Capítulo 3

Arboles

Los árboles son contenedores que permiten organizar un conjunto de objetos en forma jerárquica. Ejemplos típicos son los diagramas de organización de las empresas o instituciones y la estructura de un sistema de archivos en una computadora. Los árboles sirven para representar fórmulas, la descomposición de grandes sistemas en sistemas más pequeños en forma recursiva y aparecen en forma sistemática en muchísimas aplicaciones de la computación científica. Una de las propiedades más llamativas de los árboles es la capacidad de acceder a muchísimos objetos desde un punto de partida o raíz en unos pocos pasos. Por ejemplo, en mi cuenta poseo unos 61,000 archivos organizados en unos 3500 directorios a los cuales puedo acceder con un máximo de 10 cambios de directorio (en promedio unos 5).

Sorprendentemente *no existe un contenedor STL de tipo árbol*, si bien varios de los otros contenedores (como conjuntos y correspondencias) están implementados internamente en términos de árboles. Esto se debe a que en la filosofía de las STL el árbol es considerado o bien como un subtipo del grafo o bien como una entidad demasiado básica para ser utilizada directamente por los usuarios.

3.1. Nomenclatura básica de árboles

Un árbol es una colección de elementos llamados *"nodos"*, uno de los cuales es la *"raíz"*. Existe una relación de parentesco por la cual cada nodo tiene un y sólo un *"padre"*, salvo la raíz que no lo tiene. El nodo es el concepto análogo al de *"posición"* en la lista, es decir un objeto abstracto que representa una posición en el mismo, no directamente relacionado con

el *"elemento"* o *"etiqueta"* del nodo. Formalmente, el árbol se puede definir recursivamente de la siguiente forma (ver figura 3.1)

- Un nodo sólo es un árbol
- Si n es un nodo y T_1, T_2, ..., T_k son árboles con raíces n_1, ..., n_k entonces podemos construir un nuevo árbol que tiene a n como raíz y donde n_1, ..., n_k son *"hijos"* de n.

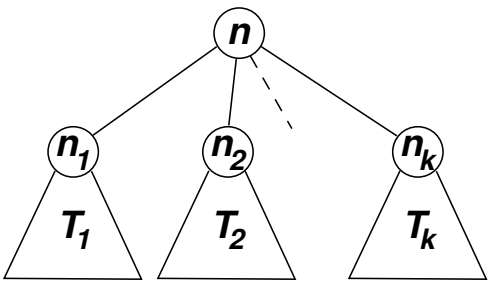

Figura 3.1: Construcción recursiva de un árbol

También es conveniente postular la existencia de un *"árbol vacío"* que llamaremos Λ.

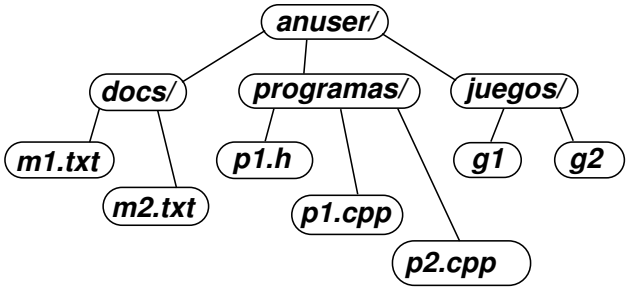

Figura 3.2: Arboles representando un sistema de archivos

Ejemplo 3.1: Consideremos el árbol que representa los archivos en un sistema de archivos. Los nodos del árbol pueden ser directorios o archivos. En el ejemplo de la figura 3.2, la cuenta **`anuser/`** contiene 3 subdirectorios **`docs/`**, **`programas/`** y **`juegos/`**, los cuales a su vez contienen una serie de archivos. En este caso la relación entre nodos hijos y padres corresponde a la de pertenencia: un nodo a es hijo de otro b, si el archivo a *pertenece* al

directorio b. En otras aplicaciones la relación padre/hijo puede representar querer significar otra cosa.

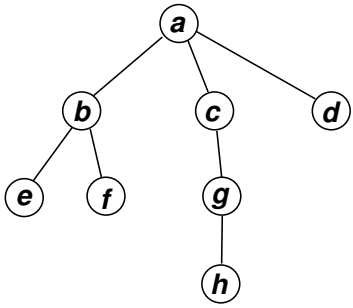

Figura 3.3: Ejemplo simple de árbol

Camino. Si $n_1, n_2, \ldots, n_k$ es una secuencia de nodos tales que n_i es padre de n_{i+1} para $i = 1 \ldots k-1$, entonces decimos que esta secuencia de nodos es un *"camino"* (*"path"*), de manera que {**anuser**, **docs**, **m2.txt**} es un camino, mientras que {**docs**, **anuser**, **programas**} no. (Coincidentemente en Unix se llama camino a la especificación completa de directorios que va desde el directorio raíz hasta un archivo, por ejemplo el camino correspondiente a **m2.txt** es **/anuser/docs/m2.txt**.) La *"longitud"* de un camino es igual al número de nodos en el camino menos uno, por ejemplo la longitud del camino {**anuser**, **docs**, **m2.txt**} es 2. Notar que siempre existe un camino de longitud 0 de un nodo a sí mismo.

Descendientes y antecesores. Si existe un camino que va del nodo a al b entonces decimos que a es antecesor de b y b es descendiente de a. Por ejemplo **m1.txt** es descendiente de **anuser** y **juegos** es antecesor de **g2**. Estrictamente hablando, un nodo es antecesor y descendiente de sí mismo ya que existe camino de longitud 0. Para diferenciar este caso trivial, decimos que a es descendiente (antecesor) propio de b si a es descendiente (antecesor) de b, *pero* $a \neq b$. En el ejemplo de la figura 3.3 a es antecesor propio de c, f y d,

Hojas. Un nodo que no tiene hijos es una *"hoja"* del árbol. (Recordemos que, por contraposición el nodo que no tiene padre es único y es la raíz.) En el ejemplo, los nodos e, f, h y d son hojas.

3.1.0.0.1. Altura de un nodo. La altura de un nodo en un árbol es la máxima longitud de un camino que va desde el nodo a una hoja. Por ejemplo, el árbol de la figura la altura del nodo c es 2. La altura del árbol es la altura de la raíz. La altura del árbol del ejemplo es 3. Notar que, para cualquier nodo n

$$\text{altura}(n) = \begin{cases} 0; & \text{si } n \text{ es una hoja} \\ 1 + \max_{s=\text{hijo de } n} \text{altura}(s); & \text{si no lo es.} \end{cases} \tag{3.1}$$

3.1.0.0.2. Profundidad de un nodo. Nivel. La *"profundidad"* de un nodo es la longitud de único camino que va desde la raíz al nodo. La profundidad del nodo g en el ejemplo es 2. Un *"nivel"* en el árbol es el conjunto de todos los nodos que están a una misma profundidad. El nivel de profundidad 2 en el ejemplo consta de los nodos e, f y g.

3.1.0.0.3. Nodos hermanos Se dice que los nodos que tienen un mismo padre son *"hermanos"* entre sí. Notar que no basta con que dos nodos estén en el mismo nivel para que sean hermanos. Los nodos f y g en el árbol de la figura 3.3 están en el mismo nivel, pero no son hermanos entre sí.

3.2. Orden de los nodos

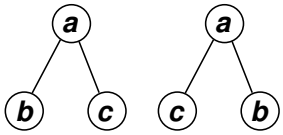

Figura 3.4: Arboles ordenados: el orden de los hijos es importante de manera que los árboles de la figura son diferentes.

En este capítulo, estudiamos árboles para los cuales el *orden* entre los hermanos es relevante. Es decir, los árboles de la figura 3.4 *son diferentes* ya que si bien a tiene los mismos hijos, están en diferente orden. Volviendo a la figura 3.3 decimos que el nodo c está a la derecha de b, o también que c es es el hermano derecho de b. También decimos que b es el *"hijo más a la izquierda"* de a. El orden entre los hermanos se propaga a los hijos, de manera que h está a la derecha de e ya que ambos son descendientes

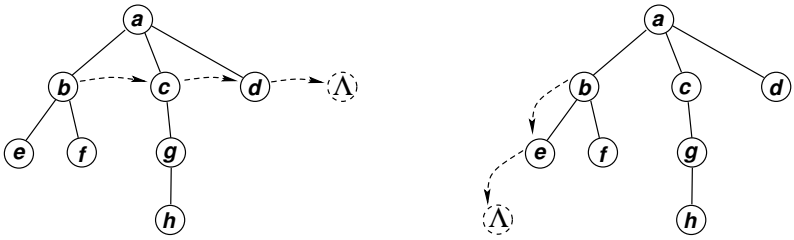

Figura 3.5: Direcciones posibles para avanzar en un árbol.

de c y b, respectivamente. A estos árboles se les llama *"árboles ordenados orientados"* (AOO).

Podemos pensar al árbol como una lista bidimensional. Así como en las listas se puede avanzar linealmente desde el comienzo hacia el fin, en cada nodo del árbol podemos avanzar en dos direcciones (ver figura 3.5)

- Por el hermano derecho, de esta forma se recorre toda la lista de hermanos de izquierda a derecha.

- Por el hijo más izquierdo, tratando de descender lo más posible en profundidad.

En el primer caso el recorrido termina en el último hermano a la derecha. Por analogía con la posición **end()** en las listas, asumiremos que después del último hermano existe un nodo ficticio no dereferenciable. Igualmente, cuando avanzamos por el hijo más izquierdo, el recorrido termina cuando nos encontramos con una hoja. También asumiremos que el hijo más izquierdo de una hoja es un nodo ficticio no dereferenciable. Notar que, a diferencia de la lista donde hay una sola posición no dereferenciable (la posición **end()**), en el caso de los árboles puede haber más de una posiciones ficticias no dereferenciables, las cuales simbolizaremos con Λ cuando dibujamos el árbol. En la figura 3.6 vemos todas las posibles posiciones ficticias $\Lambda_1, \ldots, \Lambda_8$ para el árbol de la figura 3.5. Por ejemplo, el nodo f no tiene hijos, de manera que genera la posición ficticia Λ_2.Tampoco tiene hermano derecho, de manera que genera la posición ficticia Λ_3.

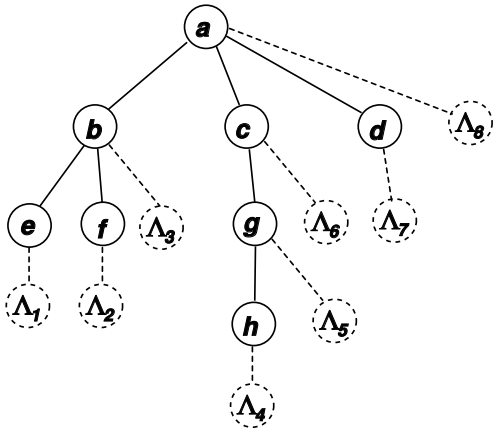

Figura 3.6: Todas las posiciones no dereferenciables de un árbol.

3.2.1. Particionamiento del conjunto de nodos

Ahora bien, dados dos nodos cualquiera m y n consideremos sus caminos a la raíz. Si m es descendiente de n entonces el camino de n está incluido en el de m o viceversa. Por ejemplo, el camino de c, que es a, c, está incluido en el de h, a, c, g, h, ya que c es antecesor de h. Si entre m y n no hay relación de descendiente o antecesor, entonces los caminos se deben bifurcar necesariamente en un cierto nivel. *El orden entre m y n es el orden entre los antecesores a ese nivel*. Esto demuestra que, dados dos nodos cualquiera m y n sólo una de las siguientes afirmaciones puede ser cierta

- $m = n$
- m es antecesor propio de n
- n es antecesor propio de m
- m está a la derecha de n
- n está a la derecha de m

Dicho de otra manera, dado un nodo n el conjunto N de todos los nodos del árbol se puede dividir en 5 conjuntos *disjuntos* a saber

$$N = \{n\} \cup \{\text{descendientes}(n)\} \cup \{\text{antecesores}(n)\} \cup \{\text{derecha}(n)\} \cup \{\text{izquierda}(n)\} \tag{3.2}$$

En la figura 3.7 vemos la partición inducida para los nodos c y f. Notar que en el caso del nodo f el conjunto de los descendientes es vacío ($\emptyset$).

