
58     typename ArbolBinario<T>::nodo Insertar_Hd( typename ArbolBinario<T>::nodo n,
59                                         const T &e);
60
61     /**
62      @brief Inserta un subarbol como hijo derecho del nodo.
63      @param n: posicion del nodo donde insertar el subarbol como hijo derecho
64      @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 izquierdo 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 árboles
121      @param a: árbol binario con el que se compara
122      @return true si los dos árboles son iguales false en caso contrario
123 */
124     bool operator==(const ArbolBinario<T> &a) const;
125
126     /**
127      @brief Desigualdad entre dos árboles
128      @param a: árbol binario con el que se compara
129      @return true si los dos árboles 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 árbol en preorden
137 */
138     void RecorridoPreOrden(ostream &os) const ;
139
140     /**
141      @brief Recorrido en Inorden
142      @param os: flujo sobre el que se da el recorrido del árbol en Inorden
143 */
144     void RecorridoInOrden(ostream &os) const ;
145
146     /**
147      @brief Recorrido en Postorden
148      @param os: flujo sobre el que se da el recorrido del árbol en Postorden
149 */
150     void RecorridoPostOrden(ostream &os) const ;
151
152     /**
153      @brief Recorrido por niveles
154      @param os: flujo sobre el que se da el recorrido del árbol por niveles
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 árbol 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 árbol 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:

```

165  typename ArbolBinario<T>::preorden_iterator &
166      ArbolBinario<T>::preorden_iterator::operator++() {
167
168      if (p==0) //si el arbol es vacio no hay nodos que listar
169          return *this;
170
171      if (p->hijoizq != 0) //si tenemos hijo izquierdo
172          p=p->hijoizq; //el siguiente es su hijo izquierdo
173
174      else { //En caso de que no tenga hijo izquierdo
175          if (p->hijodcha != 0) //el siguiente es el hijo derecho
176              p=p->hijodcha;
177
178          else { //Cuando llegamos a una hoja:
179              while (p->padre != 0 && //mientras p no sea la raiz
180                  p->padre->hijodcha == 0 || //y no tenga hijo a la derecha
181                  p->padre->hijodcha == p) // o el nodo no sea el hijo
182                      //a la derecha
183                  p=p->padre; //subimos al padre
184
185              if (p->padre==0)//si hemos llegado a la raiz
186                  //ya no hay siguiente
187              p=0; //terminamos de listar si no hay hijo derecho
188
189          else
190              p=p->padre->hijodcha; //cuando salimos del bucle,
191                  //el siguiente es el hermano de p
192      }
193  }
194
195  return *this;
196 }
```

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:

```

200  template <class T>
201  typename ArbolBinario<T>::preorden_iterator ArbolBinario<T>::
202      begin_preorden()const {
203      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 && //mientras 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 preorder_iterador necesitamos dos funciones para iniciar una inorden_iterator (begin) y saber donde termina (end).

```

229 template <class T>
230 typename ArbolBinario<T>::inorden_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             }
255         }
256     }
257 }
```

```

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
282 template <class T>
283 typename ArbolBinario<T>::postorden_iterator ArbolBinario<T>::
284             end_postorden()const {
285     typename ArbolBinario<T>::postorden_iterator nuevo(0);
286     return nuevo;
287 }

```

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.

```

1  typename <class T>
2  void ListarPreorden(ArbolBinario<T> &a){
3      typename ArbolBinario<T>::predorden_iterator it;
4      for (it=a.begin(); it!=a.end(); ++it)
5          cout<<*it<<endl;
6  }

```



Ejercicio 6.2

Usando objetos de tipo inorder_iterator y postorden_iterator crear dos funciones para listar en inorder 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 raíz 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:

```

1  class preorden_iterator{
2      info_nodo * p;
3      info_nodo * laraiz; //el nodo raiz del arbol que se lista
4      ....
5  }
6  //ahora tambien tenemos que inicializar la laraiz en begin
7  preorden_iterator begin_preorden(){
8      preorden_iterator it;
9      it.p =raiz;
10     it.laraiz = raiz;
11     return it;
12 }
13
14 preorden_iterator end_preorden(){
15     preorden_iterator it;
16     it.p =0;
17     it.laraiz = raiz;
18     return it;
19 }
20
21 typename ArbolBinario<T>::preorden_iterator &
22         ArbolBinario<T>::preorden_iterator::operator--() {
23
24     if (p==0){ //si es el end el anterior es el ultimo
25         //nodo en preorden
26         p=laraiz;
27         if (p!=0){
28             do{
29                 while (p->hijodcha!=0) p=p->hijodcha;
30                 if (p->hijoizq!=0) p=p->hijoizq;
31             }while (p->hijoizq!=0 || p->hijodcha!=0);
32             return *this;
33         }
34         else{
35             if (p->padre->hijodcha==p){ //el nodo es el hijo a la derecha
36                 if (p->padre->hijoizq!=0){
37                     p=p->padre->hijoizq;
38                     do{
39                         //avanzamos por la derecha hasta que sea hoja
40                         //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

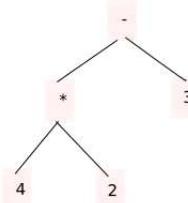
Sobrecargar los operadores – para la clase postorden_iterator e inorden_iterator.

□

6.3.3 Expresiones Algebraicas

Ya vimos en el capítulo de estructuras lineales las expresiones algebraicas 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 inorder (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 inorder, 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 Expresión sería el siguiente:

```

1 class Expresion{
2     private:
3         ArbolBinario<string> datos;
4
5     public:
6         Expresion(){}
7         /**
8          * @brief Inicia una expresion con la cadena de entrada
9          * @note si la cadena no tiene valores correctos la expresion se inicia a vacio
10         */
11        Expresion(const string &e);
12
13        /**
14         * * @brief Evalua la expresion devolviendo el resultado
15         */
16        float Evalua()const;
17
18        /**
19         * * @brief Obtiene la notacion en prefijo
20         */
21        string Expresion_Prefijo()const;
22
23
24        /**
25         * * @brief Obtiene la notacion en postfijo
26         */
27        string Expresion_Postfijo()const;
28
29
30    };
  
```

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.

```
1 #include "expresion.h"
2 #include <sstream>
3
4 /*****
5 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 veamos la 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 inorder
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             left_op=res;
74
75             ++in;
76         }
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 Árboles 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 }
```

En la figura 6.16 se puede ver a la derecha como se representa el árbol que se muestra a la izquierda. Como se puede ver un nodo ahora tiene tres punteros: al padre , hijo mas a la izquierda y hermano a la derecha del nodo.

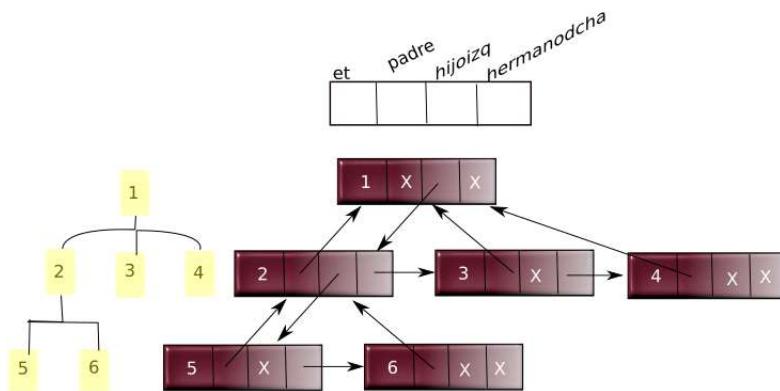


Figura 6.16: Ejemplo de un árbol general y como se representa

De esta forma la representación de Árbol General sería de la siguiente forma:

```

1  typename <class T>
2  class ArbolGeneral{
3      private:
4          info_nodo<T> *raiz;
5
6          //aqui todas las funciones privadas
7
8      public:
9          //aqui la interfaz de ArbolGeneral
10
11
12      ...
13  };

```

Veámos a continuación las funciones asociadas a la parte privada de la clase Árbol General. En primer lugar vamos a implementar la función Copiar, que copia las etiquetas de un conjunto de nodos en un nuevo conjunto de nodos.

```

1  template <class T>
2  void ArbolGeneral<T>::Copiar(info_nodo<T>* s, info_nodo<T>* &d) {
3      if (s==0)
4          d = 0;
5
6      else {
7          d = new info_nodo<T>(s->et);
8          Copiar(s->hijoizq,d->hijoizq);
9          Copiar(s->hermanodcha,d->hermanodcha);
10         // le asignamos a los nodos que hemos copiado su padre
11         if (d->hijoizq != 0){
12             d->hijoizq->padre=d;
13             for (info_nodo<T> aux = d->hijoizq->hermanodcha;
14                  aux!=0;aux= aux->hermanodcha)
15                 aux->padre= d;
16         }
17     }
18 }

```

De esta forma el conjunto de nodos origen se indica por el puntero que tiene la variable s. Y el conjunto de nodos destino se identifican desde la variable d. En primero lugar si s no apunta a nada (es decir 0) entonces d también lo hará. En otro caso se solicita nueva memoria para un info_nodo con igual etiqueta que la etiqueta que contiene el nodo al que apunta s. Recursivamente se aplica el procedimiento de copiar para el hijo más a la izquierda y el hermano a la derecha. Un vez realizada la copia hace falta ajustar los padres de los hijos del nodo al que apunta d. Para ello se ejecutan las líneas 11-15.

Otro método privado que se define es Destruir que permite liberar toda la memoria que cuelga desde un nodo dado.

```

1  template<class T>
2  void ArbolGeneral<T>::Destruir (info_nodo<T>* t) {
3      // Debemos empezar con el hermano a la derecha del ultimo nodo hoja
4      // del arbol, si no lo hacemos en este orden, perdemos los enlaces.