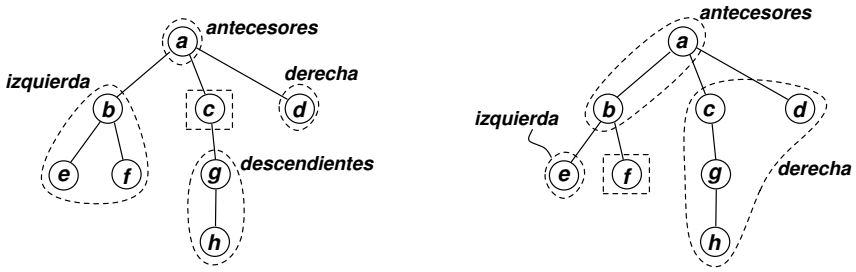

Figura 3.7: Clasificación de los nodos de un árbol con respecto a un nodo. Izquierda: con respecto al nodo c. Derecha: con respecto al nodo f

3.2.2. Listado de los nodos de un árbol

3.2.2.1. Orden previo

Existen varias formas de recorrer un árbol listando los nodos del mismo, generando una lista de nodos. Dado un nodo n con hijos n_1, n_2, $\ldots n_m$, el *"listado en orden previo"* (*"preorder"*) del nodo n que denotaremos como $\mathrm{oprev}(n)$ se puede definir recursivamente como sigue

$$\mathrm{oprev}(n) = (n, \mathrm{oprev}(n_1), \mathrm{oprev}(n_2), \ldots, \mathrm{oprev}(n_m)) \tag{3.3}$$

Además el orden previo del árbol vacío es la lista vacía: $\mathrm{oprev}(\Lambda) = ()$.

Consideremos por ejemplo el árbol de la figura 3.3. Aplicando recursivamente (3.3) tenemos

$$\begin{aligned}
\mathrm{oprev}(a) &= a, \mathrm{oprev}(b), \mathrm{oprev}(c), \mathrm{oprev}(d) \\
&= a, b, \mathrm{oprev}(e), \mathrm{oprev}(f), c, \mathrm{oprev}(g), d \\
&= a, b, \mathrm{oprev}(e), \mathrm{oprev}(f), c, \mathrm{oprev}(g), d \\
&= a, b, e, f, c, g, \mathrm{oprev}(h), d \\
&= a, b, e, f, c, g, h, d
\end{aligned} \tag{3.4}$$

Una forma más visual de obtener el listado en orden previo es como se muestra en la figura 3.8. Recorremos el borde del árbol en el sentido contrario a las agujas del reloj, partiendo de un punto imaginario a la izquierda del nodo raíz y terminando en otro a la derecha del mismo, como muestra la línea de puntos. Dado un nodo como el b el camino pasa cerca de él en varios puntos (3 en el caso de b, marcados con pequeños números en el camino). El orden previo consiste en *listar los nodos una sola vez, la primera vez que el camino*

pasa cerca del árbol. Así en el caso del nodo b, este se lista al pasar por 1. Queda como ejercicio para el lector verificar el orden resultante coincide con el dado en (3.4).

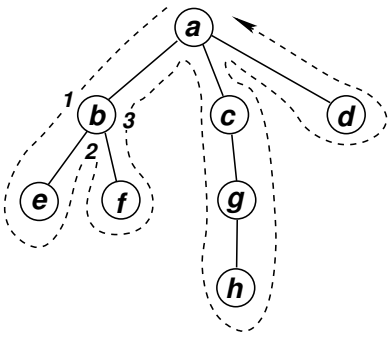

Figura 3.8: Recorrido de los nodos de un árbol en orden previo.

3.2.2.2. Orden posterior

El *"orden posterior"* (*"postorder"*) se puede definir en forma análoga al orden previo pero reemplazando (3.3) por

$$\operatorname{opost}(n) = (\operatorname{opost}(n_1), \operatorname{opost}(n_2), \ldots, \operatorname{opost}(n_m), n) \tag{3.5}$$

y para el árbol del ejemplo resulta ser

$$\begin{aligned} \operatorname{opost}(a) &= \operatorname{opost}(b), \operatorname{opost}(c), \operatorname{opost}(d), a \\ &= \operatorname{opost}(e), \operatorname{opost}(f), b, \operatorname{opost}(g), c, d, a \\ &= e, f, b, \operatorname{opost}(h), g, c, d, a \\ &= e, f, b, h, g, c, d, a \end{aligned} \tag{3.6}$$

Visualmente se puede realizar de dos maneras.

- Recorriendo el borde del árbol igual que antes (esto es en sentido contrario a las agujas del reloj), listando el nodo *la última vez que el recorrido pasa por al lado del mismo*. Por ejemplo el nodo b sería listado al pasar por el punto 3.

- Recorriendo el borde en el sentido opuesto (es decir en el mismo sentido que las agujas del reloj), y listando los nodos la *primera vez* que el

camino pasa cera de ellos. Una vez que la lista es obtenida, *invertimos la lista*. En el caso de la figura el recorrido en sentido contrario daría (a, d, c, g, h, b, f, e). Al invertirlo queda como en (3.6).

Existe otro orden que se llama *"simétrico"*, pero este sólo tiene sentido en el caso de árboles binarios, así que no será explicado aquí.

3.2.2.3. Orden posterior y la notación polaca invertida

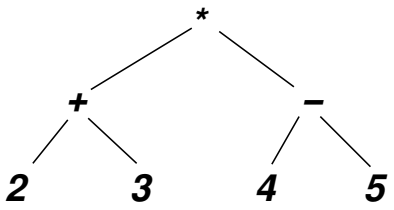

Figura 3.9: Arbol correspondiente a la expresión matemática $(2+3)*(4-5)$

Las expresiones matemáticas como $(2 + 3) * (4 - 5)$ se pueden poner en forma de árbol como se muestra en la figura 3.9. La regla es

- Para operadores binarios de la forma $a + b$ se pone el operador $(+)$ como padre de los dos operandos (a y b). Los operandos pueden ser a su vez expresiones. Funciones binarias como $rem(10, 5)$ (rem es la función resto) se tratan de esta misma forma.
- Operadores unarios (como -3) y funciones (como $\sin(20)$) se escriben poniendo el operando como hijo del operador o función.
- Operadores asociativos con más de dos operandos (como $1+3+4+9$) deben asociarse de a 2 (como en $(((1 + 3) + 4) + 9)$).

De esta forma, expresiones complejas como

$$(3 + \sin(4 + 20) * (5 - e^3)) * (20 + 10 - 7) \tag{3.7}$$

pueden ponerse en forma de árbol, como en la figura 3.10.

El listado en orden posterior de este árbol coincide con la notación polaca invertida (RPN) discutida en la sección §2.2.1.

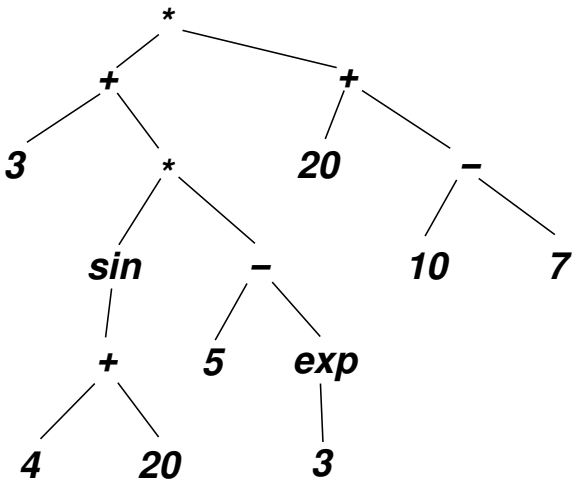

Figura 3.10: Arbol correspondiente a la expresión matemática (3.7)

3.2.3. Notación Lisp para árboles

Una expresión matemática compleja que involucra funciones cuyos argumentos son a su vez llamadas a otras funciones puede ponerse en forma de árbol. Por ejemplo, para la expresión

$$f(g(a,b),h(t,u,v),q(r,s(w))) \tag{3.8}$$

corresponde un árbol como el de la figura 3.11. En este caso cada función es un nodo cuyos hijos son los argumentos de la función. En Lisp la llamada a un función $f(x,y,z)$ se escribe de la forma `(f x y z)`, de manera que la llamada anterior se escribiría como

```
1.  (f (g a b) (h t u v) (q r (s w)))
```

Para expresiones más complejas como la de (3.7), la forma Lisp para el árbol (figura 3.10) da el código Lisp correspondiente

```
1.  (* (+ 3 (* (sin (+ 4 20)) (- 5 (exp 3)))) (+ 20 (- 10 7)))
```

Esta notación puede usarse para representar árboles en forma general, de manera que, por ejemplo, el árbol de la figura 3.3 puede ponerse, en notación Lisp como

```
1. (a (b e f) (c (g h)) d)
```

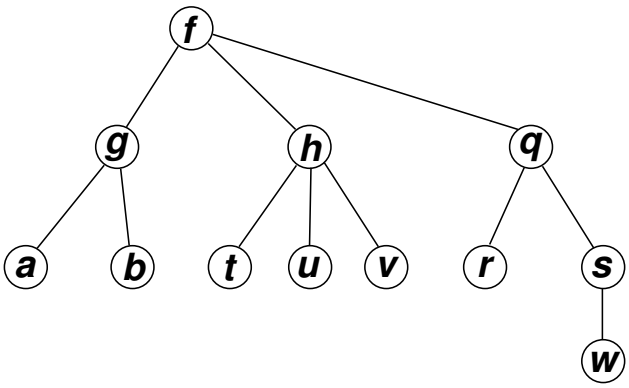

Figura 3.11: Arbol correspondiente a una expresión de llamadas a funciones.

Notemos que el orden de los nodos es igual al del orden previo. Se puede dar una definición precisa de la notación Lisp como para el caso de los órdenes previo y posterior:

$$\text{lisp}(n) = \begin{cases} \text{si } n \text{ es una hoja:} & n \\ \text{caso contrario:} & (n\ \text{lisp}(n_1)\ \text{lisp}(n_2)\ \ldots\ \text{lisp}(n_m)) \end{cases} \tag{3.9}$$

donde $n_1 \ldots n_m$ son los hijos del nodo n.

Es evidente que existe una relación unívoca entre un árbol y su notación Lisp. Los paréntesis dan la estructura adicional que permite establecer la relación unívoca. La utilidad de esta notación es que permite fácilmente escribir árboles en una línea de texto, sin tener que recurrir a un gráfico. Basado en esta notación, es fácil escribir una función que convierta un árbol a una lista y viceversa.

También permite *"serializar"* un árbol, es decir, convertir una estructura *"bidimensional"* como es el árbol, en una estructura unidimensional como es una lista. El serializar una estructura compleja permite almcenarla en disco o comunicarla a otro proceso por mensajes.

3.2.4. Reconstrucción del árbol a partir de sus órdenes

Podemos preguntarnos si podemos reconstruir un árbol a partir de su listado en orden previo. Si tal cosa fuera posible, entonces sería fácil representar árboles en una computadora, almacenando dicha lista. Sin embargo puede verse fácilmente que árboles distintos pueden dar el mismo orden previo (ver figura 3.12) o posterior (ver figura 3.13).

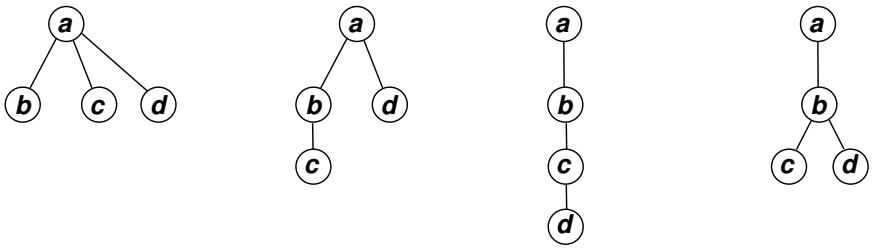

Figura 3.12: Los cuatro árboles de la figura tienen el mismo orden previo (a, b, c, d)

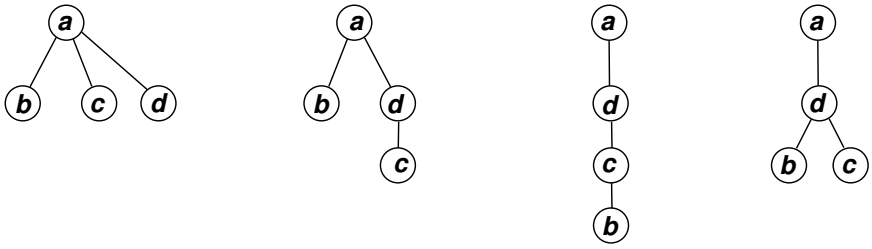

Figura 3.13: Los cuatro árboles de la figura tienen el mismo orden posterior (b, c, d, a)

Sin embargo, es destacable que, dado el orden previo y posterior de un árbol sí se puede reconstruir el árbol. Primero notemos que (3.3) implica que el orden de los nodos queda así

$$\text{oprev}(n) = (n, n_1, \text{descendientes}(n_1), n_2, \text{descendientes}(n_2), \ldots, n_m, \text{descendientes}(n \tag{3.10}$$

mientras que

$$\text{opost}(n) = (\text{descendientes}(n_1), n_1, \text{descendientes}(n_2), n_2, \ldots, \text{descendientes}(n_m), n_m \tag{3.11}$$

Notemos que el primer nodo listado en orden previo es la raíz, y el segundo su primer hijo n_1. Todos los nodos que están *después* de n_1 en orden previo pero *antes* de n_1 en orden posterior son los descendientes de n_1. Prestar atención a que el orden en que aparecen los descendientes de un dado nodo en (3.10) puede no coincidir con el que aparecen en (3.11). De esta forma podemos deducir cuales son los descendientes de n_1. El nodo siguiente, en orden previo, a todos los descendientes de n_1 debe ser el segundo hijo n_2. Todos los nodos que están después de n_2 en orden previo pero antes de n_2

en orden posterior son descendientes de n_2. Así siguiendo podemos deducir cuales son los hijos de n y cuales son descendientes de cada uno de ellos.

Ejemplo 3.2: *Consigna:* Encontrar el árbol A tal que

$$\begin{aligned} \text{orden previo} &= (z, w, a, x, y, c, m, t, u, v) \\ \text{orden posterior} &= (w, x, y, a, t, u, v, m, c, z) \end{aligned} \tag{3.12}$$

Solución: De los primeros dos nodos en orden previo se deduce que z debe ser el nodo raíz y w su primer hijo. No hay nodos antes de w en orden posterior de manera que w no tiene hijos. El nodo siguiente a w en orden previo es a que por lo tanto debe ser el segundo hijo de z. Los nodos que están antes de a pero después de w en orden posterior son x e y, de manera que estos son descendientes de a. De la misma forma se deduce que el tercer hijo de z es c y que sus descendientes son m, t, u, v. A esta altura podemos esbozar un dibujo del árbol como se muestra en la figura 3.14. Las líneas de puntos indican que, por ejemplo, sabemos que m, t, u, v son descendientes de c, pero todavía no conocemos la estructura de ese subárbol.

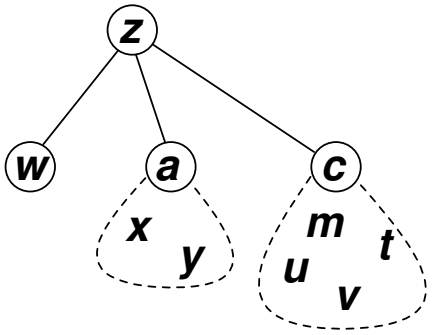

Figura 3.14: Etapa parcial en la reconstrucción del árbol del ejemplo 3.2

Ahora bien, para hallar la estructura de los descendientes de c volvemos a (3.12), y vemos que

$$\begin{aligned} \text{oprev}(c) &= (c, m, t, u, v) \\ \text{opost}(c) &= (t, u, v, m, c) \end{aligned} \tag{3.13}$$

de manera que el procedimiento se puede aplicar recursivamente para hallar los hijos de c y sus descendientes y así siguiendo hasta reconstruir todo el árbol. El árbol correspondiente resulta ser, en este caso el de la figura 3.15, o en notación Lisp `(z w (a x y) (c (m t u v)))`.

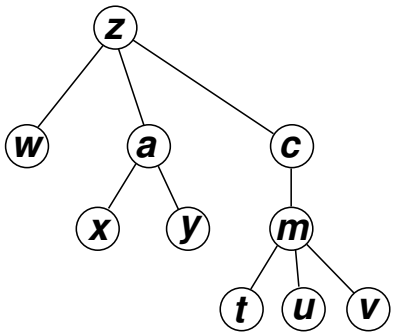

Figura 3.15: Arbol reconstruido a partir de los órdenes previo y posterior especificados en (3.12)

3.3. Operaciones con árboles

3.3.1. Algoritmos para listar nodos

Implementar un algoritmo para recorrer los nodos de un árbol es relativamente simple debido a su naturaleza intrínsecamente recursiva, expresada en (3.3). Un posible algoritmo puede observarse en el código 3.1. Si bien el algoritmo es genérico hemos usado ya algunos conceptos familiares de las STL, por ejemplo las posiciones se representan con una clase `iterator`. Recordar que para árboles se puede llegar al *"fin del contenedor"*, es decir los nodos Λ, en más de un punto del contenedor. El código genera una lista de elementos `L` con los elementos de `T` en orden previo.

```
1. void preorder(tree &T,iterator n,list &L) {
2.   L.insert(L.end(),/* valor en el nodo 'n'... */);
3.   iterator c = /* hijo mas izquierdo de n... */;
4.   while (/* 'c' no es 'Lambda'... */) {
5.     preorder(T,c,L);
6.     c = /* hermano a la derecha de c... */;
7.   }
8. }
```

Código 3.1: *Algoritmo para recorrer un árbol en orden previo. [Archivo: preorder.cpp]*

```
1.  void postorder(tree &T,iterator n,list &L) {
2.    iterator c = /* hijo mas izquierdo de n . . . */;
3.    while (c != T.end()) {
4.      postorder(T,c,L);
5.      c = /* hermano a la derecha de c . . . */;
6.    }
7.    L.insert(L.end(),/* valor en el nodo 'n'. . . */);
8.  }
```

Código 3.2: *Algoritmo para recorrer un árbol en orden posterior. [Archivo: postorder.cpp]*

```
1.  void lisp_print(tree &T,iterator n) {
2.    iterator c = /* hijo mas izquierdo de n . . . */;
3.    if (/* 'c' es 'Lambda'. . . */) {
4.      cout << /* valor en el nodo 'n'. . . */;
5.    } else {
6.      cout << "(" << /* valor de 'n' . . . */;
7.      while (/* 'c' no es 'Lambda'. . . */) {
8.        cout << " ";
9.        lisp_print(T,c);
10.       c = /* hermano derecho de c . . . */;
11.     }
12.     cout << ")";
13.   }
14. }
```

Código 3.3: *Algoritmo para imprimir los datos de un árbol en notación Lisp. [Archivo: lispprint.cpp]*

En el código 3.2 se puede ver un código similar para generar la lista con el orden posterior, basada en (3.5). Similarmente, en código 3.3 puede verse la implementación de una rutina que imprime la notación Lisp de un árbol.

3.3.2. Inserción en árboles

Para construir árboles necesitaremos rutinas de inserción supresión de nodos. Como en las listas, las operaciones de inserción toman un elemento y una posición e insertan el elemento en esa posición en el árbol.

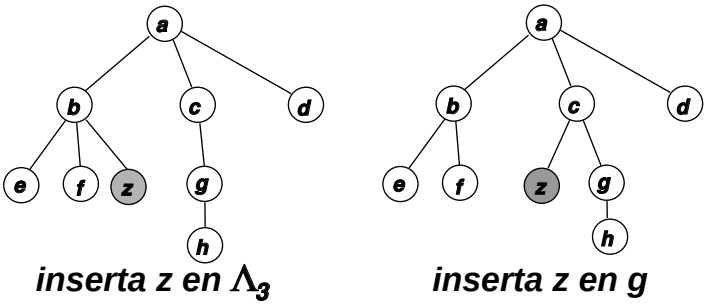

Figura 3.16: Resultado de insertar el elemento z en el árbol de la figura 3.6. *Izquierda:* Inserta z en la posición Λ_3. *Derecha:* Inserta z en la posición g.

- Cuando insertamos un nodo en una posición Λ entonces simplemente el elemento pasa a generar un nuevo nodo en donde estaba el nodo ficticio Λ. Por ejemplo, el resultado de insertar el elemento z en el nodo Λ_3 de la figura 3.6 se puede observar en la figura 3.16 (izquierda). (*Observación:* En un abuso de notación estamos usando las mismas letras para denotar el contenido del nodo que el nodo en sí.)

- Cuando insertamos un nodo en una posición dereferenciable, entonces simplemente el elemento pasa a generar un nuevo nodo hoja en el lugar en esa posición, tal como operaría la operación de inserción del TAD lista en la lista de hijos. Por ejemplo, consideremos el resultado de insertar el elemento z en la posición g. El padre de g es c y su lista de hijos es (g). Al insertar z en la posición de g la lista de hijos pasa a ser (z, g), de manera que z pasa a ser el hijo más izquierdo de c (ver figura 3.16 derecha).

- Así como en listas `insert(p,x)` invalida las posiciones después de `p` (inclusive), en el caso de árboles, una inserción en el nodo `n` invalida las posiciones que son descendientes de `n` y que están a la derecha de `n`.

3.3.2.1. Algoritmo para copiar árboles

```
1. iterator tree_copy(tree &T,iterator nt,
2.                    tree &Q,iterator nq) {
3.   nq = /* nodo resultante de insertar el
4.           elemento de 'nt' en 'nq' . . . */;
```

```
5.   iterator
6.     ct = /* hijo mas izquierdo de 'nt' . . .*/,
7.     cq = /* hijo mas izquierdo de 'nq' . . .*/;
8.   while (/* 'ct' no es 'Lambda'. . . */) {
9.     cq = tree_copy(T,ct,Q,cq);
10.    ct = /* hermano derecho de 'ct'. . . */;
11.    cq = /* hermano derecho de 'cq'. . . */;
12.  }
13.  return nq;
14. }
```

Código 3.4: *Seudocódigo para copiar un árbol. [Archivo: treecpy.cpp]*

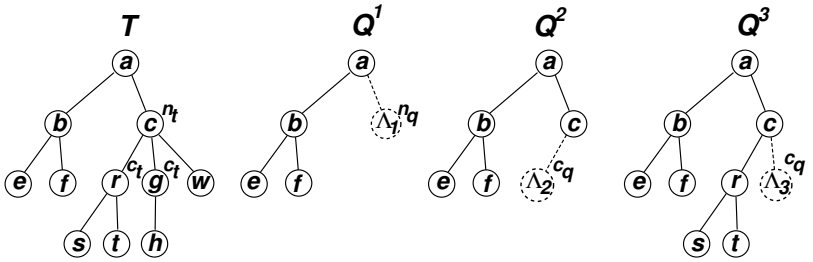

Figura 3.17: Algoritmo para copiar árboles.

Con estas operaciones podemos escribir el seudocódigo para una función que copia un árbol (ver código 3.4). Esta función copia el subárbol del nodo `nt` en el árbol `T` en la posición `nq` en el árbol `Q` y devuelve la posición de la raíz del subárbol insertado en `Q` (actualiza el nodo `nq` ya que después de la inserción es inválido). La función es recursiva, como lo son la mayoría de las operaciones no triviales sobre árboles. Consideremos el algoritmo aplicado al árbol de la figura 3.17 a la izquierda. Primero inserta el elemento que esta en `nt` en la posición `nq`. Luego va copiando cada uno de los subárboles de los hijos de `nq` como hijos del nodo `nt`. El nodo `ct` itera sobre los hijos de `nt` mientras que `cq` lo hace sobre los hijos de `nq`. Por ejemplo, si consideramos la aplicación del algoritmo a la copia del árbol de la figura 3.17 a la izquierda, concentrémonos en la copia del subárbol del nodo c del árbol T al Q.