```

```

5   // Es decir, para destruir el arbol tenemos que hacerlo en recorrido
6   // postorden
7   if (t != 0) {
8       Destruir(t->hijoizq);           // cada hijo resuelve su destrucción
9       Destruir(t->hermanodcha);     // antes de hacer el delete
10      delete t;
11  }
12  // cuando t es cero no entra al if, vuelve a la llamada recursiva
13  // y hace el siguiente paso.
14 }
```

Este método libera la memoria en primer lugar de todos los hijos y a continuación la memoria de los hermanos a la derecha. La función Copiar se usará para implementar el constructor de copia como el operador de asignación. Destruir se usará en la implementación del operador de asignación y el destructor. De esta forma tenemos que:

```

1 template<class T>
2 ArbolGeneral<T>::ArbolGeneral(const ArbolGeneral<T> & ag){
3     Copiar(ag.raiz,raiz);
4 }
5
6 template<class T>
7 ArbolGeneral<T>::~ArbolGeneral(){
8     Destruir(raiz);
9 }
10
11
12 template<class T>
13 ArbolGeneral<T> & ArbolGeneral<T>::operator=(const ArbolGeneral<T> & ag){
14     if (this!=&ag){
15         Destruir(raiz);
16         Copiar(ag.raiz,raiz);
17     }
18     return *this;
19 }
```

A continuación presentamos dos métodos que hacen crecer en número de nodos del árbol. El primer método permite insertar un nuevo hijo a la izquierda a un nodo. El actual hijo más a la izquierda del nodo pasa a ser el hermano a la derecha del nuevo hijo a la izquierda.

```

1 template <class T>
2 void ArbolGeneral<T>::InsertarHijoIzquierda (info_nodo<T>* n, info_nodo<T>* &t2) {
3     // El hijo a la izquierda de n pasaria a ser el hermano a la derecha
4     // de t2
5     if (t2 != 0) {
6         t2->hermanodcha = n->hijoizq;  t2 no se destruye?
7         t2->padre=n;
8         n->hijoizq=t2;
9         t2=0;
10    }
11 }
12 }
```

El segundo método, *InsertarHermanoDerecha* inserta un nuevo hermano a la derecha de un nodo. El hermano derecho actual pasa a ser el hermano derecho del nuevo nodo.

```

1 template <class T>
2 void ArbolGeneral<T>::InsertarHermanoDerecha (info_nodo<T>* n,
3                                                 info_nodo<T>* &t2) {
4     if (t2 != 0) {
5         t2->hermanodcha = n->hermanodcha;
6         t2->padre = n;
7         n->hermanodcha = t2;
8         t2 = 0;
9     }
10 }
```

Como contraposición a los anteriores métodos ahora vemos dos funciones que hacen decrecer el numero de nodos del árbol. La primera función elimina del árbol todo lo que cuelga a partir del hijo más a la izquierda de un nodo dado. Para realizar el proceso de forma correcta el nodo adoptará como nuevo hijo más a la izquierda el hermano del hijo a la izquierda que se quita. Adicionalmente el subárbol hijo más a la izquierda que se quita se devuelve como un árbol.

```

1 template <class T>
2 info_nodo<T>* ArbolGeneral<T>::PodarHijoIzquierda (info_nodo<T>* n) {
3     info_nodo<T>* res = 0; // creamos un nodo auxiliar
4     if (n->hijoizq != 0) {
5         // res apunta al subárbol hijo izquierda
6         res = n->hijoizq;
7         // el hijo a la izquierda del parente
8         // pasa a ser el hermano a la derecha
9         // del que era hijo a la izquierda
10        n->hijoizq = res->hermanodcha;
11        // y el hijo a la izquierda queda como
12        // la raíz del arbol a devolver
13        res->padre = res->hermanodcha=0;
14
15    }
16    return res;
17 }
```

De la misma forma podemos quitar el subárbol hermano a la derecha de un nodo dado. Para ello tendremos que poner como nuevo subárbol a la derecha el nodo que fuese hermano a la derecha del que quitamos.

```

1 template <class T>
2 info_nodo<T>* ArbolGeneral<T>::PodarHermanoDerecha (info_nodo<T>* n) {
3     info_nodo<T>* res = 0;
4     if (n->hermanodcha != 0) {
5         // res apunta al hermano a la derecha a eliminar
6         res = n->hermanodcha;
7         // le ponemos un nuevo hermano a la derecha.
8         n->hermanodcha = res->hermanodcha;
9         //res será un arbol para ello
10        //no tiene padre ni hermano a la derecha
11        res->padre = res->hermanodcha = 0;
12    }
13
14    return res;
15 }
```

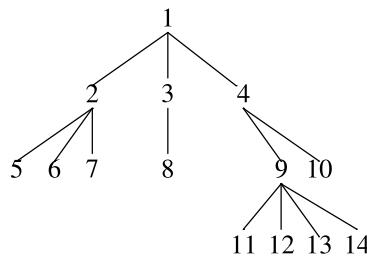
Finalmente el siguiente método obtiene la altura de un árbol. Recordad que la altura se define como el camino más largo que va desde la raíz a una hoja.

```

1 template <class T>
2 int ArbolGeneral<T>::altura (info_nodo<T>* t) {
3     // las hojas tendrán altura cero y la raíz la altura máxima
4     if (t == 0)
5         return -1;
6
7     else {
8         int max = -1;
9         info_nodo<T>* aux;
10        // recorremos los hijos del nodo
11        for (aux=t->hijoizq;aux!=0;aux=aux->hermanodcha) {
12            // comprobamos si la altura de los hijos es mayor a la máxima
13            // que tenemos ya calculada
14
15            //altura del nodo
16            int alturahijo = altura(aux);
17
18            if (alturahijo > max)
19                max = alturahijo;
20        }
21        // la altura de los hijos mas 1 por el padre
22        return max+1;
23    }
24 }
```

6.4.1 Recorridos en árboles generales

Los distintos recorridos del siguiente árbol serían:



Preorden: 1 2 5 6 7 3 8 4 9 11 12 13 14 10
Inorden: 5 2 6 7 1 8 3 11 9 12 13 14 4 10
Postorden: 5 6 7 2 8 3 11 12 13 14 9 10 4 1
Niveles: 1 2 3 4 5 6 7 8 9 10 11 12 13 14

En C++ la implementación de los recorridos serían:

```

1  template <class T>
2  void ListarPreorden (info_nodo<T>* t) {
3      if (t != 0) {
4          cout << t->et << ' '; // primero listamos la raiz
5          info_nodo<T>* aux; // y luego sus hijos
6          for (aux=t->hijoizq;aux!=0;aux=aux->hermanodcha)
7              ListarPreorden(aux);
8      }
9  }
10
11 template <class T>
12 void ListarInorden (info_nodo<T>* n) {
13     if (n != 0) {
14         ListarInorden (n->hijoizq); // listamos el hijo a la izquierda
15         cout << n->et << ' '; // despues la raiz
16         info_nodo<T>* aux=n->hijoizq;
17         if (aux != 0) {
18             aux = aux->hermanodcha; // y luego los hijos a la
19             while (aux!=0) { // derecha
20                 ListarInorden(aux);
21                 aux = aux->hermanodcha;
22             }
23         }
24     }
25 }
26
27 template <class T>
28 void ListarPostorden (info_nodo<T>* n) {
29     if (n != 0) {
30         info_nodo<T>* aux;
31         for (aux = n->hijoizq; aux != 0; aux = aux->hermanodcha)
32             ListarPostorden (aux);
33
34         cout << n->et << ' ';
35     }
36 }
37
  
```

```

38 // para la siguiente funcion debemos haber hecho en la cabecera un
39 // #include <queue>
40 template <class T>
41 void ListarNiveles (info_nodo<T>* n) {
42     // imprimimos un nodo y despues guardamos en la cola a sus hijos
43     if (n != 0) {
44         queue<info_nodo<T>*> c;
45         c.push(n);
46         while (!c.empty()) {
47             info_nodo<T>* aux = c.front();
48             c.pop();
49             cout << aux->et << ' ';
50             for (aux=aux->hijoizq;aux!=0;aux=aux->hermanodcha)
51                 c.push(aux);
52         } // cuando la cola quede vacia
53         //se termina el listado por niveles
54     }
55 }
```

Otras funciones interesantes son:

- *size*: devuelve el numero de nodos del árbol.
- *iguales*: devuelve true si dos árboles son iguales o false en caso contrario

```

1 template <class T>
2 int size (info_nodo<T>* n) {
3     if (n == 0)
4         return 0;
5
6     else {
7         int nt = 1; // al menos hay un nodo
8         info_nodo<T>* aux;
9         for (aux=n->hijoizq;aux!=0;aux=aux->hermanodcha)
10            nt += size(aux);
11
12         return nt;
13     }
14 }
15 template <class T>
16 bool iguales (info_nodo<T>* t1, info_nodo<T>* t2) {
17     if (t1==0 && t2==0)
18         return true; // ambos son arboles vacios
19
20     else {
21         if (t1 == 0 || t2 == 0)
22             return false; // uno es vacio y el otro no
23
24         else {
25             if (t1->et != t2->et)
26                 return false;
27         }
28     }
29 }
```

```

28         else {
29             info_nodo<T>* aux1, *aux2;
30             bool igual = true;
31             for (aux1=t1->hijoizq;
32                 aux2=t2->hijoizq; igual && aux1!=0 && aux2!=0;
33                 aux1=aux1->hermanodcha; aux2=aux2->hermanodcha) {
34
35                 igual = iguales(aux1,aux2);
36             }
37             // ahora bien, puede ser que un arbol este contenido en otro,
38             // es decir, su tamaño sea diferente, por lo que comprobamos
39             // si los dos han terminado
40             return igual && aux1==0 && aux2==0;
41         }
42     }
43 }
44 }
```

Ejemplo 6.4.1

Suponiendo que tenemos la clase ArbolGeneral implementar dentro de esta la clase preorden_iterator.

```

1  class preorden_iterator{
2  private:
3     info_nodo<T>* p;
4  public:
5     //Constructor por defecto
6     preorden_iterator(){}
7
8     //Constructor con parametro
9     preorden_iterator(const nodo<T> & n): p(n.p){}
10
11    // se define const porque no modifica p
12    T & operator * () const{
13        return *p;
14    }
15
16    bool operator ==(const preorden_iterator & pre) const {
17        return p==pre.p;
18    }
19    bool operator !=(const preorden_iterator & pre) const{
20        return p!=pre.p;
21    }
22
23    preorden_iterator & operator ++(){
24        if (p==0) // si no apunta a nada
25            return *this;
26        else {
27            if (p->hijoizq!=0)
28                //el siguiente es el hijo mas a la izquierda
```

```

29         p= p->hijoizq;
30
31     else{
32
33         if (p->hermanodcha!=0)
34             // el siguiente sera el hermano a la derecha
35             p = p->hermanodcha;
36         else{
37             // no tiene hijo mas a la izquierda ni
38             //hermano a la derecha
39             //submos por los ascendentes
40             //hasta encontrar un nodo que tenga hermano a la derecha
41             while (p->padre!=0 && p->padre->hermanodcha==0)
42                 p=p->padre;
43
44         if (p->padre!=0)
45             p= p->padre->hermanodcha;//el siguiente es el
46                                         //hermano a la derecha
47         else
48             p=0;//no hay mas nodos
49
50     }
51   }
52 }
53 return *this;
54 }
55
56
57 }
58 // para poder implementar el begin y end
59 friend class ArbolGeneral<T>;
60 //para poder iniciar un preorden con un nodo
61 friend class nodo<T>;
62 };

```

Dentro de la clase ArbolGeneral debemos definir las funciones begin_preorden y end_preorden

```

1  preorden_iterator begin_preorden(){
2      preorden_iterator it;
3      it.p = raiz;
4      return it;
5  }
6  preorden_iterator end_preorden(){
7      preorden_iterator it;
8      it.p.=0;
9      return it;
10 }

```



6.5 Árboles parcialmente ordenados (APO)

Son árboles binarios con la condición de que la etiqueta de cada nodo es menor o igual que la etiqueta de sus hijos y además, es un árbol completo, es decir, tiene todos los niveles completos excepto el último donde los huecos están a la derecha.