Cuando llamamos a `tree_copy(T,nt,Q,nq)`, `nt` es c y `nq` es Λ_1, (mostrado como Q^1 en la figura). La línea 3 copia la raíz del subárbol que en este caso es el nodo c insertándolo en Λ_1. Después de esta línea, el

árbol queda como se muestra en la etapa Q^2. Como en la inserción en listas, la línea actualiza la posición **nq**, la cuál queda apuntando a la posición que contiene a c. Luego **ct** y **nt** toman los valores de los hijos más izquierdos, a saber r y Λ_2. Como **ct** no es Λ entonces el algoritmo entra en el lazo y la línea 9 copia todo el subárbol de r en Λ_2, quedando el árbol como en Q^3. De paso, la línea actualiza el iterator **cq**, de manera que **ct** y **cq** quedan apuntando a los dos nodos r en sus respectivos árboles. Notar que en este análisis no consideramos la llamada recursiva a **tree_copy()** sino que simplemente asumimos que estamos analizando la instancia específica de llamada a **tree_copy** donde **nt** es c y no aquéllas llamadas generadas por esta instancia. En las líneas 10–11, los iterators **ct** y **cq** son avanzados, de manera que quedan apuntando a g y Λ_2. En la siguiente ejecución del lazo la línea 9 copiará todo el subárbol de g a Λ_3. El proceso se detiene después de copiar el subárbol de w en cuyo caso **ct** obtendrá en la línea 10 un nodo Λ y la función termina, retornando el valor de **nq** actualizado.

```
1.  iterator mirror_copy(tree &T,iterator nt,
2.                       tree &Q,iterator nq) {
3.    nq = /* nodo resultante de insertar
4.            el elemento de 'nt' en 'nq' */;
5.    iterator
6.      ct = /* hijo mas izquierdo de 'nt' . . .*/,
7.      cq = /* hijo mas izquierdo de 'nq' . . .*/;
8.    while (/* 'ct' no es 'Lambda'. . . */) {
9.      cq = mirror_copy(T,ct,Q,cq);
10.     ct = /* hermano derecho de 'ct' . . . */;
11.   }
12.   return nq;
13. }
```

Código 3.5: *Seudocódigo para copiar un árbol en espejo. [Archivo: mirrorcpy.cpp]*

Con menores modificaciones la función puede copiar un árbol en forma espejada, es decir, de manera que todos los nodos hermanos queden en orden inverso entre sí. Para eso basta con *no avanzar* el iterator **cq** donde se copian los subárboles, es decir eliminar la línea 11 . Recordar que si en una lista se van insertando valores en una posición *sin avanzarla*, entonces los elementos quedan ordenados *en forma inversa* a como fueron ingresa-

dos. El algoritmo de copia espejo **mirror_copy()** puede observarse en el código 3.5.

Con algunas modificaciones el algoritmo puede ser usado para obtener la copia de un árbol reordenando las hojas en un orden arbitrario, por ejemplo dejándolas ordenadas entre sí.

3.3.3. Supresión en árboles

Al igual que en listas, solo se puede suprimir en posiciones dereferenciables. En el caso de suprimir en un nodo hoja, solo se elimina el nodo. Si el nodo tiene hijos, eliminarlo equivale a eliminar todo el subárbol correspondiente. Como en listas, eliminando un nodo devuelve la posición del hermano derecho que llena el espacio dejado por el nodo eliminado.

```
1.  iterator_t prune_odd(tree &T,iterator_t n) {
2.    if (/*valor de 'n'. . . */ % 2)
3.      /* elimina el nodo 'n' y refresca. . . */;
4.    else {
5.      iterator_t c =
6.        /* hijo mas izquierdo de 'n' . . . */;
7.      while (/*'c' no es 'Lambda' . . . */)
8.        c = prune_odd(T,c);
9.      n = /* hermano derecho de 'n'. . . */;
10.   }
11.   return n;
12. }
```

Código 3.6: *Algoritmo que elimina los nodos de un árbol que son impares, incluyendo todo su subárbol [Archivo: pruneodd.cpp]*

Por ejemplo consideremos el algoritmo **prune_odd** (ver código 3.6) que *"poda"* un árbol, eliminando todos los nodos de un árbol que son impares *incluyendo sus subárboles.* Por ejemplo, si `T=(6 (2 3 4) (5 8 10))`. Entonces después de aplicar **prune_odd** tenemos `T=(6 (2 4))`. Notar que los nodos 8 y 10 han sido eliminados ya que, si bien son pares, pertenecen al subárbol del nodo 5, que es impar. Si el elemento del nodo es impar todo el subárbol del nodo es eliminado en la línea 3, caso contrario los hijos son podados aplicándoles recursivamente la función. Notar que tanto si el valor contenido en **n** es impar como si no, **n** avanza una posición dentro de **prune_odd**, ya sea al eliminar el nodo en la línea 3 o al avanzar explícitamente en la línea 9.

3.3.4. Operaciones básicas sobre el tipo árbol

Los algoritmos para el listado presentados en las secciones previas, sugieren las siguientes operaciones abstractas sobre árboles

- Dado un nodo (posición o iterator sobre el árbol), obtener su hijo más izquierdo. (Puede retornar una posición Λ).
- Dado un nodo obtener su hermano derecho. (Puede retornar una posición Λ).
- Dada una posición, determinar si es Λ o no.
- Obtener la posición de la raíz del árbol.
- Dado un nodo obtener una referencia al dato contenido en el nodo.
- Dada una posición (dereferenciable o no) y un dato, insertar un nuevo nodo con ese dato en esa posición.
- Borrar un nodo y todo su subárbol correspondiente.

3.4. Interfaz básica para árboles

```
1.  class iterator_t {
2.    /* . . . . */
3.  public:
4.    iterator_t lchild();
5.    iterator_t right();
6.  };
7.
8.  class tree {
9.    /* . . . . */
10. public:
11.   iterator_t begin();
12.   iterator_t end();
13.   elem_t &retrieve(iterator_t p);
14.   iterator_t insert(iterator_t p,elem_t t);
15.   iterator_t erase(iterator_t p);
16.   void clear();
17.   iterator_t splice(iterator_t to,iterator_t from);
18. };
```

Código 3.7: *Interfaz básica para árboles. [Archivo: treebas1.h]*

Una interfaz básica, parcialmente compatible con la STL puede observarse en el código 3.7. Como con las listas y correspondencias tenemos una clase `iterator_t` que nos permite iterar sobre los nodos del árbol, tanto dereferenciables como no dereferenciables. En lo que sigue `T` es un árbol, `p`, `q` y `r` son nodos (iterators) y `x` es un elemento de tipo `elem_t`.

- `q = p.lchild()`: Dada una posición dereferenciable `p` retorna la posición del hijo más izquierdo (*"leftmost child"*). La posición retornada puede ser dereferenciable o no.
- `q = p.right()`: Dada una posición dereferenciable `p` retorna la posición del hermano derecho. La posición retornada puede ser dereferenciable o no.
- `T.end()`: retorna un iterator no dereferenciable.
- `p=T.begin()`: retorna la posición del comienzo del árbol, es decir la raíz. Si el árbol está vacío, entonces retorna `end()`.
- `x = T.retrieve(p)`: Dada una posición dereferenciable retorna una referencia al valor correspondiente.
- `q = T.insert(p,x)`: Inserta un nuevo nodo en la posición `p` conteniendo el elemento `x`. `p` puede ser dereferenciable o no. Retorna la posición del nuevo elemento insertado.
- `p = T.erase(p)`: Elimina la posición dereferenciable `p` y todo el subárbol de `p`.
- `T.clear()`: Elimina todos los elementos del árbol (equivale a `T.erase(T.begin())`).
- `T.splice(to,from)`: Elimina todo el subárbol del nodo dereferenciable `from` y lo inserta en el nodo (dereferenciable o no) `to`. `to` y `from` no deben tener relación de antecesor o descendiente y pueden estar en diferentes árboles. Por ejemplo, consideremos el ejemplo de la figura 3.18 (izquierda), donde deseamos mover todo el subárbol del nodo `r` en el árbol `T` a la posición del nodo `v` en el árbol `Q`. El resultado es como se muestra en la parte izquierda de la figura y se obtiene con el llamado `T.splice(r,v)`.

```
1. void preorder(tree &T,iterator_t n,list<int> &L) {
2.   L.insert(L.end(),T.retrieve(n));
3.
4.   iterator_t c = n.lchild();
5.   while (c!=T.end()) {
6.     preorder(T,c,L);
```

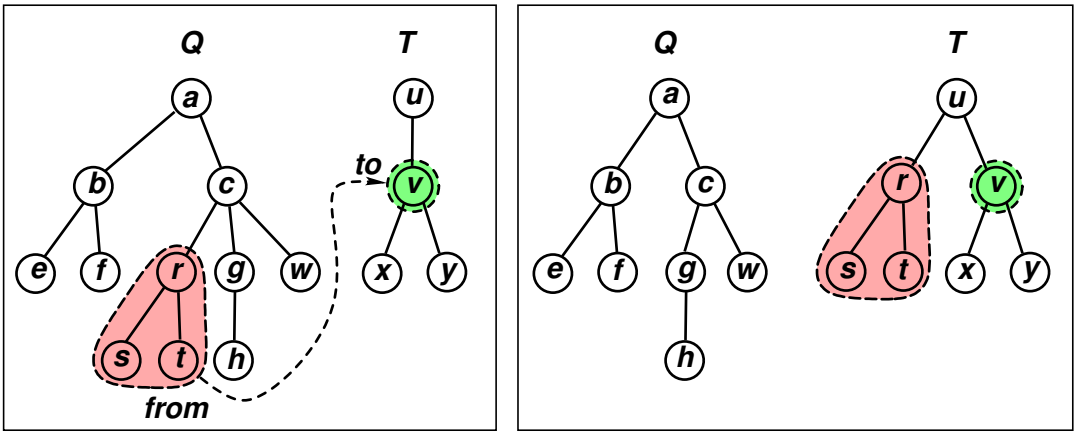

Figura 3.18: Operación `splice`.

```
7.    c = c.right();
8.   }
9.  }
10. void preorder(tree &T,list<int> &L) {
11.  if (T.empty()) return;
12.  preorder(T,T.begin(),L);
13. }
14.
15. //---:---<*>---:---<*>---:---<*>---:---<*>
16. void postorder(tree &T,iterator_t n,list<int> &L) {
17.  iterator_t c = n.lchild();
18.  while (c!=T.end()) {
19.   postorder(T,c,L);
20.   c = c.right();
21.  }
22.  L.insert(L.end(),T.retrieve(n));
23. }
24. void postorder(tree &T,list<int> &L) {
25.  if (T.empty()) return;
26.  postorder(T,T.begin(),L);
27. }
28.
29. //---:---<*>---:---<*>---:---<*>---:---<*>
30. void lisp_print(tree &T,iterator_t n) {
31.  iterator_t c = n.lchild();
32.  if (c==T.end()) cout << T.retrieve(n);
33.  else {
```

```
34.     cout << "(" << T.retrieve(n);
35.     while (c!=T.end()) {
36.       cout << " ";
37.       lisp_print(T,c);
38.       c = c.right();
39.     }
40.     cout << ")";
41.   }
42. }
43. void lisp_print(tree &T) {
44.   if (T.begin()!=T.end()) lisp_print(T,T.begin());
45. }
46.
47. //---:---<*>---:---<*>---:---<*>---:---<*>
48. iterator_t tree_copy(tree &T,iterator_t nt,
49.                       tree &Q,iterator_t nq) {
50.   nq = Q.insert(nq,T.retrieve(nt));
51.   iterator_t
52.     ct = nt.lchild(),
53.     cq = nq.lchild();
54.   while (ct!=T.end()) {
55.     cq = tree_copy(T,ct,Q,cq);
56.     ct = ct.right();
57.     cq = cq.right();
58.   }
59.   return nq;
60. }
61.
62. void tree_copy(tree &T,tree &Q) {
63.   if (T.begin() != T.end())
64.     tree_copy(T,T.begin(),Q,Q.begin());
65. }
66.
67. //---:---<*>---:---<*>---:---<*>---:---<*>
68. iterator_t mirror_copy(tree &T,iterator_t nt,
69.                         tree &Q,iterator_t nq) {
70.   nq = Q.insert(nq,T.retrieve(nt));
71.   iterator_t
72.     ct = nt.lchild(),
73.     cq = nq.lchild();
74.   while (ct != T.end()) {
75.     cq = mirror_copy(T,ct,Q,cq);
76.     ct = ct.right();
77.   }
78.   return nq;
79. }
```

```
80.
81. void mirror_copy(tree &T,tree &Q) {
82.   if (T.begin() != T.end())
83.     mirror_copy(T,T.begin(),Q,Q.begin());
84. }
85.
86. //---:---<*>---:---<*>---:---<*>---:---<*>
87. iterator_t prune_odd(tree &T,iterator_t n) {
88.   if (T.retrieve(n) % 2) n = T.erase(n);
89.   else {
90.     iterator_t c = n.lchild();
91.     while (c != T.end()) c = prune_odd(T,c);
92.     n = n.right();
93.   }
94.   return n;
95. }
96.
97. void prune_odd(tree &T) {
98.   if (T.begin() != T.end()) prune_odd(T,T.begin());
99. }
```

Código 3.8: *Diversos algoritmos sobre árboles con la interfaz básica. [Archivo: treetools.cpp]*

Los algoritmos **preorder**, **postorder**, **lisp_print**, **tree_copy**, **mirror_copy** y **prune_odd** descriptos en las secciones previas se encuentran implementados con las funciones de la interfaz básica en el código 3.8.

3.4.1. Listados en orden previo y posterior y notación Lisp

El listado en orden previo es simple. Primero inserta, el elemento del nodo **n** en el fin de la lista **L**. Notar que para obtener el elemento se utiliza el método **retrieve**. Luego se hace que **c** apunte al hijo más izquierdo de **n** y se va aplicando **preorder()** en forma recursiva sobre los hijos **c**. En la línea 7 se actualiza **c** de manera que recorra la lista de hijos de **n**. La función **postorder** es completamente análoga, sólo que el elemento de **n** es agregado *después* de los órdenes posteriores de los hijos.

3.4.2. Funciones auxiliares para recursión y sobrecarga de funciones

En general estas funciones recursivas se escriben utilizando una función auxiliar. En principio uno querría llamar a la función como **`preorder(T)`** *asumiendo que se aplica al nodo raíz de* `T`. Pero para después poder aplicarlo en forma recursiva necesitamos agregar un argumento adicional que es un nodo del árbol. Esto lo podríamos hacer usando una función recursiva adicional **`preorder_aux(T,n)`**. Finalmente, haríamos que **`preorder(T)`** llame a **`preorder_aux`**:

[copy]

```
1. void preorder_aux(tree &T,iterator_t n,list<int> &L) {
2.   /* . . . */
3. }
4. void preorder(tree &T,list<int> &L) {
5.   preorder_aux(T,T.begin(),L);
6. }
```

Pero como C++ admite *"sobrecarga del nombre de funciones"*, no es necesario declarar la función auxiliar con un nombre diferente. Simplemente hay dos funciones **`preorder()`**, las cuales se diferencian por el número de argumentos.

A veces se dice que la función **`preorder(T)`** actúa como un *"wrapper"* (*"envoltorio"*) para la función **`preorder(T,n)`**, que es la que hace el trabajo real. **`preorder(T)`** sólo se encarga de pasarle los parámetros correctos a **`preorder(T,n)`**. De paso podemos usar el wrapper para realizar algunos chequeos como por ejemplo verificar que el nodo **`n`** no sea Λ. Si un nodo Λ es pasado a **`preorder(T,n)`** entonces seguramente se producirá un error al querer dereferenciar **`n`** en la línea 2.

Notar que Λ puede ser pasado a **`preorder(T,n)`** sólo si **`T`** es vacío, ya que una vez que un nodo dereferenciable es pasado a **`preorder(T,n)`**, el test de la línea 5 se encarga de no dejar nunca pasar un nodo Λ a una instancia inferior de **`preorder(T,n)`**. Si **`T`** no es vacío, entonces **`T.begin()`** es dereferenciable y a partir de ahí nunca llegará a **`preorder(T,n)`** un nodo Λ. Notar que es mucho más eficiente verificar que el árbol no este vacío en **`preorder(T)`** que hacerlo en **`preorder(T,n)`** ya que en el primer caso la verificación se hace una sola vez para todo el árbol.

La rutina **`lisp_print`** es básicamente equivalente a las de orden previo y orden posterior. Una diferencia es que **`lisp_print`** no apendiza a una lista, ya que en ese caso habría que tomar alguna convención para representar

los paréntesis. **lisp_print()** simplemente imprime por terminal la notación Lisp del árbol.

3.4.3. Algoritmos de copia

Pasemos ahora a estudiar **tree_copy()** y **mirror_copy()**. Notar el llamado a **insert** en la línea 50. Notar que la línea actualiza el valor de **nq** ya que de otra manera quedaría inválido por la inserción. Notar como las líneas 52–53 y 56–57 van manteniendo los iterators **ct** y **cq** sobre posiciones equivalentes en **T** y **Q** respectivamente.

3.4.4. Algoritmo de poda

Notar que si el elemento en **n** es impar, todo el subárbol de **n** es eliminado, y **n** queda apuntando al hermano derecho, por el **erase** de la línea 88, caso contrario, **n** queda apuntando al hermano derecho por el avance explícito de la línea 92.

3.5. Implementación de la interfaz básica por punteros

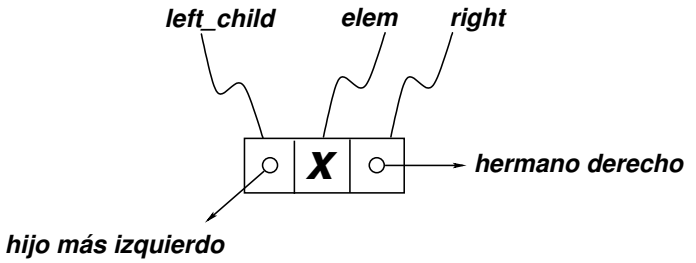

Figura 3.19: Celdas utilizadas en la representación de árboles por punteros.

Así como la implementación de listas por punteros mantiene los datos en celdas enlazadas por un campo **next**, es natural considerar una implementación de árboles en la cual los datos son almacenados en celdas que contienen, además del dato **elem**, un puntero **right** a la celda que corresponde al hermano derecho y otro **left_child** al hijo más izquierdo (ver

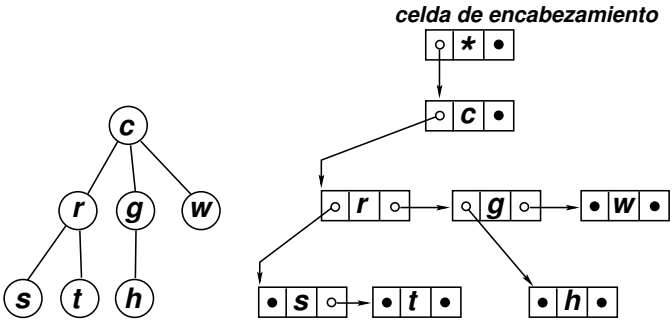

Figura 3.20: Representación de un árbol con celdas enlazadas por punteros.

figura 3.19). En la figura 3.20 vemos un árbol simple y su representación mediante celdas enlazadas.

3.5.1. El tipo iterator

Por analogía con las listas podríamos definir el tipo `iterator_t` como un `typedef` a `cell *`. Esto bastaría para representar posiciones dereferenciables y posiciones no dereferenciables que provienen de haber aplicado `right()` al último hermano, como la posición Λ_3 en la figura 3.21. Por supuesto habría que mantener el criterio de usar *"posiciones adelantadas"* con respecto al dato. Sin embargo no queda en claro como representar posiciones no dereferenciables como la Λ_1 que provienen de aplicar `lchild()` a una hoja.

Una solución posible consiste en hacer que el tipo `iterator_t` contenga, además de un puntero a la celda que contiene el dato, punteros a celdas que de otra forma serían inaccesibles. Entonces el iterator consiste en *tres punteros a celdas* (ver figura 3.22) a saber,

- Un puntero `ptr` a la celda que contiene el dato. (Este puntero es nulo en el caso de posiciones no dereferenciables).
- Un puntero `prev` al *hermano izquierdo*.
- Un puntero `father` al padre.

Así, por ejemplo a continuación mostramos los juegos de punteros correspondientes a varias posiciones dereferenciables y no dereferenciables en el árbol de la figura 3.6:

- nodo e: `ptr=`e, `prev=NULL`, `father=`b.

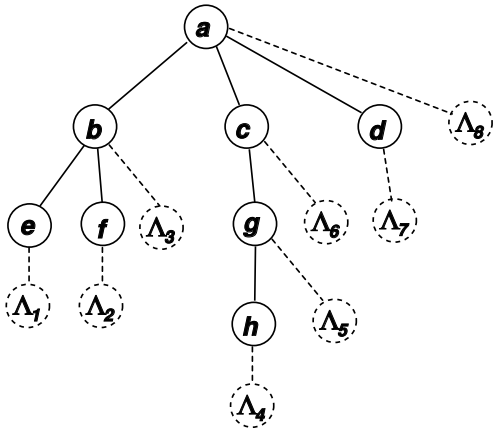

Figura 3.21: Todas las posiciones no dereferenciables de un árbol.

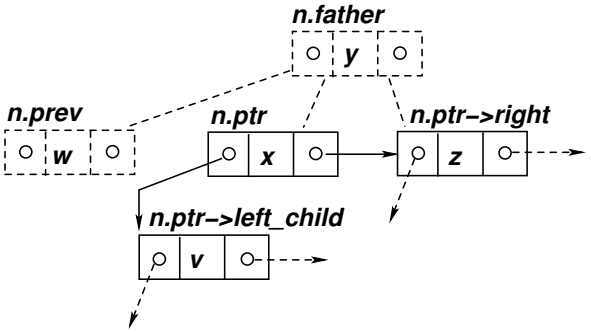

Figura 3.22: Entorno local de un iterator sobre árboles.

- nodo f: **ptr=**f, **prev=**e, **father=**b.
- nodo Λ_1: **ptr=NULL**, **prev=NULL**, **father=**e.
- nodo Λ_2: **ptr=NULL**, **prev=NULL**, **father=**f.
- nodo Λ_3: **ptr=NULL**, **prev=**f, **father=**b.
- nodo g: **ptr=**g, **prev=NULL**, **father=**c.
- nodo Λ_6: **ptr=NULL**, **prev=**g, **father=**c.

Estos tres punteros tienen la suficiente información como para ubicar a todas las posiciones (dereferenciables o no) del árbol. Notar que todas las posiciones tienen un puntero **father** no nulo, mientras que el puntero **prev** puede o no ser nulo.

Para tener un código más uniforme se introduce una celda de encabeza-

miento, al igual que con las listas. La raíz del árbol, si existe, es una celda hija de la celda de encabezamiento. Si el árbol está vacío, entonces el iterator correspondiente a la raíz (y que se obtiene llamando a **begin()**) corresponde a **ptr=NULL**, **prev=NULL**, **father=**celda de encabezamiento.