Ejemplo 6.5.1

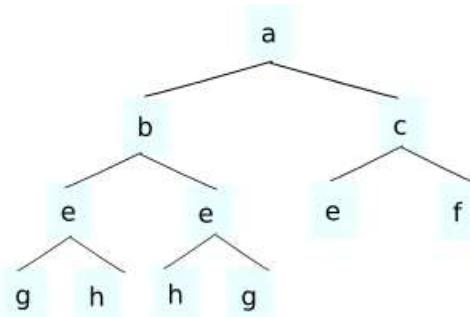


Figura 6.17: Ejemplo de un APO

□

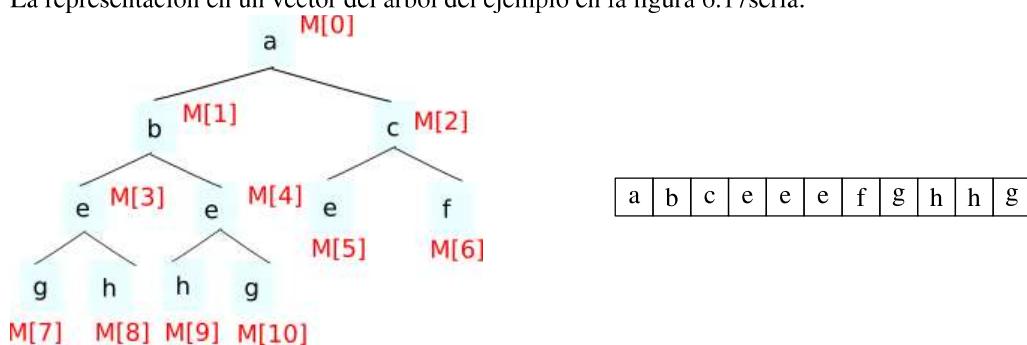
Las funciones típicas de un APO son:

1. Insertar un elemento manteniendo la condición de APO.
2. Borrar el mínimo. El mínimo siempre se encuentra en la raíz del APO.

Su representación óptima es un vector. Aunque lo visualicemos como un árbol binario, para representarlo usaremos un vector con una serie de restricciones. Esta representación junto con las restricciones que vamos a detallar a continuación se le denomina un montón o heap en inglés. El APO se almacena en el vector por niveles. Así supongamos un heap M, entonces debe cumplirse que:

1. M[0] es la raíz
2. M[1] es el hijo a la izquierda
3. M[2] es el hijo a la derecha
4. En general, el nodo k estará en M[k]
5. Sus hijos, si existen, son los elementos M[2k+1] y M[2k+2]
6. Y su padre, teniendo en cuenta que $k = 2n + 1$ y n es la posición que ocupa el padre, $n = \frac{k-1}{2}$ (hacemos división entera). Si fuese $k = 2n + 2$ obtendríamos $n = \frac{k-2}{2}$

La representación en un vector del árbol del ejemplo en la figura 6.17 sería:



6.5.1 Insertar un elemento en un APO

Los pasos para insertar un elemento en un APO son:

1. Insertamos el elemento en el hueco que haya en el último nivel, si no lo hay creamos un nivel nuevo.
 2. Intercambiamos el nodo con el padre hasta que se cumpla la condición de APO.
- La eficiencia es de $O(\log_2(n))$, ya que en cada cambio dejamos sin analizar la mitad del subárbol.

Ejemplo 6.5.2

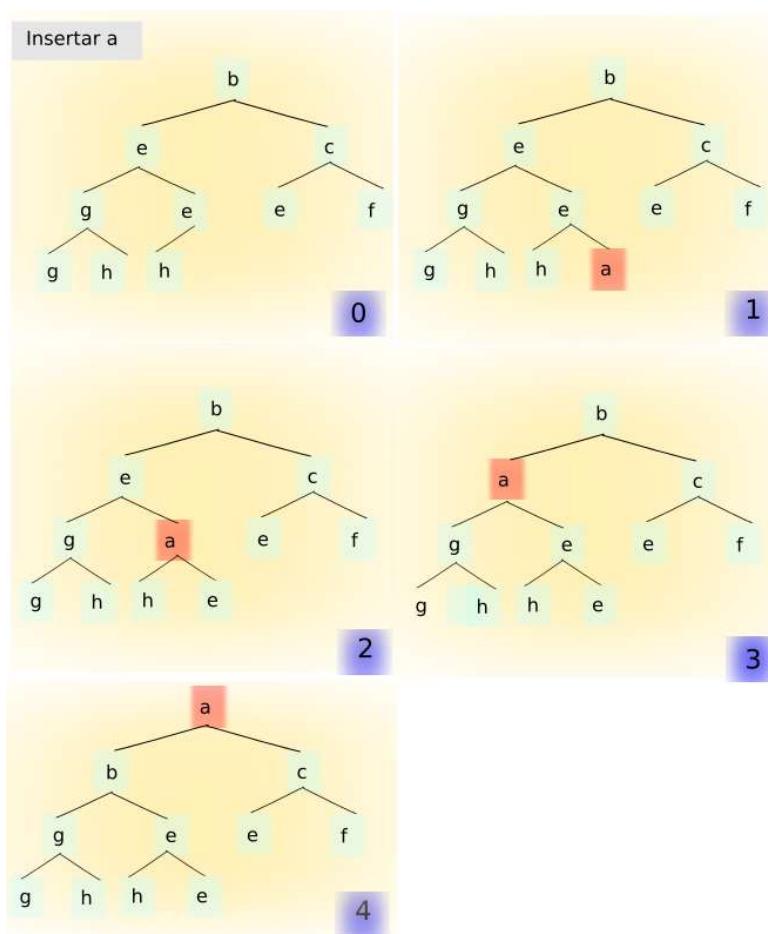


Figura 6.18: Pasos para insertar un elemento en un APO.

En la figura 6.18 se muestra los pasos para insertar un nuevo elemento en un APO. En la viñeta 0 de la figura tenemos el APO original. Como se puede observar todos los nodos cumplen que tiene una etiqueta mayor o igual que la de su padre. Supongamos que queremos insertar el carácter 'a'. Como se puede observar en la viñeta 1 se inserta en el último nivel en el primer hueco que nos encontramos a la izquierda. Pero el árbol que se forma ya no cumple la condición de APO pues el padre tiene etiqueta 'e' que es mayor que 'a'. Por lo tanto procedemos a intercambiar la etiqueta 'e' por la etiqueta 'a', dando lugar a la viñeta 2. De nuevo no se cumple la condición de APO y para lograrlo realizamos un conjunto de

intercambios hasta que 'a' alcanza la raíz. Estos intercambios se puede ver desde la viñeta 1 a la viñeta 4.



Suponiendo que la representación de nuestro APO es:

```

1 template <class T>
2 class APO{
3     private:
4         T *datos;           // vector en el que guardamos los nodos
5         int nelementos;    // tamaño del árbol
6         int reservados;   // espacio de memoria reservado en datos
7         ...

```

El algoritmo de inserción podría ser el siguiente:

```

1 template <class T>
2 void APO<T>::Insertar(const T & x){
3     // si no tenemos espacio
4     if (nelementos == reservados)
5         resize(reservados*2); // ampliamos la memoria
6
7     // insertamos un elemento en la última casilla del vector
8     datos[nelementos] = x;
9     // incrementamos el número de elementos del vector
10    nelementos++;
11
12    // Y ahora empezamos a comparar con el nodo padre hasta que
13    // se cumpla la condición de APO
14    int pos = nelementos-1;
15    while (pos > 0 && datos[pos] < datos[(pos-1)/2]) {
16        swap (datos[pos],datos[(pos-1)/2]);
17        pos = (pos-1)/2;
18    }
19}

```

Podemos también hacer uso de vector de la STL y todo se vuelve más fácil.

```

1 template <class T>
2 class APO{
3     private:
4         vecto<T> datos;           // vector en el que guardamos los nodos
5         ...

```

Con esta representación el algoritmo de inserción sería:

```

1 template <class T>
2 void APO<T>::Insertar(const T & x){
3     // insertamos un elemento en la última casilla del vector
4     datos.push_back(x);
5     // Y ahora empezamos a comparar con el nodo padre hasta que
6     // se cumpla la condición de APO
7     int pos = datos.size()-1;
8     while (pos > 0 && datos[pos] < datos[(pos-1)/2]) {
9         swap (datos[pos],datos[(pos-1)/2]);
10        pos = (pos-1)/2;

```

```

11    }
12  }

```

6.5.2 Borrar el mínimo (la raíz)

Cuando borramos un elemento en un APO siempre vamos a borrar la raíz, que es donde está situado el elemento mínimo de entre los que se encuentran en el APO. Los pasos para borrar la raíz de un APO son:

1. El elemento que se pone en la raíz es el que está en el último nivel más a la derecha.
2. El más pequeño de los dos hijos de la raíz se intercambia con ésta, así hasta que se obtenga la condición de APO.