3.5.2. Las clases cell e iterator_t

```
1.  class tree;
2.  class iterator_t;
3.
4.  //---:---<*>---:---<*>---:---<*>---:---<*>
5.  class cell {
6.    friend class tree;
7.    friend class iterator_t;
8.    elem_t elem;
9.    cell *right, *left_child;
10.   cell() : right(NULL), left_child(NULL) {}
11. };
12.
13. //---:---<*>---:---<*>---:---<*>---:---<*>
14. class iterator_t {
15. private:
16.   friend class tree;
17.   cell *ptr,*prev,*father;
18.   iterator_t(cell *p,cell *prev_a, cell *f_a)
19.     : ptr(p), prev(prev_a), father(f_a) { }
20. public:
21.   iterator_t(const iterator_t &q) {
22.     ptr = q.ptr;
23.     prev = q.prev;
24.     father = q.father;
25.   }
26.   bool operator!=(iterator_t q) { return ptr!=q.ptr; }
27.   bool operator==(iterator_t q) { return ptr==q.ptr; }
28.   iterator_t()
29.     : ptr(NULL), prev(NULL), father(NULL) { }
30.
31.   iterator_t lchild() {
32.     return iterator_t(ptr->left_child,NULL,ptr);
33.   }
34.   iterator_t right() {
35.     return iterator_t(ptr->right,ptr,father);
36.   }
```

```
37.   };
38.
39.   //---:---<*>---:---<*>---:---<*>---:---<*>
40.   class tree {
41.   private:
42.     cell *header;
43.     tree(const tree &T) {}
44.   public:
45.
46.     tree() {
47.       header = new cell;
48.       header->right = NULL;
49.       header->left_child = NULL;
50.     }
51.     ~tree() { clear(); delete header; }
52.
53.     elem_t &retrieve(iterator_t p) {
54.       return p.ptr->elem;
55.     }
56.
57.     iterator_t insert(iterator_t p,elem_t elem) {
58.       assert(!(p.father==header && p.ptr));
59.       cell *c = new cell;
60.       c->right = p.ptr;
61.       c->elem = elem;
62.       p.ptr = c;
63.       if (p.prev) p.prev->right = c;
64.       else p.father->left_child = c;
65.       return p;
66.     }
67.     iterator_t erase(iterator_t p) {
68.       if(p==end()) return p;
69.       iterator_t c = p.lchild();
70.       while (c!=end()) c = erase(c);
71.       cell *q = p.ptr;
72.       p.ptr = p.ptr->right;
73.       if (p.prev) p.prev->right = p.ptr;
74.       else p.father->left_child = p.ptr;
75.       delete q;
76.       return p;
77.     }
78.
79.     iterator_t splice(iterator_t to,iterator_t from) {
80.       assert(!(to.father==header && to.ptr));
81.       if (from.ptr->right == to.ptr) return from;
82.       cell *c = from.ptr;
83.
```

```
84.       if (from.prev) from.prev->right = c->right;
85.       else from.father->left_child = c->right;
86.
87.       c->right = to.ptr;
88.       to.ptr = c;
89.       if (to.prev) to.prev->right = c;
90.       else to.father->left_child = c;
91.
92.       return to;
93.     }
94.
95.     iterator_t find(elem_t elem) {
96.       return find(elem,begin());
97.     }
98.     iterator_t find(elem_t elem,iterator_t p) {
99.       if(p==end() || retrieve(p) == elem) return p;
100.      iterator_t q,c = p.lchild();
101.      while (c!=end()) {
102.        q = find(elem,c);
103.        if (q!=end()) return q;
104.        else c = c.right();
105.      }
106.      return iterator_t();
107.    }
108.    void clear() { erase(begin()); }
109.    iterator_t begin() {
110.      return iterator_t(header->left_child,NULL,header);
111.    }
112.    iterator_t end() { return iterator_t(); }
```

Código 3.9: *Implementación de la interfaz básica de árboles por punteros. [Archivo: treebas.h]*

Una implementación de la interfaz básica código 3.7 por punteros puede observarse en el código 3.9.

- Tenemos primero las declaraciones *"hacia adelante"* de **tree** e **iterator_t**. Esto nos habilita a declarar **friend** a las clases **tree** e **iterator_t**.
- La clase **cell** sólo declara los campos para contener al puntero al hijo más izquierdo y al hermano derecho. El constructor inicializa los punteros a **NULL**.

- La clase **iterator_t** declara **friend** a **tree**, pero no es necesario hacerlo con **cell**. **iterator_t** declara los campos punteros a celdas y por comodidad declaramos un constructor privado **iterator_t(cell *p,cell *pv, cell *f)** que simplemente asigna a los punteros internos los valores de los argumentos.
- Por otra parte existe un constructor público **iterator_t(const iterator_t &q)**. Este es el *"constructor por copia"* y es utilizado cuando hacemos por ejemplo **iterator_t p(q);** con **q** un iterador previamente definido.
- El operador de asignación de iteradores (por ejemplo **p=q**) es *sintetizado* por el compilador, y simplemente se reduce a una copia *bit a bit* de los datos miembros de la clase (en este caso los tres punteros **prt**, **prev** y **father**) lo cual es apropiado en este caso.
- Haber definido **iterator_t** como una clase (y no como un **typedef**) nos obliga a definir también los operadores **!=** y **==**. Esto nos permitirá comparar nodos por igualdad o desigualdad (**p==q** o **p!=q**). De la misma forma que con las posiciones en listas, los nodos *no* pueden compararse con los operadores de relación de orden (**<**, **<=**, **>** y **>=**). Notar que los operadores **==** y **!=** sólo comparan el campo **ptr** de forma que *todas las posiciones no dereferenciables (Λ) son "iguales" entre sí*. Esto permite comparar en los lazos cualquier posición **p** con **end()**, entonces **p==end()** retornará **true** incluso si **p** no es exactamente igual a **end()** (es decir tiene campos **father** y **prev** diferentes).
- El constructor por defecto **iterator_t()** devuelve un iterator con los tres punteros nulos. Este iterator no debería ser normalmente usado en ninguna operación, pero es invocado automáticamente por el compilador cuando declaramos iterators como en: **iterator p;**
- Las operaciones **lchild()** y **right()** son las que nos permiten movernos dentro del árbol. Sólo pueden aplicarse a posiciones dereferenciables y pueden retornar posiciones dereferenciables o no. Para entender como funcionan consideremos la información contenida en un iterator. Si consideramos un iterator **n** (ver figura 3.23), entonces la posición de **m=n.lchild()** está definida por los siguientes punteros

  - **m.ptr= n.ptr->left_child**
  - **m.prev= NULL** (ya que **m** es un *hijo más izquierdo*)
  - **m.father= n.ptr**

  Por otra parte, si consideramos la función hermano derecho: **q=n.right()**, entonces **q** está definida por los siguientes punteros,

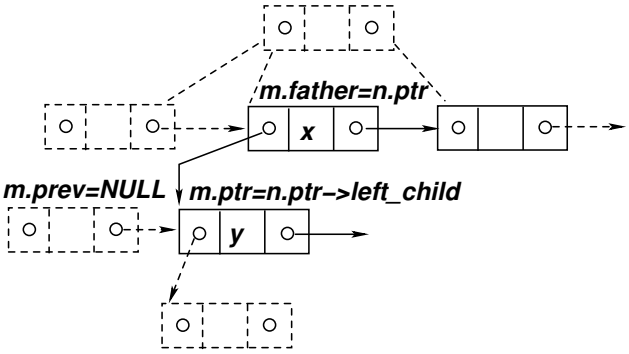

Figura 3.23: La función `lchild()`.

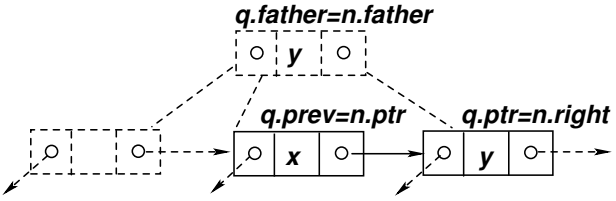

Figura 3.24: La función `right()`.

- `q.ptr= n.ptr->right`
- `q.prev= n.ptr`
- `q.father= n.father`

3.5.3. La clase tree

- La clase **`tree`** contiene un único dato que es un puntero a la celda de encabezamiento. Esta celda es alocada e inicializada en el constructor **`tree()`**.
- La función **`retrieve(p)`** simplemente retorna el dato contenido en la celda apuntada por **`p.ptr`**.
- La función **`insert(p,x)`** aloca una nueva celda **`c`** e inicializa sus campos datos. El dato **`elem`** se obtiene del argumento a la llamada y el puntero al hermano derecho pasa a ser el puntero **`ptr`** de la posición donde se va a insertar, ya que (al igual que con las listas) el valor insertado queda *a la izquierda* de la posición donde se inserta. Como la

nueva celda no va a tener hijos, el puntero **left_child** queda en **NULL** (esto se hace al crear la celda en el constructor de la clase **cell**). Después de insertar y enlazar la nueva celda hay que calcular la posición de la nueva celda insertada, que es el valor de retorno de **insert()**.

- **p.ptr=c**, la nueva celda insertada.
- **p.prev** no se altera ya que el hermano a la izquierda de la nueva celda es el mismo que el de la celda donde se insertó.
- **p.father** tampoco cambia, porque la nueva celda tiene el mismo padre que aquella posición donde se insertó.

Finalmente hay que actualizar los punteros en algunas celdas vecinas.

- Si **p.prev** no es nulo, entonces la celda *no es el hijo más izquierdo* y por lo tanto hay que actualizar el puntero **right** de la celda a la izquierda.
- Caso contrario, la nueva celda *pasa a ser el hijo más izquierdo* de su padre y por lo tanto hay que actualizar el puntero **left_child** de éste.

- En **erase(p)** la línea 70 eliminan todos los subárboles de **p** en forma recursiva. Notar que esta parte del código es genérica (independiente de la implementación particular). Finalmente las líneas 71–76 eliminan la celda correspondiente a **p** actualizando los punteros a las celdas vecinas si es necesario.

- La función **splice(to,from)** primero elimina el subárbol de **from** (líneas 84–85) en forma muy similar a las líneas 73–74 de **erase** pero sin eliminar recursivamente el subárbol, como en **erase()** ya que debe ser insertado en la posición **to**. Esta inserción se hace en las líneas 87–90, notar la similitud de estas líneas con la función **insert()**.

- La función **find(elem)** no fue descrita en la interfaz básica pero es introducida aquí. Retorna un iterator al nodo donde se encuentra el elemento **elem**. La implementación es completamente genérica se hace recursivamente definiendo la función auxiliar **find(elem,p)** que busca el elemento **elem** *en el subárbol del nodo* p.

- **clear()** llama a **erase()** sobre la raíz.

- **`begin()`** construye el iterator correspondiente a la raíz del árbol, es decir que los punteros correspondientes se obtienen a partir de la celda de encabezamiento como
  - **`ptr=header->left_child`**
  - **`prev=NULL`**
  - **`father=header`**

- Se ha incluido un *"constructor por copia"* (línea 43) en la parte privada. Recordemos que el constructor por copia es usado cuando un usuario declara objetos en la siguente forma

[copy]
```
1.  tree T2(T1);
```

es decir, al declarar un nuevo objeto **`T2`** a partir de uno preexistente **`T1`**.

Para todas las clases que contienen punteros a otros objetos (y que pertenecen a la clase, es decir que debe encargarse de alocarlos y desalocarlos, como son las celdas en el caso de listas y árboles) hay que tener cuidado. Si uno deja que el compilador sintetice un constructor por copia entonces la copia se hace *"bit a bit"* con lo cual para los punteros simplemente copia el valor del puntero, no creando duplicados de los objetos apuntados. A esto se le llama *"shallow copy"*. De esta forma el objeto original y el duplicado comparten objetos internos, y eso debe hacerse con cuidado, o puede traer problemas. A menos que uno, por alguna razón prefiera este comportamiento, en general hay que optar por una de las siguientes alternativas

- Implementar el constructor por copia correctamente, es decir copiando los componentes internos (*"deep copy"*). Este constructor funcionaría básicamente como la función **`tree_copy()`** descripta más arriba (ver §3.3.2.1). (La implementación de la interfaz avanzada y la de árbol binario están hechas así).
- Declarar al constructor por copia, implementándolo con un cuerpo vacío y poniéndolo en la parte privada. Esto hace que si el usuario intenta escribir un código como el de arriba, el compilador dará un mensaje de error. Esto evita que un usuario desprevenido que no sabe que el constructor por copia no hace el *"deep copy"*, lo use por accidente. Por ejemplo

[copy]
```
1.  tree T1;
```

```
2. // pone cosas en T1...
3. tree T2(T1);
4. // Obtiene un nodo en T2
5. iterator_t n = T2.find(x);
6. // vacia el arbol T1
7. T1.clear();
8. // Intenta cambiar el valor en 'n'
9. T2.retrieve(n) = y; // ERROR!!
```

En este ejemplo, el usuario obtiene una *"shallow copy"* `T2` del árbol `T1` y genera un iterator a una posición dereferenciable `n` en `T2`. Después de vaciar el árbol `T1` y querer acceder al elemento en `n` genera un error en tiempo de ejecución, ya que en realidad la estructura interna de `T2` era compartida con `T1`. Al vaciar `T1`, se vacía también `T2`.