Ejemplo 6.5.3

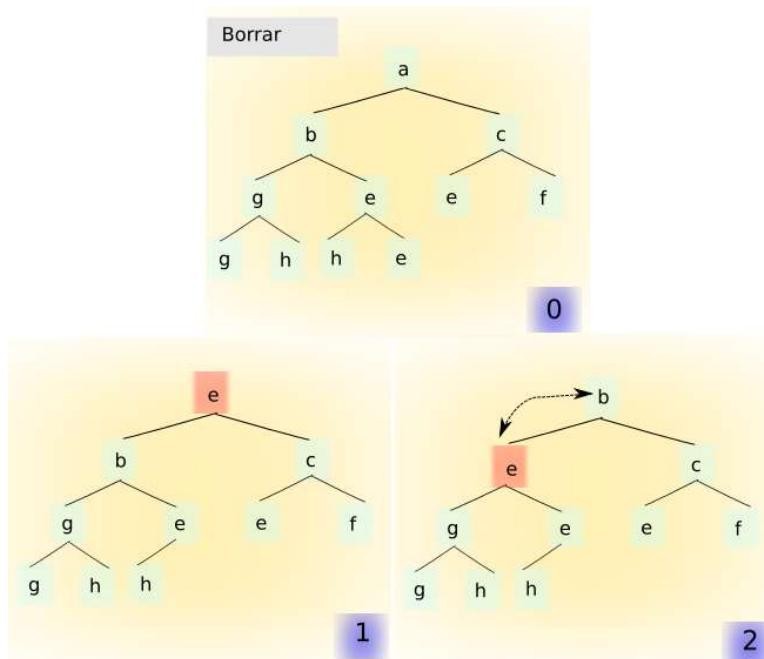


Figura 6.19: Pasos para borrar en un APO.

En la figura 6.19 se muestra los pasos para borrar el elemento mínimo (que se sitúa en la raíz) en un APO. Al borrar la raíz, esta se debe sustituir por el elemento que está situado en el último nivel mas a la derecha (en la estructura heap el último elemento del vector). Así en el ejemplo que se muestra en la figura 6.19 al borrar 'a' se sustituye por 'e'. Este nuevo árbol que se muestra en la viñeta 1 no cumple la condición de APO. Por eso lo que se hace es intercambiar 'e' por el menor de sus hijos que es 'b'. El árbol resultante se muestra en la viñeta 2. Además el árbol de la viñeta 2 ya es un APO, por lo tanto el procedimiento de borrado aquí se detiene. \square

La eficiencia de borrar es $O(\log_2(n))$, ya que cada vez efectuamos un intercambio estamos eliminando en el análisis la mitad de nodos.

Los pasos fundamentales de este algoritmo en C++ sería:

```

1 // colocamos en la raiz el ultimo nodo del arbol

```

```

2   datos[0] = datos[datos.size()-1];
3   datos.pop_back(); // y lo eliminamos
4   int ultimo = datos.size()-1;
5   int pos = 0;
6   bool acabar = false;
7
8   while (pos <= (ultimo-1)/2 && !acabar) {
9       int pos_min;
10      // este primer if-else sirve para saber
11      // cual de los dos hijos es menor y asi
12      // saber cual intercambiar con el nodo superior
13      if ((pos*2)+1 == ultimo)
14          // si es el hijo a la izquierda y el unico hijo
15          pos_min = ultimo;
16
17      else {
18          if (datos[2*pos+1] < datos[2*pos+2]) // si tenemos mas hijos
19              // si el hijo a la izquierda es menor
20              pos_min = (2*pos)+1;
21
22          else // el hijo a la derecha es menor
23              pos_min = (2*pos)+2;
24      }
25
26      // una vez calculado el hijo menor, si es menor que su padre los
27      // intercambiamos
28      if (datos[pos] > datos[pos_min]) {
29          swap(datos[pos], datos[pos_min]);
30          pos = pos_min; // actualizamos el valor de pos min para la
31                      // siguiente iteracion
32      }
33
34      else // ya se cumple la condicion de APO
35          acabar = true;
36  }

```

6.6 Árboles binarios de búsqueda (ABB)

ABB: Es un árbol binario con las etiquetas ordenadas de forma que el elemento situado en un nodo es mayor que todos los elementos que se encuentran en el subárbol izquierdo y menor que los que se encuentran en el subárbol derecho. En general, si nos fijamos en un nodo, su hijo izquierdo es menor y el derecho mayor. Aplicándolo recursivamente llegamos a que todas las etiquetas del subárbol izquierdo son menores y las etiquetas del subárbol derecho son mayores.

Ejemplo 6.6.1

El árbol binario que se muestra es un árbol binario de búsqueda. Como se puede observar por ejemplo el hijo a la izquierda de 8 es 4 que es menor que 8, y además todo el subárbol izquierdo tiene valores menores que 8. Por otro lado el hijo a la derecha es 12 y además todo el subárbol derecho tiene etiquetas

mayores que 8.

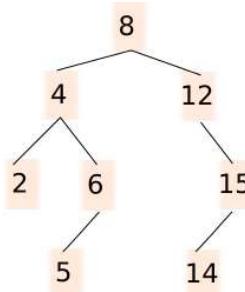


Figura 6.20: Ejemplo de ABB

Como detalle a analizar en un ABB el menor de los elementos será o bien el nodo mas a la izquierda que sea hoja o que como mucho tenga un hijo a la derecha. De la misma forma el elemento mayor almacenado en un ABB será aquel que se situe más a la derecha teniendo como mucho un hijo a la izquierda o siendo hoja. En el ABB que muestra en la figura 6.20 el mínimo es 2 (el elemento que se encuentra más a la izquierda y que en este caso es hoja). Y el mayor elemento es 15 que aunque no es una hoja solamente tiene un hijo a la izquierda.

□

Ejemplo 6.6.2

Si nos dan las siguientes etiquetas:

$$\{10, 5, 14, 7, 12, 3, 8, 6\}$$

¿cómo podemos obtener el ABB? Los pasos a seguir son los siguientes:

1. El primer elemento del conjunto de etiquetas, 10, es la raíz. Se construye un árbol con un solo nodo como se puede ver en la figura 6.21 viñeta 1.
2. A continuación nos dan la etiqueta 5, ya que es menor que 10 esta se coloca como hijo izquierdo de 10, el hijo derecho será 14 (ver viñeta 2).
3. Para insertar 7 en primer lugar se compara con 10 que es menor, por lo tanto redirijimos nuestro proceso de inserción por el subárbol izquierdo. Ahora se compara con 5 al ser mayor y 5 no tener hijo a la derecha, el 7 pasa a ser el hijo a la derecha de 5.
4. Tras hacer la inserción de 12 se obtiene el ABB que se muestra en la viñeta 3.
5. Así, para insertar el 6 en el árbol debemos ir nodo por nodo viendo si tirar para la derecha o la izquierda. Los pasos serían:
 - a) $6 < 10 \rightarrow$ tiramos a la izquierda
 - b) $6 > 5 \rightarrow$ tiramos al subárbol derecho
 - c) $6 < 7 \rightarrow$ como no tiene hijo izquierdo, ponemos a 6 como hijo izquierdo.

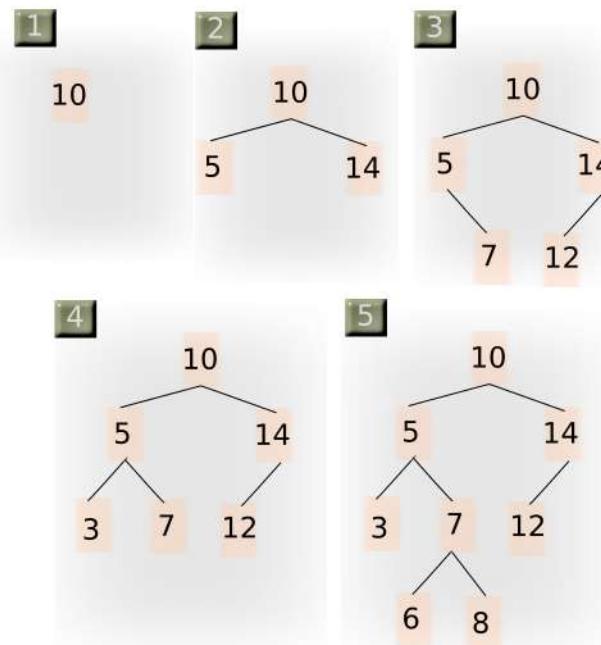


Figura 6.21: Ejemplo de construcción ABB

El ABB resultante se puede ver en la figura 6.21 viñeta 5. □

En promedio, es decir en un conjunto de búsquedas, la mayoría va a tener una eficiencia de $\log_2(n)$. Esto es así ya que cada vez que realizamos una comparación, en promedio, no tendremos que comparar con la mitad de los restantes valores. Pero existe un caso donde la búsqueda de un elemento tiene eficiencia $O(n)$. Esta es la situación que ocurre cuando las claves se disponen en una sola rama.

El recorrido inorden ordena las etiquetas de menor a mayor. Por ejemplo, el recorrido inorden el árbol calculado en el ejemplo 6.6.2 sería:

3 5 6 7 8 10 12 14

El tipo set de la STL está implementado con un árbol binario de búsqueda.

6.6.1 Implementación de un ABB

En cuanto a la implementación de un ABB en C++ podemos tomar como implementación base la vista para Arbol Binario en la sección 6.3.2. A diferencia de esta tenemos que añadir una función para buscar, insertar y borrar un elemento en el ABB. Por otro lado el único iterador que nos interesa es el inorden para dado el ABB obtener una ordenación de las claves que almacena. Por lo tanto se puede mantener simplemente la clase nodo del árbol binario (ver sección 6.3.2) y sobrecargar en este el operador ++ para pasar al siguiente nodo en inorden y respecto al operador -- igual.

La representación de un ABB en C++ sería:

```

1 template <class T>
2 struct info_nodo {
3     T et;
4     info_nodo<T> * padre, * hijoizq, * hijoder;
```

```

5  };
6  //funcion que busca en un ABB una etiqueta, si no esta devuelve 0
7  template <class T>
8  info_nodo<T> * Buscar (info_nodo<T> * n, T x) {
9      if (n != 0) {
10          if (n->et == x)
11              return n;
12
13          else {
14              if (n->et > x)
15                  return Buscar (n->hijoizq, x);
16
17              else
18                  return Buscar (n->hijoder, x);
19          }
20      }
21
22      else // la etiqueta no esta en el arbol
23          return 0;
24 }
25 // misma funcion pero de forma iterativa
26 info_nodo<T> * Buscar (info_nodo<T> * n, T x) {
27     if (n == 0)
28         return 0;
29
30     else {
31         info_nodo<T> * p = n; // variable para recorrer el arbol
32         while (p != 0) {
33             if (p->et == x)
34                 return p;
35
36             else {
37                 if (p->et > x)
38                     p = p->hijoizq;
39
40                 else
41                     p = p->hijoder;
42             }
43         }
44
45         return 0; // salimos del while sin haberlo encontrado
46     }
47 }
48 /* La version iterativa es mas rápida que la recursiva
49 ya que la recursiva guarda contextos de las llamadas
50 y por tanto es mucho mas costosa */

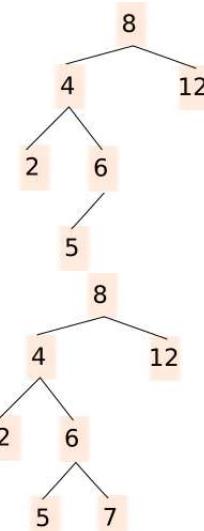
```

El algoritmo de inserción consiste en buscar la posición donde poner el elemento que queremos insertar e insertarlo ahí.

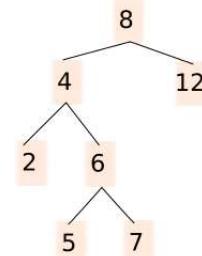
Ejemplo 6.6.3

Insertar $x = 7$ en el siguiente árbol sigue el siguiente razonamiento lógico:

1. $7 < 8 \rightarrow$ voy al subárbol izquierdo
2. $7 > 4 \rightarrow$ voy al subárbol derecho
3. $7 > 6 \rightarrow$ voy al subárbol derecho que está vacío, como está vacío, inserto el 7 aquí.



El resultado sería



□

La implementación en C++ sería:

```

1 // Devuelve true o false si se ha podido insertar x o no
2 template <class T>
3 bool Insertar (info_nodo<T> * & n, T x) {
4     bool res = false;
5     if (n == 0) {
6         n = new info_nodo<T> (x);
7         return true;
8     }
9
10    else {
11        if (n->et < x) {
12            res = Insertar (n->hijoder,x);
13            if (res)
14                n->hijoder->padre = n;
15
16            return res;
17        }
18
19        else {
20            if (n->et > x) {
21                res = Insertar(n->hijoizq,x);
22                if (res)
23                    n->hijoizq->padre = n;
24
25                return res;
26            }
27            else // la etiqueta es x, no se puede insertar
28                return false;
29        }
  
```

```

30      }
31  }
```

Para el algoritmo de borrado, tenemos tres posibilidades:

Primera posibilidad : el info_nodo de x es una hoja. En cuyo caso, simplemente eliminamos dicho nodo. El código correspondiente sería:

```

info_nodo<T>* aux = n;
// suponiendo que n apunta a x
if (aux->padre != 0) {
    if (aux->padre->hijoder == n)
        aux->padre->hijoder = 0;

    else
        aux->padre->hijoizq = 0;
}

delete aux;
```

Segunda posibilidad : el nodo no es hoja. En este caso, se subdivide en otros tres casos:

1. **Que sólo tenga hijo a la derecha**, en cuyo caso se pondría en el lugar de n, su hijo a la derecha:

```

info_nodo<T> * padre = n->padre;
if (padre != 0) {
    if (padre->hijoder == n) {
        padre->hijoder = n->hijoder;
        padre->hijoder->padre = padre;
    }

    else {
        if (padre->hijoizq == n) {
            padre->hijoizq = n->hijoder;
            padre->hijoizq->padre = padre;
        }
    }
}
```

```

info_nodo<T> * aux = n;
n = n->hijoder;
delete aux;
```

2. **Que sólo tenga hijo a la izquierda**, en cuyo caso se pondría en el lugar de n, su hijo a la izquierda:

```

info_nodo<T> * padre = n->padre;
if (padre != 0) {
    if (padre->hijoizq == n) {
        padre->hijoizq = n->hijoder;
        padre->hijoizq = padre->padre;
    }

    else {
        if (padre->hijoder == n) {
```

```

        padre->hijoder = n->hijoizq;
        padre->hijoder->padre = padre;
    }
}
}

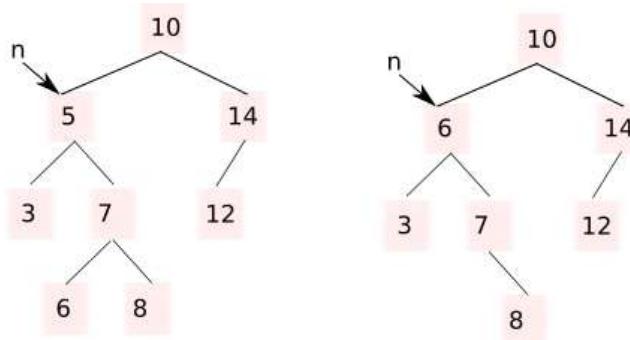
info_nodo<T> * aux = n;
n = n->hijoizq;
delete aux;

```

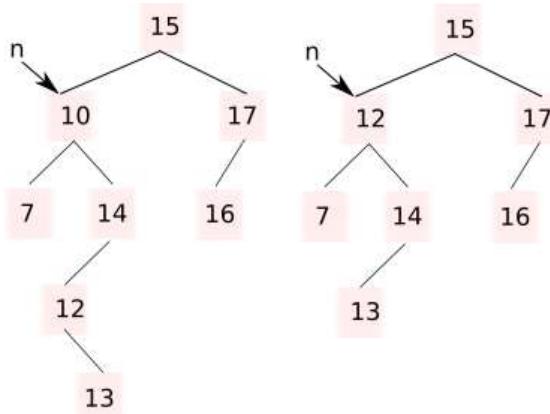
3. **Que tenga dos hijos**, en cuyo caso tenemos que sustituir el nodo por su “siguiente”, es decir, el siguiente nodo con más valor. Para obtener dicho “siguiente” debemos seguir los siguientes pasos:

- Nos movemos hacia el hijo a la derecha de n .
- Después, nos movemos hacia el hijo a la izquierda del hijo a la derecha de n
- Vamos moviéndonos por todos los hijos a la izquierda hasta llegar a un nodo que no tenga más hijos a la izquierda. Puede ser una hoja (no tener hijos) o tener hijo a la derecha.
- Cuando llegamos a dicho nodo, cambiamos el valor de la etiqueta de n por el del nodo y borramos el nodo siguiendo la posibilidad 1 (no tener hijos) o la posibilidad 2, caso 1 (tener hijo derecho).

Por ejemplo, queremos borrar el nodo con etiqueta 5. El árbol de la izquierda representa el antes y el de la derecha el después:



Otro ejemplo, en el que ahora el nodo con el que vamos a hacer el cambio no es una hoja, sino que sólo tiene un hijo a la derecha. El resultado final sería sustituir n por el nodo sin hijo izquierdo que hemos encontrado y al borrar éste nodo, dejamos como hijo izquierdo de su padre, a su hijo derecho:



Así, el algoritmo de borrar final, y las funciones auxiliares que necesita, quedarían:

```

1 // funcion que enlaza un hijo con su padre
2 template <class T>
3 void PutHijo_Padre(info_nodo *n, info_nodo *nuevo){
4     if (n->padre!=0){
5         if (nuevo!=0)// no tiene padre
6             nuevo->padre = n->padre;
7         // el padre de n tendra como hijo a n
8         if (n->padre->hder==n)
9             // su hijo a la derecha
10            n->padre->hder = nuevo;
11
12     else // el padre de n tendra como hijo
13         // a su hijo a la izquierda
14         n->padre->hizq=nuevo;
15
16 }
17 }
18
19 // En esta funcion es donde encapsulamos cada uno de los casos que
20 // hemos visto por separado y donde se borra el nodo.
21 template <class T>
22 void EliminarRaiz(info_nodo *&n){
23     if (n->hizq==0 && n->hder==0){ // Posibilidad 1: n no tiene hijos
24         PutHijo_Padre(n,0); // establecemos su parent a 0
25         delete n; n=0; // lo borramos
26
27 }
28
29 else if (n->hizq==0){// Posibilidad 2, CASO 1:
30     //n tiene hijo derecho
31     info_nodo *aux=n;
32     PutHijo_Padre(n,n->hder); // hijo a la derecha de n
33     if (n->padre==0)// si n es la raiz
34         n= n->hder; // La raiz del arbol

```

```

35                     //ahora es su hijo derecho
36         delete aux;aux=0;// Eliminamos n
37
38     }
39     else if (n->hder==0){// Posibilidad 2, CASO 2:
40         //n tiene hijo izquierdo
41         info_nodo *aux=n;
42         PutHijo_Padre(n,n->hizq);
43         if (n->padre==0)
44             n = n->hizq;
45         delete aux; aux=0;
46
47     }
48     else{ // Posibilidad 2, CASO 3: n tiene dos hijos
49         // Buscamos el siguiente:
50         info_nodo *aux=n->hder;// Nos movemos al hijo a la derecha
51         while (aux->hizq!=0)// Avanzamos hasta que llegamos a un nodo
52             aux=aux->hizq;// que no tiene hijo a la izquierda
53         // OJO: no tiene por que ser una hoja
54         n->et=aux->et;// y cambiamos la etiqueta de n por la del nodo
55         // y por ultimo, borramos este nodo.
56         Borrar(n->hder,aux->et);
57     }
58
59 }
60
61 // Funcion que llama a EliminarRaiz para borrar alguna componente del arbol
62 template <class T>
63 void Borrar(info_nodo * &n,const T &e){
64     if (n!=0){
65         if (n->et==e) // Si encontramos la raiz del subarbol que
66             EliminarRaiz(n); // queremos borrar, lo borramos
67         else if (n->et<e) // si el nodo que queremos borrar es mayor
68             Borrar(n->hder,e); // o menor que la raiz del subarbol actual,
69         else // nos movemos a la izquierda o a la derecha
70             Borrar(n->hizq,e); // recursivamente hasta encontrarlo.
71     }
72 }
```

Ejemplo 6.6.4

Construir un programa que dado un conjunto de claves de tipo char las muestre, por la salida estándar, de forma ordenada.

```

1 #include <ABB.h>
2 #include <iostream>
3 typename <class T>
4 void ImprimirAbb(ABB<T> &A){
5     typename ABB<T>::nodo n;
6     //el operator ++ de nodo hace avanzar a nodo en inorden
```

```

7   for (n=A.begin(); n!=A.end(); ++n){
8     std::cout<<*n<<std::endl;
9   }
10 }
11 typename <class T>
12 void LeerDatos(ABB<T> & A){
13   T dato;
14   while (std::cin>>dato){
15     A.Insertar(dato);
16   }
17 }
18
19 int main(){
20   ABB<char> A;
21   LeerDatos(A);
22   ImprimirAbb(A);
23 }
```