Con la inclusión de la línea 43 del código 3.9. , la instrucción `tree T2(T1);` genera un error en tiempo de compilación.

- Implementarlo como público, pero que de un error.

[copy]

```
1. public:
2.   tree(const tree &T) {
3.     error("Constructor por copia no implementado!!");
4.   }
```

De esta forma, si el usuario escribe `tree T2(T1);` entonces compilará pero en tiempo de ejecución, si pasa por esa línea va a dar un error.

3.6. Interfaz avanzada

```
1. #ifndef AED_TREE_H
2. #define AED_TREE_H
3.
4. #include <cassert>
5. #include <iostream>
6. #include <cstddef>
7. #include <cstdlib>
8.
9. namespace aed {
10.
11.   //---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>---:
12.   template<class T>
13.   class tree {
```

```
14.   public:
15.     class iterator;
16.   private:
17.     class cell {
18.       friend class tree;
19.       friend class iterator;
20.       T t;
21.       cell *right, *left_child;
22.       cell() : right(NULL), left_child(NULL) {}
23.     };
24.     cell *header;
25.
26.     iterator tree_copy_aux(iterator nq,
27.                            tree<T> &TT,iterator nt) {
28.       nq = insert(nq,*nt);
29.       iterator
30.         ct = nt.lchild(),
31.         cq = nq.lchild();
32.       while (ct!=TT.end()) {
33.         cq = tree_copy_aux(cq,TT,ct);
34.         ct = ct.right();
35.         cq = cq.right();
36.       }
37.       return nq;
38.     }
39.   public:
40.     static int cell_count_m;
41.     static int cell_count() { return cell_count_m; }
42.     class iterator {
43.     private:
44.       friend class tree;
45.       cell *ptr,*prev,*father;
46.       iterator(cell *p,cell *prev_a,cell *f_a) : ptr(p),
47.         prev(prev_a), father(f_a) { }
48.     public:
49.       iterator(const iterator &q) {
50.         ptr = q.ptr;
51.         prev = q.prev;
52.         father = q.father;
53.       }
54.       T &operator*() { return ptr->t; }
55.       T *operator->() { return &ptr->t; }
56.       bool operator!=(iterator q) { return ptr!=q.ptr; }
57.       bool operator==(iterator q) { return ptr==q.ptr; }
58.       iterator() : ptr(NULL), prev(NULL), father(NULL) { }
59.
60.       iterator lchild() { return iterator(ptr->left_child,NULL,ptr); }
```

```
61.       iterator right() { return iterator(ptr->right,ptr,father); }
62.
63.       // Prefix:
64.       iterator operator++() {
65.         *this = right();
66.         return *this;
67.       }
68.       // Postfix:
69.       iterator operator++(int) {
70.         iterator q = *this;
71.         *this = right();
72.         return q;
73.       }
74.     };
75.
76.     tree() {
77.       header = new cell;
78.       cell_count_m++;
79.       header->right = NULL;
80.       header->left_child = NULL;
81.     }
82.     tree<T>(const tree<T> &TT) {
83.       if (&TT != this) {
84.         header = new cell;
85.         cell_count_m++;
86.         header->right = NULL;
87.         header->left_child = NULL;
88.         tree<T> &TTT = (tree<T> &) TT;
89.         if (TTT.begin()!=TTT.end())
90.           tree_copy_aux(begin(),TTT,TTT.begin());
91.       }
92.     }
93.     tree &operator=(tree<T> &TT) {
94.       if (this != &TT) {
95.         clear();
96.         tree_copy_aux(begin(),TT,TT.begin());
97.       }
98.       return *this;
99.     }
100.    ~tree() { clear(); delete header; cell_count_m--; }
101.    iterator insert(iterator p,T t) {
102.      assert(!(p.father==header && p.ptr));
103.      cell *c = new cell;
104.      cell_count_m++;
105.      c->right = p.ptr;
106.      c->t = t;
107.      p.ptr = c;
```

```
108.      if (p.prev) p.prev->right = c;
109.      else p.father->left_child = c;
110.      return p;
111.    }
112.    iterator erase(iterator p) {
113.      if(p==end()) return p;
114.      iterator c = p.lchild();
115.      while (c!=end()) c = erase(c);
116.      cell *q = p.ptr;
117.      p.ptr = p.ptr->right;
118.      if (p.prev) p.prev->right = p.ptr;
119.      else p.father->left_child = p.ptr;
120.      delete q;
121.      cell_count_m--;
122.      return p;
123.    }
124.
125.    iterator splice(iterator to,iterator from) {
126.      assert(!(to.father==header && to.ptr));
127.      if (from.ptr->right == to.ptr) return from;
128.      cell *c = from.ptr;
129.
130.      if (from.prev) from.prev->right = c->right;
131.      else from.father->left_child = c->right;
132.
133.      c->right = to.ptr;
134.      to.ptr = c;
135.      if (to.prev) to.prev->right = c;
136.      else to.father->left_child = c;
137.
138.      return to;
139.    }
140.    iterator find(T t) { return find(t,begin()); }
141.    iterator find(T t,iterator p) {
142.      if(p==end() || p.ptr->t == t) return p;
143.      iterator q,c = p.lchild();
144.      while (c!=end()) {
145.        q = find(t,c);
146.        if (q!=end()) return q;
147.        else c++;
148.      }
149.      return iterator();
150.    }
151.    void clear() { erase(begin()); }
152.    iterator begin() { return iterator(header->left_child,NULL,header); }
153.    iterator end() { return iterator(); }
154.
```

```
155.   };
156.
157.   template<class T>
158.   int tree<T>::cell_count_m = 0;
159.
160.   template<class T>
161.   void swap(tree<T> &T1, tree<T> &T2) { T1.swap(T2); }
162. }
163. #endif
```

Código 3.10: *Interfaz avanzada para árboles. [Archivo: tree.h]*

Una interfaz similar a la descripta en la sección §3.4 pero incluyendo templates, clases anidadas y sobrecarga de operadores puede observarse en el código 3.10.

- La clase `tree` pasa a ser ahora un template, de manera que podremos declarar `tree<int>`, `tree<double>`.
- Las clases `cell` e `iterator` son ahora clases anidadas dentro de `tree`. Externamente se verán como `tree<int>::cell` y `tree<int>::iterator`. Sin embargo, sólo `iterator` es pública y es usada *fuera* de `tree`.
- La dereferenciación de posiciones (nodos) `x=retrieve(p)` se reemplaza por `x=*p`. Para eso debemos *"sobrecargar"* el operador `*`. Si el tipo elemento (es decir el tipo `T` del template) contiene campos, entonces vamos a querer extraer campos de un elemento almacenado en un nodo, por lo cual debemos hacer `(*p).campo`. Para poder hacer esto usando el operador `->` (es decir `p->campo`) debemos sobrecargar el operador `->`. Ambos operadores devuelven referencias de manera que es posible usarlos en el miembro izquierdo, como en `*p=x` o `p->campo=z`.
- Igual que con la interfaz básica, para poder hacer comparaciones de iterators debemos sobrecargar también los operadores `==` y `!=`. También tiene definido el constructor por copia.
- El avance por hermano derecho `p = p.right();` ahora se puede hacer con `p++`, de todas formas mantenemos la función `right()` que a veces resulta ser más compacta. Por ejemplo `q = p.right()` se traduce en `q=p; q++;` en la versión con operadores.

- La función estática **cell_count()**, permite obtener el número total de celdas alocadas por todos las instancias de la clase, e incluye las celdas de encabezamiento. Esta función fue introducida para debugging, normalmente no debería ser usada por los usuarios de la clase. Como es estática puede invocarse como **tree<int>::cell_count()** o también sobre una instancia, **T.cell_count()**.
- Se ha incluido un constructor por copia, de manera que se puede copiar árboles usando directamente el operador **=**, por ejemplo

```
[copy]
1.  tree<int> T,Q;
2.  // carga elementos en T . . . .
3.  Q = T;
```

Así como también pasar árboles por copia y definir contenedores que contienen árboles como por ejemplo una lista de árboles de enteros. **list< tree<int> >**. Esta función necesita otra función recursiva auxiliar que hemos llamado **tree_copy_aux()** y que normalmente no debería ser usada directamente por los usuarios de la clase, de manera que la incluimos en la sección privada.

3.6.1. Ejemplo de uso de la interfaz avanzada

```
1.  typedef tree<int> tree_t;
2.  typedef tree_t::iterator node_t;
3.
4.  int count_nodes(tree_t &T,node_t n) {
5.    if (n==T.end()) return 0;
6.    int m=1;
7.    node_t c = n.lchild();
8.    while(c!=T.end()) m += count_nodes(T,c++);
9.    return m;
10. }
11.
12. int count_nodes(tree_t &T) {
13.   return count_nodes(T,T.begin());
14. }
15.
16. int height(tree_t &T,node_t n) {
17.   if (n==T.end()) return -1;
18.   node_t c = n.lchild();
19.   if (c==T.end()) return 0;
20.   int son_max_height = -1;
```

```
21.   while (c!=T.end()) {
22.     int h = height(T,c);
23.     if (h>son_max_height) son_max_height = h;
24.     c++;
25.   }
26.   return 1+son_max_height;
27. }
28.
29. int height(tree_t &T) {
30.   return height(T,T.begin());
31. }
32.
33. void
34. node_level_stat(tree_t &T,node_t n,
35.                int level,vector<int> &nod_lev) {
36.   if (n==T.end()) return;
37.   assert(nod_lev.size()>=level);
38.   if (nod_lev.size()==level) nod_lev.push_back(0);
39.   nod_lev[level]++;
40.   node_t c = n.lchild();
41.   while (c!=T.end()) {
42.     node_level_stat(T,c++,level+1,nod_lev);
43.   }
44. }
45.
46. void node_level_stat(tree_t &T,
47.                    vector<int> &nod_lev) {
48.   nod_lev.clear();
49.   node_level_stat(T,T.begin(),0,nod_lev);
50.   for (int j=0;j<nod_lev.size();j++) {
51.     cout << "[level: " << j
52.          << ", nodes: " << nod_lev[j] << "]";
53.   }
54.   cout << endl;
55. }
56.
57. //---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>---:
58. int max_node(tree_t &T,node_t n) {
59.   if (n==T.end()) return -1;
60.   int w = *n;
61.   node_t c = n.lchild();
62.   while (c!=T.end()) {
63.     int ww = max_node(T,c++);
64.     if (ww > w) w = ww;
65.   }
66.   return w;
```

```
67. }
68.
69. int max_node(tree_t &T) {
70.   return max_node(T,T.begin());
71. }
72.
73. //---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>---:
74. int max_leaf(tree_t &T,node_t n) {
75.   if (n==T.end()) return -1;
76.   int w = *n;
77.   node_t c = n.lchild();
78.   if (c==T.end()) return w;
79.   w = 0;
80.   while (c!=T.end()) {
81.     int ww = max_leaf(T,c++);
82.     if (ww > w) w = ww;
83.   }
84.   return w;
85. }
86.
87. int max_leaf(tree_t &T) {
88.   return max_leaf(T,T.begin());
89. }
90.
91. //---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>---:
92. int leaf_count(tree_t &T,node_t n) {
93.   if (n==T.end()) return 0;
94.   node_t c = n.lchild();
95.   if (c==T.end()) return 1;
96.   int w = 0;
97.   while (c!=T.end()) w += leaf_count(T,c++);
98.   return w;
99. }
100.
101. int leaf_count(tree_t &T) {
102.   return leaf_count(T,T.begin());
103. }
```

Código 3.11: *Algunos ejemplos de uso de la interfaz avanzada para árboles. [Archivo: treetools2.cpp]*

En el código 3.11 vemos algunos ejemplos de uso de esta interfaz.

- Todo los ejemplos usan árboles de enteros, `tree<int>`. Los `typedef` de las líneas 1-2 permiten definir tipos `tree_t` y `node_t` que abrevian el código.
- Las funciones implementadas son
  - `height(T)` su altura,
  - `count_nodes(T)` cuenta los nodos de un árbol,
  - `leaf_count(T)` el número de hojas,
  - `max_node(T)` el máximo valor del elemento contenido en los nodos,
  - `max_leaf(T)` el máximo valor del elemento contenido en las hojas,
- `node_level_stat(T,nod_lev)` calcula el número de nodos que hay en cada nivel del árbol, el cual se retorna en el `vector<int> nod_lev`, es decir, `nod_lev[l]` es el número de nodos en el nivel `l`.