Ejemplo 6.6.5

Almacenar la información de un conjunto de alumnos (dni, nombre, apellidos, email) para que se encuentre ordenada por dni.

```

1 #include <ABB.h>
2 #include <iostream>
3 #include <string>
4 struct alumno{
5   string dni;
6   string nombre;
7   string apellidos;
8   string email;
9 };
10 bool operator<(const alumno &a1, const alumno &a2) const{
11   return a1.dni<a2.dni;
12 }
13 //suponemos que los campos de cada alumno se encuentra
14 //en una linea
15 std::istream & operator>>(std::istream &is,alumno & a){
16   is.getline(a.dni);
17   is.getline(a.nombre);
18   is.getline(a.apellidos);
19   is.getline(a.email);
20   return is;
21 }
22 typename <class T>
23 void LeerDatos(ABB<T> & A){
24   T dato;
25   while (std::cin>>dato){
26     A.Insertar(dato);
27   }
```

```

28 }
29
30 int main(){
31     ABB<alumno>A;
32     LeerDatos(A);
33     //hacemos algo con los datos
34     ...
35 }
```

□

6.7 Árboles binarios de búsqueda AVL (Adelson-Velskii y Landis)

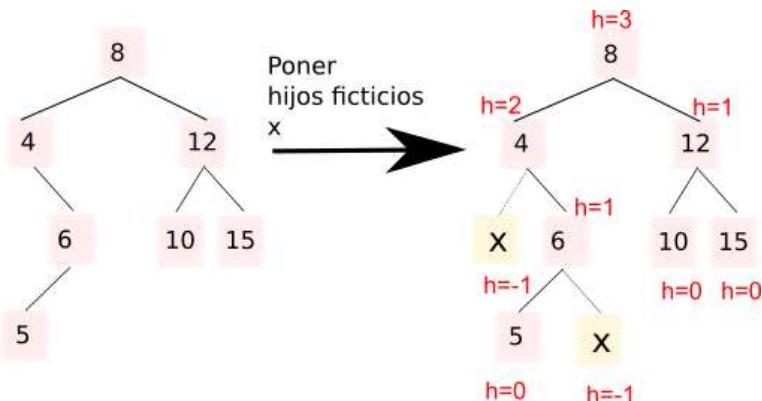
Son árboles de búsqueda equilibrados. Si un ABB está muy desequilibrado, los tiempos de búsqueda no son $\log_2(n)$, en el peor de los casos podría ser $O(n)$. Lo ideal sería que ambas partes tuvieran más o menos el mismo número de nodos para que en cada iteración, descartar la mitad de nodos del árbol y por tanto, de verdad tener un tiempo de búsqueda $\log_2(n)$.

Se dice que un ABB es AVL si la diferencia de altura de los subárboles izquierdo (T_i) y derecho (T_d) que cuelgan de un nodo es como mucho 1. Es decir, si T_i tiene altura h , T_d puede tener como máximo altura $h+1$ y viceversa.

Si no recuerdas lo que era la altura, repasa la sección 6.2.1.

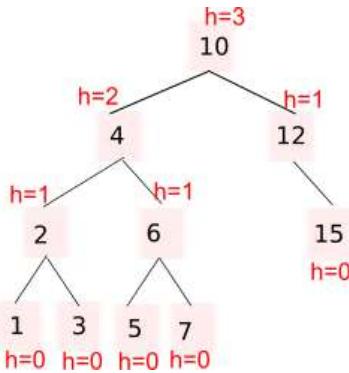
Ejemplo 6.7.1

Dado el árbol a la izquierda vamos a obtener su altura. Pero antes vamos a transformarlo en el árbol de la derecha. Este árbol se obtiene añadiendo al árbol de la izquierda el hijo que le falta cuando un nodo tiene un sólo hijo. A este hijo ficticio le hemos puesto la etiqueta x .



1. Los nodos que no existen (x) tienen altura $h = -1$
2. Las hoja tienen altura $h = 0$
3. Por ejemplo el nodo 6 tiene altura 1, ya que sería la altura máxima de sus hijos más 1.
4. Tenemos un desequilibrio en el 4, ya que sus hijos tienen $h(x) = -1$ y $h(2) = 1$, la altura de ambos difiere en más de 1, por lo que no es AVL.

□

Ejemplo 6.7.2

Este árbol si está equilibrado, aunque no tengamos el mismo número de nodos en T_l y T_d ya que se cumple la definición de AVL. \square

La representación en C++ sería:

```

1 template <class T>
2 struct info_nodo_AVL {
3     T et;
4     info_nodo_AVL<T> * hijoizq, * hijoder, * padre;
5     int altura;
6 };
7
8 // El proceso de búsqueda es identico al ABB normal.
  
```

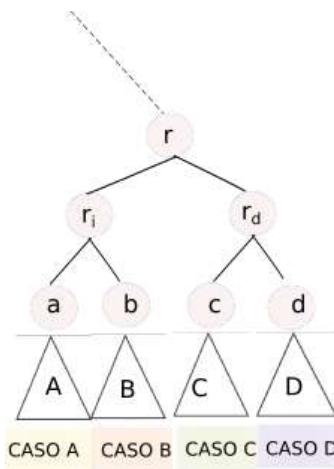
Cada nodo del árbol almacena su altura. Al realizar una inserción en el árbol la altura del nodo puede verse afectada.

6.7.1 Inserción en un AVL

En el proceso de inserción podemos desequilibrar el árbol, por tanto, debemos volver a hacer que esté equilibrado. Por tanto los pasos a seguir para insertar un elemento en un AVL serían:

1. Buscar dónde insertar el nuevo elemento
2. Insertarlo
3. Equilibrar el árbol

Como se puede ver en la siguiente figura el desequilibrio ocurre en el nodo r. Pero puede ocurrir porque se haya realizado la nueva inserción en el subárbol A, B, C o D.

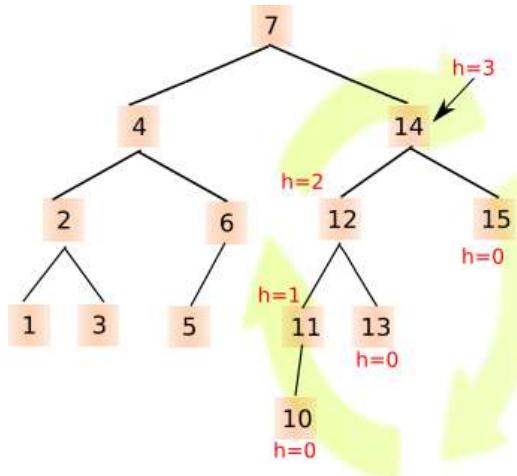


El procedimiento para volver a equilibrar el árbol depende de dónde se haya hecho la nueva inserción. Para lograr el equilibrio se aplicará rotaciones simples (ocurren cuando la nueva inserción se ha hecho en el subárbol A o D), o rotaciones dobles que ocurren cuando se hace la nueva inserción en los subárboles B y C. Veamos en mayor detalle estos casos.

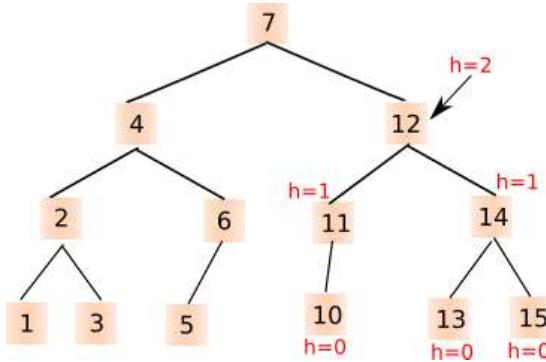
6.7.2 Rotaciones simples

CASO A : El desequilibrio se produce al insertar un nuevo elemento en la parte más a la izquierda del árbol (subárbol A). Para equilibrarlo de nuevo, se hace una *rotación simple a la derecha*.

Ejemplo 6.7.3



Supongamos que el árbol estaba equilibrado y se inserta la clave 10 dando lugar al árbol que se ve en la imagen anterior. En este caso, el desequilibrio está en 14, pues la altura de 12 es $h = 2$ y la de 15, $h = 0$. Al hacer la rotación simple a la derecha, el árbol queda ya equilibrado:



□

El algoritmo asociado a la rotación simple a derecha sería.:

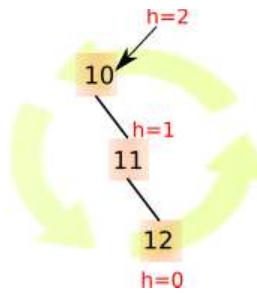
```

1  template <class T>
2  void SimpleDerecha (info_nodo_AVL<T> * & n) { // n = 14 en el ejemplo
3      info_nodo_AVL<T> * aux = n->hijoizq; // 12 en el ejemplo
4      info_nodo_AVL<T> * padre = n->padre; // 7
5      // a 14 le ponemos como hijo izquierdo 13
6      n->hijoizq = aux->hijoder;
7      if (n->hijoizq != 0)
8          // el padre de 13 pasa a ser 14
9          n->hijoizq->padre = n;
10     n->padre = aux; // el padre de 14 pasa a ser 12
11     aux->padre = padre; // el padre de 12 es 7
12     aux->hder = n; // 12 tiene como hijo derecho a 14
13     n = aux;
14     ActualizarAltura(n->hder);
15 }
16
17 // Esta función Actualiza el campo n->altura
18 template <class T>
19 void ActualizarAltura (info_nodo_AVL<T> * & n) {
20     if (n != 0) {
21         n->altura = std::max(Altura(n->hijoizq), Altura(n->hijoder))+1;
22         // La función Altura devuelve n->altura si es
23         // distinta de 0 y -1 si es 0
24         ActualizarAltura(n->padre);
25     }
26 }
```

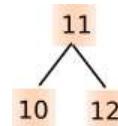
Como se puede analizar en la función ActualizarAltura se debe modificar si es necesario la altura de toda la rama hasta llegar al nodo raíz.

CASO D : El nodo que produce el desequilibrio se inserta en el subárbol D, para volver a equilibrarlo se hace una *rotación simple a la izquierda*.

Ejemplo 6.7.4

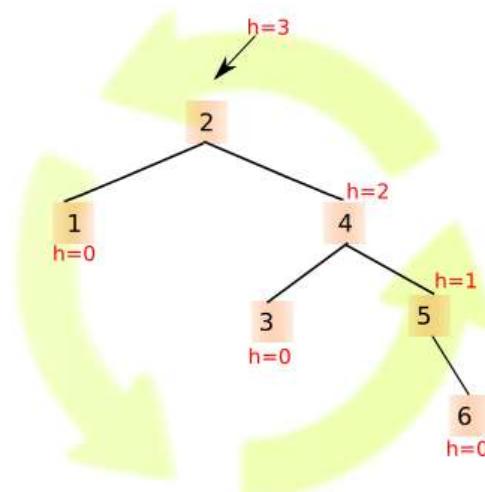


En este caso, el desequilibrio se encuentra en 10. Hay que recordar que si falta el hijo a la izquierda de 10, creamos un nodo ficticio que tiene altura -1 (hermano de 11) por lo tanto la diferencia es 2 en altura. El árbol, una vez equilibrado haciendo rotación simple a la izquierda quedaría así:

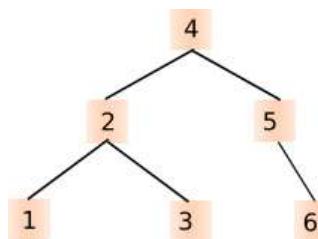


□

Ejemplo 6.7.5



En este caso el desequilibrio estaría en el 2, ya que su hijo derecho tiene altura $h = 2$ y el izquierdo, $h = 0$. Se resolvería haciendo una rotación simple a la izquierda:



El 3 se pone como hijo a la derecha de 2 para mantener la condición de AVL.
La implementación en C++ sería:

□

```

1 template <class T>
2 void SimpleIzquierda (info_nodo_AVL<T> * & n) { // n = 2
3     info_nodo_AVL<T> * aux = n->hijoder;           // 4 en el ejemplo
4     info_nodo_AVL<T> * padre = n->padre;           // nulo
5     n->hder = aux->hijoizq; // a 2 se le pone a 3 como hijo derecho
6     if (n->hder!=0)
7         n->hder->padre = n;// el padre de 3 pasa a ser 2
8
9     n->padre = aux;      // el padre de 2 es 4
10    aux->padre = padre; // el padre de 4 es nulo, porque es la raiz
11    aux->hijoizq = n;  // el hijo izquierdo de 4 es 2
12    n = aux;
13    ActualizarAltura(n->hijoizq);
14 }

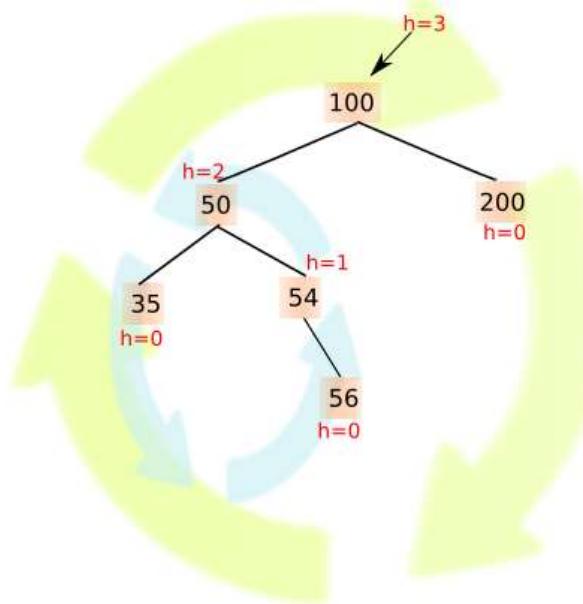
```

6.7.3 Rotaciones dobles

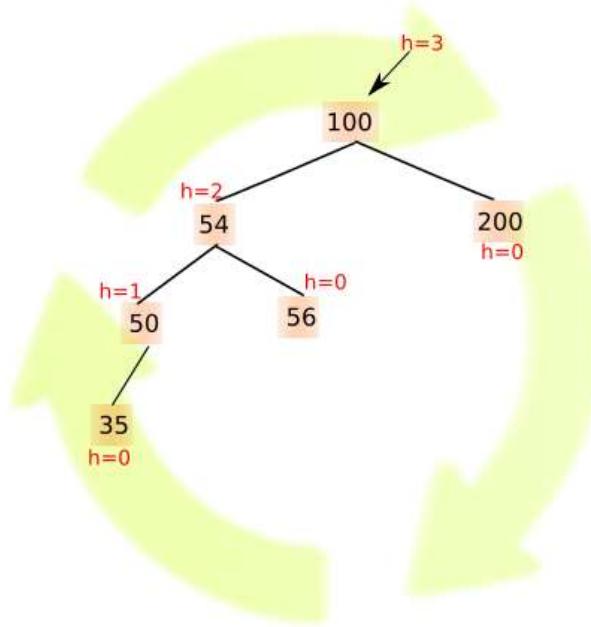
CASO B : el desequilibrio se produce al insertar un nodo en el subárbol B. Para equilibrar el árbol debemos seguir dos pasos:

1. Hacer una rotación simple a la izquierda sobre el hijo izquierdo del nodo donde se produzca el desequilibrio
2. Hacer una rotación simple a la derecha sobre el nodo donde surge el desequilibrio.

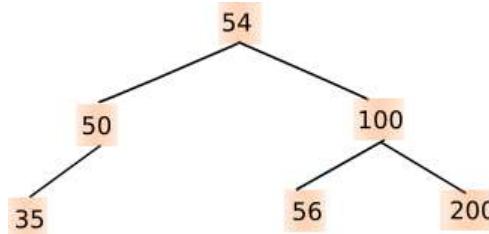
Ejemplo 6.7.6



El desequilibrio se da en el nodo con etiqueta 100, en este caso, para equilibrarlo debemos dar dos pasos. En primer lugar debemos hacer una rotación simple a la izquierda en el nodo de etiqueta 50:



Pero, el árbol aún no está equilibrado, falta el último paso que sería hacer una rotación simple a la derecha sobre 100:



□

La rotación doble consistiría, por tanto, en llamar a las funciones de rotación simples pasando como argumento los nodos correspondientes:

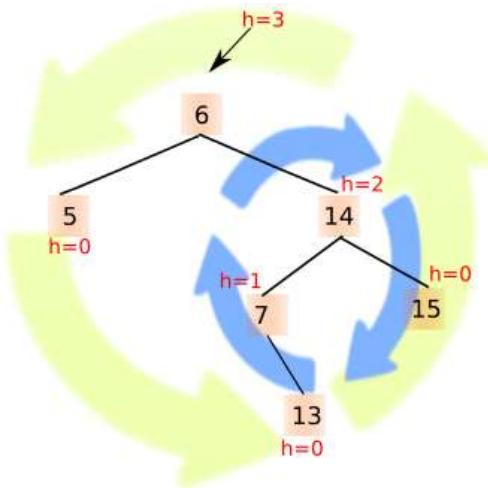
```

1 template <class T>
2 void Doble_IzquierdaDerecha (info_nodo_AVL<T> * & n) {
3     SimpleIzquierda (n->hijoizq);
4     SimpleDerecha(n);
5 }
  
```

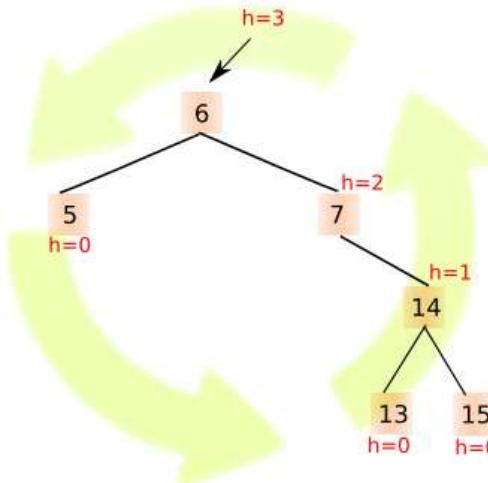
CASO C: es equivalente pero hay que hacerlo al contrario, es decir, los pasos a seguir serían:

1. Hacer una rotación simple a la derecha sobre el hijo derecho del nodo que tenga desequilibrio
2. Y hacer una rotación simple a la izquierda sobre el dicho nodo.

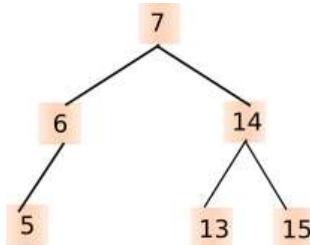
Ejemplo 6.7.7



El desequilibrio está en el nodo de etiqueta 6, para equilibrar el árbol debemos hacerlo en dos pasos. En primer lugar, haremos una rotación simple a la derecha sobre 14:



Y por último, hacemos una rotación simple a la izquierda sobre el nodo desequilibrado, 6:



□

El código implementado en C++ sería:

```

1 template <class T>
2 void Doble_DerechaIzquierda (info_nodo_AVL<T> * & n) {
3     SimpleDerecha(n->hijoder);

```

```

4           SimpleIzquierda (n);
5   }

```

Ejemplo 6.7.8

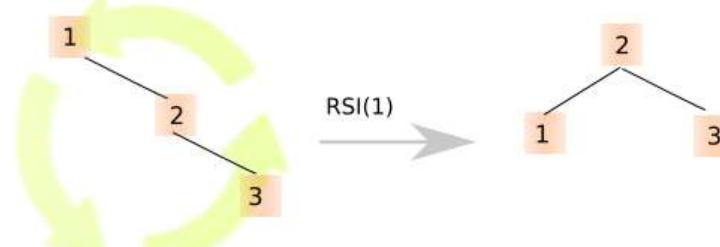
Dada una lista de números crear un árbol binario de búsqueda AVL:

$\{1, 2, 3, 4, 5, 6, 7, 15, 14, 13, 12, 11, 10, 9\}$

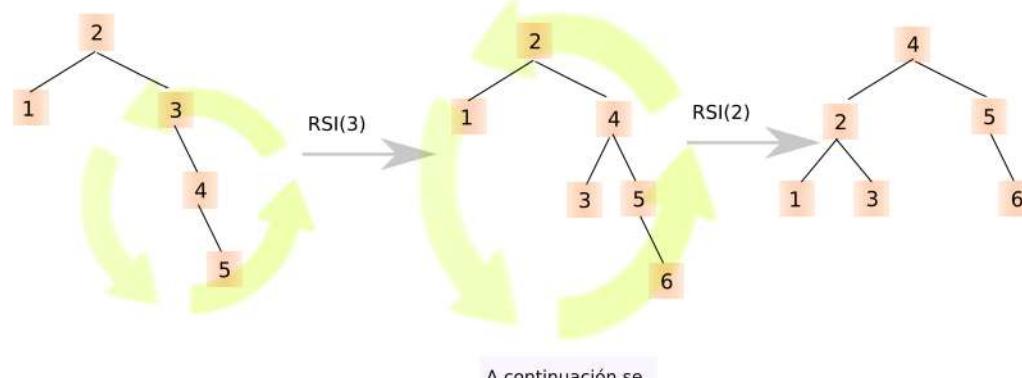
Los pasos para resolver este ejercicio son, elemento a elemento:

1. Insertar el elemento que corresponda.
2. Comprobar que el árbol está equilibrado.
 - a) Si lo está, seguimos insertando elementos donde corresponda
 - b) Si no lo está, lo equilibraremos antes de seguir insertando.

Entonces, empezaríamos insertando el 1 como raíz del árbol y al ser el único nodo no quedaría desequilibrado. Después, insertaríamos el 2 como hijo a la derecha y seguiría estando equilibrado, así que por último, insertamos el 3 como hijo a la derecha de 2 y como resultado, tenemos un desequilibrio **caso D**, por lo que lo equilibraremos haciendo una rotación simple a la izquierda sobre 1:

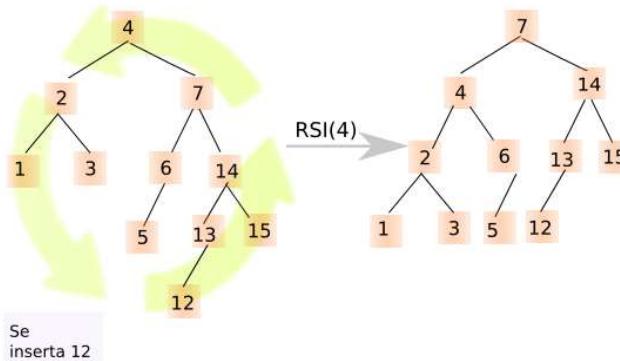
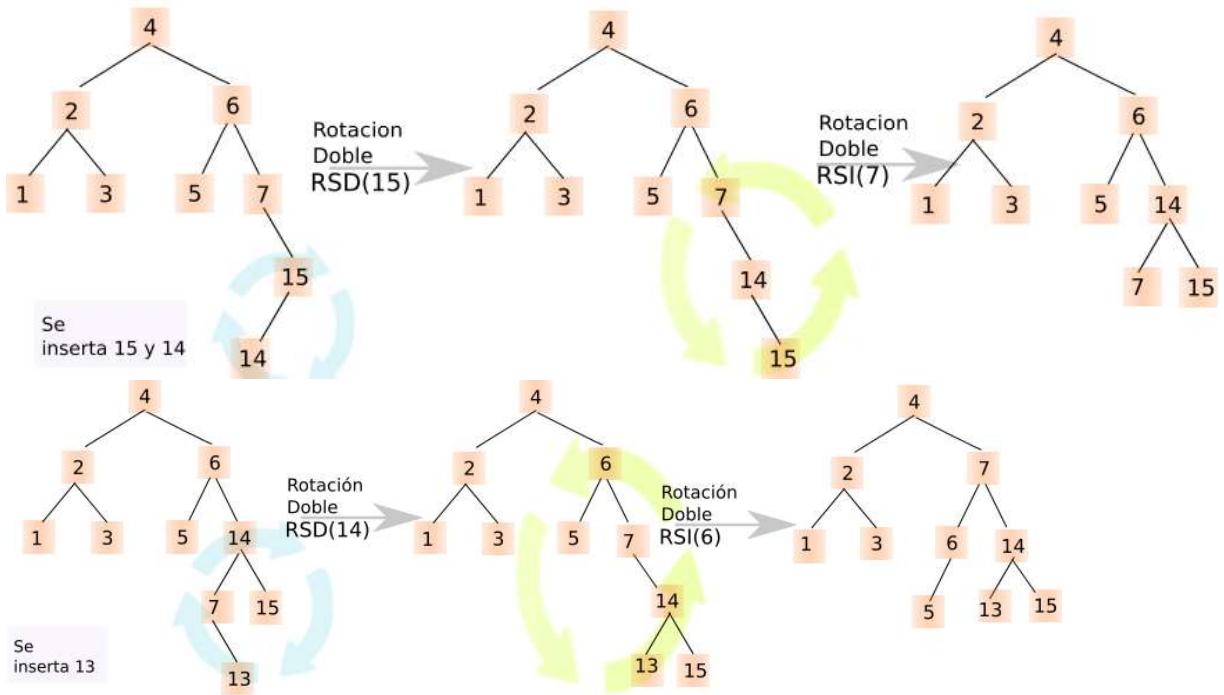
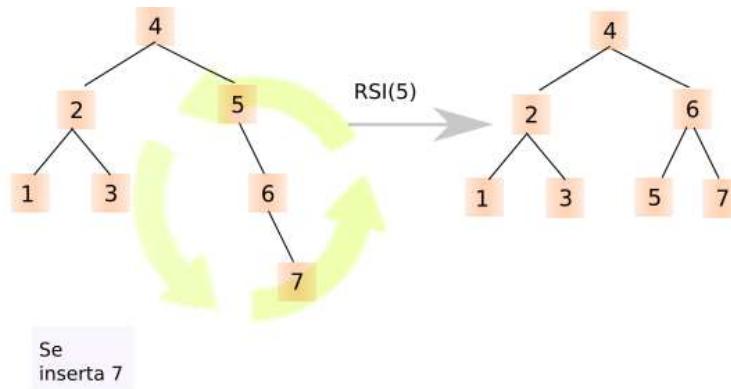


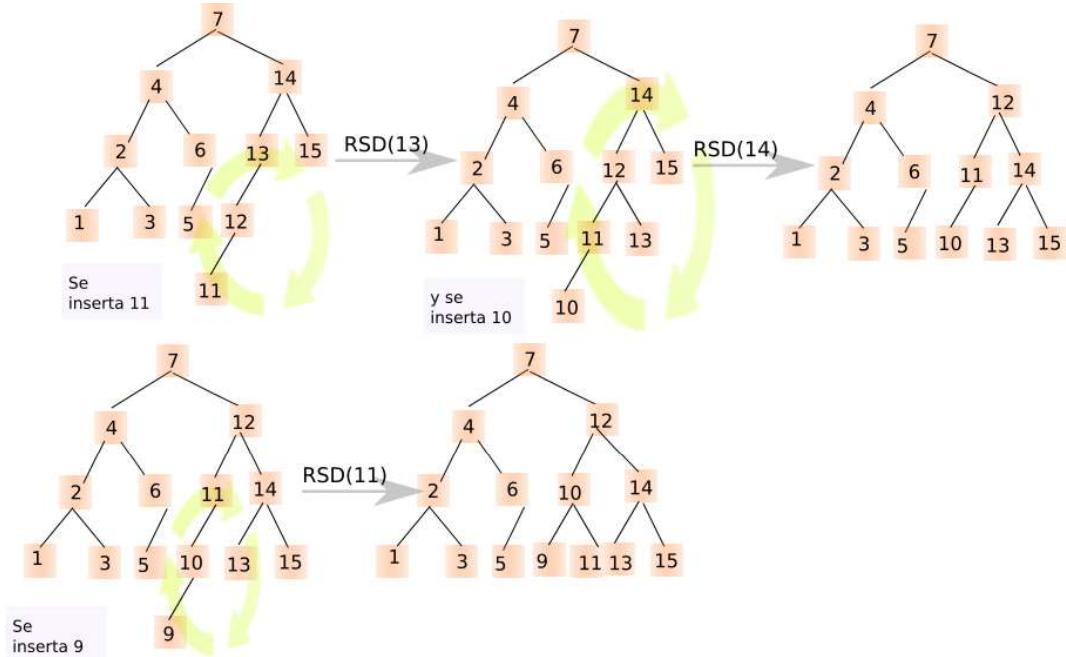
A continuación se hace la inserción de las claves 4, 5, y 6. Cuando se hace la inserción de la clave 5 se produce un desequilibrio en 3. A continuación se introduce la clave 6 y se produce un desequilibrio en 2.



A continuación se
inserta 6

Ahora se inserta 7 y se aplica una rotación simple a izquierda. Al insertar 15 y 14 se debe aplicar una rotación doble compuesta de una rotación simple a derecha sobre 15 y un rotación simple a izquierda sobre 7. Y así seguiríamos paso por paso con cada número. Como se puede observar a continuación las rotaciones dobles deben hacerse en dos pasos:





Por tanto, el algoritmo de inserción en C++ sería:

```

53 template <class T>
54 bool InsertarAVL (info_nodo_AVL<T> * & raiz, T x) {
55     if (raiz == 0) {
56         // constructor: crea un nodo vacío con et=x
57         raiz = new info_nodo_AVL(x);
58         raiz->altura = 0;
59         return true; // el nodo ha podido insertarse con éxito
60     }
61     else {
62         if (x < raiz->et) {
63             if (raiz->hijoder != 0)
64                 raiz->hijoder->padre = raiz;
65                 if (InsertarAVL(raiz->hijoizq, x)) { // el árbol ha crecido,
66                     // vemos la diferencia de altura entre ambos hijos
67                     switch (Altura(raiz->hijoizq) - Altura(raiz->hijoder)) {
68                         // los valores del switch deben ser 0, 1 o 2, si son mayores
69                         // en algún momento no hemos insertado un elemento bien y
70                         // nuestro árbol no está equilibrado.
71                         case 0:
72                             return false; // el árbol no ha crecido
73
74                         case 1: // ha crecido por la izquierda, sumamos 1 a la altura
75                             raiz->altura++; // de la raíz
76                             return true;
77
78                         case 2:

```

□

```

79     /* CASO A*/
80     if (Altura(raiz->hijoizq->hijoizq) >
81         Altura(raiz->hijoizq->hijoder))
82         SimpleDerecha(raiz);
83
84     /*CASO B*/
85     else
86         Doble_IzquierdaDerecha(raiz);
87
88     return false; // la altura no crece porque hemos
89             // equilibrado el arbol
90 }
91 }
92 }
93
94 else { // x es mayor que la etiqueta
95     if (raiz->hijoizq != 0)
96         raiz->hijoizq->padre = raiz;
97
98     if (InsertarAVL(raiz->hijoder, x)) {
99         switch (Altura(raiz->hijoder) - Altura(raiz->hijoizq)) {
100             case 0:
101                 return false; // el arbol no ha crecido
102
103             case 1: // ha crecido por la izquierda, sumamos 1 a la altura
104                 raiz->altura++; // de la raiz
105                 return true;
106
107             case 2:
108                 /* CASO D */
109                 if (Altura(raiz->hijoder->hijoder) >
110                     Altura(raiz->hijoder->hijoizq))
111                     SimpleIzquierda(raiz);
112
113             /* CASO C */
114             else
115                 Doble_DerechoIzquierda(raiz);
116
117             return false;
118         }
119     }
120 }
121 }
122 }
```