3.7. Tiempos de ejecución

Operación	$T(n)$
`begin()`, `end()`, `n.right()`, `n++`, `n.left_child()`, `*n`, `insert()`, `splice(to,from)`	$O(1)$
`erase()`, `find()`, `clear()`, `T1=T2`	$O(n)$

Tabla 3.1: Tiempos de ejecución para operaciones sobre árboles.

En la Tabla 3.1 vemos los tiempos de ejecución para las diferentes operaciones sobre árboles. Es fácil ver que todas las funciones básicas tienen costo $O(1)$. Es notable que una función como `splice()` también sea $O(1)$. Esto se debe a que la operación de mover todo el árbol de una posición a otra se realiza con una operación de punteros. Las operaciones que no son $O(1)$ son `erase(p)` que debe eliminar todos los nodos del subárbol del nodo `p`, `clear()` que equivale a `erase(begin())`, `find(x)` y el constructor por copia (`T1=T2`). En todos los casos n es o bien el número de nodos del subárbol (`erase(p)` y `find(x,p)`) o bien el número total de nodos del árbol (`clear()`, `find(x)` y el constructor por copia `T1=T2`).

3.8. Arboles binarios

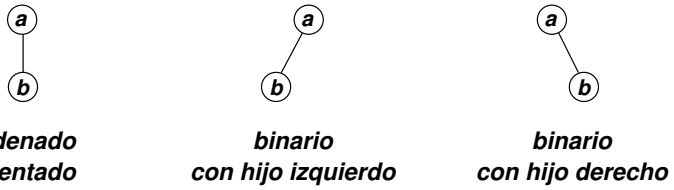

Figura 3.25: Diferentes casos de un árbol con dos nodos.

Los árboles que hemos estudiado hasta ahora son *"árboles ordenados orientados"* (AOO) ya que los hermanos están ordenados entre sí y hay una orientación de los caminos desde la raíz a las hojas. Otro tipo importante de árbol es el *"árbol binario"* (AB) en el cual cada nodo puede tener a lo sumo dos hijos. Además, si un dado nodo n tiene un sólo hijo, entonces este puede ser el hijo derecho o el hijo izquierdo de n. Por ejemplo si consideramos las posibles estructuras de árboles con dos nodos (ver figura 3.25), tenemos que para el caso de un AOO la única posibilidad es un nodo raíz con un nodo hijo. Por otra parte, si el árbol es binario, entonces existen dos posibilidades, que el único hijo sea el hijo izquierdo o el derecho. Dicho de otra forma los AB del centro y la derecha son diferentes, mientras que si fueran AOO entonces serían ambos iguales al de la izquierda.

3.8.1. Listados en orden simétrico

Los listados en orden previo y posterior para AB coinciden con su versión correspondiente para AOO. El *"listado en orden simétrico"* se define recursivamente como

$$\begin{aligned} \mathrm{osim}(\Lambda) &=< \text{lista vacía} > \\ \mathrm{osim}(n) &= (\mathrm{osim}(s_l), n, \mathrm{osim}(s_r)) \end{aligned} \tag{3.14}$$

donde $s_{l,r}$ son los hijos izquierdo y derecho de n, respectivamente.

3.8.2. Notación Lisp

La notación Lisp para árboles debe ser modificada un poco con respecto a la de AOO, ya que debemos introducir algún tipo de notación para un hijo Λ. Básicamente, en el caso en que un nodo tiene un sólo hijo, reemplazamos

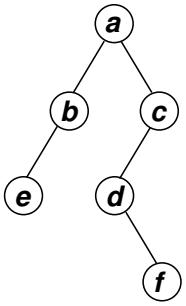

Figura 3.26: Ejemplo de árbol binario.

con un punto la posición del hijo faltante. La definición recursiva es

$$\text{lisp}(n) = \begin{cases} n & ;\ \text{si } s_l = \Lambda \text{ y } s_r = \Lambda \\ (n\ \text{lisp}(s_l)\ \text{lisp}(s_r)) & ;\ \text{si } s_l \neq \Lambda \text{ y } s_r \neq \Lambda \\ (n\ \cdot\ \text{lisp}(s_r)) & ;\ \text{si } s_l = \Lambda \text{ y } s_r \neq \Lambda \\ (n\ \text{lisp}(s_l)\ \cdot) & ;\ \text{si } s_l \neq \Lambda \text{ y } s_r = \Lambda \end{cases} \tag{3.15}$$

Por ejemplo, la notación Lisp del árbol de la figura 3.26 es

$$\text{lisp}(a) = (a\ (b\ e\ \cdot)\ (c\ (d\ \cdot\ f)\ \cdot)) \tag{3.16}$$

3.8.3. Árbol binario lleno

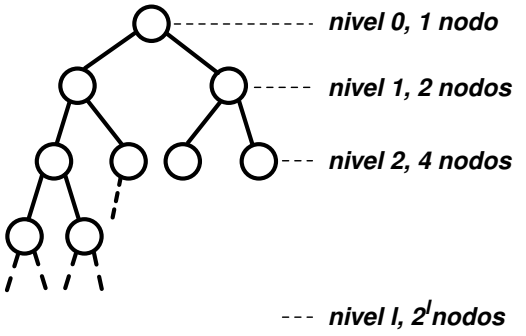

Figura 3.27: Cantidad máxima de nodos por nivel en un árbol binario lleno.

Un AOO no está limitado en cuanto a la cantidad de nodos que puede contener en un dado nivel. Por el contrario (ver figura 3.27) el árbol binario

puede tener a lo sumo dos nodos en el nivel 1, 4 nodos en el nivel 2, y en general, 2^l nodos en el nivel l. En total, un árbol binario de l niveles puede tener a lo sumo

$$n \leq 1 + 2 + 4... + 2^l \tag{3.17}$$

nodos. Pero ésta es una serie geométrica de razón dos, por lo cual

$$n \leq \frac{2^{l+1} - 1}{2 - 1} = 2^{l+1} - 1. \tag{3.18}$$

O bien

$$n < 2^{l+1}. \tag{3.19}$$

Inversamente el número de niveles puede obtenerse a partir del número de nodos como

$$\begin{aligned} l + 1 &> \log_2 n \\ l + 1 &> \text{floor}(\log_2 n) \\ l &\geq \text{floor}(\log_2 n) \end{aligned} \tag{3.20}$$

3.8.4. Operaciones básicas sobre árboles binarios

Las operaciones sobre AB difieren de las de AOO (ver sección §3.3.4) en las funciones que permiten *"moverse"* en el árbol. Si usáramos las operaciones de AOO para acceder a los hijos de un nodo en un AB, entonces para acceder al hijo derecho deberíamos hacer una operación *"hijo-más-izquierdo"* para acceder al hijo izquierdo y después una operación *"hermano-derecho"*. Pero el hijo izquierdo no necesariamente debe existir, por lo cual la estrategia de movimientos en el árbol debe ser cambiada. La solución es que existan dos operaciones independientes *"hijo-izquierdo"* e *"hijo-derecho"*. También, en AB *sólo se puede insertar en un nodo* Λ ya que la única posibilidad de insertar en un nodo dereferenciable, manteniendo el criterio usado para AOO, sería insertar en un hijo izquierdo que no tiene hermano derecho. En ese caso (manteniendo el criterio usado para AOO) el hijo izquierdo debería pasar a ser el derecho y el nuevo elemento pasaría a ser el hijo izquierdo y de todas formas esta operación no agregaría ninguna funcionalidad.

Entonces, las operaciones para el AB son las siguientes.

- Dado un nodo, obtener su *hijo izquierdo*. (Puede retornar una posición Λ).

- Dado un nodo, obtener su *hijo derecho*. (Puede retornar una posición Λ).
- Dada una posición, determinar si es Λ o no.
- Obtener la posición de la raíz del árbol.
- Dado un nodo obtener una referencia al dato contenido en el nodo.
- Dada una posición *no dereferenciable* y un dato, insertar un nuevo nodo con ese dato en esa posición.
- Borrar un nodo y todo su subárbol correspondiente.

Notar que sólo cambian las dos primeras y la inserción con respecto a las de AOO.

3.8.5. Interfaces e implementaciones

3.8.5.1. Interfaz básica

```
1.  class iterator_t {
2.    /* . . . */
3.   public:
4.    iterator_t left();
5.    iterator_t right();
6.  };
7.
8.  class btree {
9.    /* . . . */
10.  public:
11.   iterator_t begin();
12.   iterator_t end();
13.   elem_t & retrieve(iterator_t p);
14.   iterator_t insert(iterator_t p,elem_t t);
15.   iterator_t erase(iterator_t p);
16.   void clear();
17.   iterator_t splice(iterator_t to,iterator_t from);
18. };
```

Código 3.12: *Interfaz básica para árboles binarios. [Archivo: btreebash.h]*

En el código 3.12 vemos una interfaz posible (recordemos que las STL *no* tienen clases de árboles) para AB. Como siempre, la llamamos básica porque no tiene templates, clases anidadas ni sobrecarga de operadores. Es

similar a la mostrada para AOO en código 3.7, la única diferencia es que en la clase **iterator** las funciones **left()** y **right()** retornan los *hijos izquierdo y derecho*, respectivamente, en lugar de las funciones **lchild()** (que en AOO retornaba el hijo más izquierdo) y **right()** (que en AOO retorna el *hermano* derecho).

3.8.5.2. Ejemplo de uso. Predicados de igualdad y espejo

```
1.  bool equal_p (btree &T,iterator_t nt,
2.            btree &Q,iterator_t nq) {
3.    if (nt==T.end() xor nq==Q.end()) return false;
4.    if (nt==T.end()) return true;
5.    if (T.retrieve(nt) != Q.retrieve(nq)) return false;
6.    return equal_p(T,nt.right(),Q,nq.right()) &&
7.      equal_p(T,nt.left(),Q,nq.left());
8.  }
9.  bool equal_p(btree &T,btree &Q) {
10.   return equal_p(T,T.begin(),Q,Q.begin());
11. }
```

Código 3.13: *Predicado que determina si dos árboles son iguales. [Archivo: equalp.cpp]*

Como ejemplo de uso de esta interfaz vemos en código 3.13 una función predicado (es decir una función que retorna un valor booleano) que determina si dos árboles binarios **T** y **Q** son iguales. Dos árboles son iguales si

- Ambos son vacíos
- Ambos no son vacíos, los valores de sus nodos son iguales y los hijos respectivos de su nodo raíz son iguales.

Como, descripto en §3.4.2 la función se basa en una función auxiliar recursiva que toma como argumento adicionales dos nodos **nt** y **nq** y determina si los subárboles de **nt** y **nq** son iguales entre sí.

La función recursiva primero determina si uno de los nodos es Λ y el otro no o viceversa. En ese caso la función debe retornar **false** inmediatamente. La expresión lógica buscada podría ser

[copy]

```
1.    if ((nt==T.end() && nq=!Q.end()) ||
```

```
2.      (nt!=T.end() && nq==Q.end())) return false;
```

pero la expresión se puede escribir en la forma más compacta usada en la línea 3 usando el operador lógico **xor** (*"o exclusivo"*). Recordemos que **x xor y** retorna verdadero sólo si uno de los operandos es verdadero y el otro falso. Si el código llega a la línea línea 4 es porque o bien ambos nodos son Λ o bien los dos no lo son. Por lo tanto, si **nt** es Λ entonces ambos lo son y la función puede retornar **true** ya que dos árboles vacíos ciertamente son iguales. Ahora, si el código llega a la línea 5 es porque ambos nodos no son Λ. En ese caso, los valores contenidos deben ser iguales. Por lo tanto la línea 5 retorna **false** si los valores son distintos. Finalmente, si el código llega a la línea línea 6 sólo resta comparar los subárboles derechos de **nt** y **nq** y sus subárboles izquierdos, los cuales deben ser iguales entre sí. Por supuesto estas comparaciones se hacen en forma recursiva.

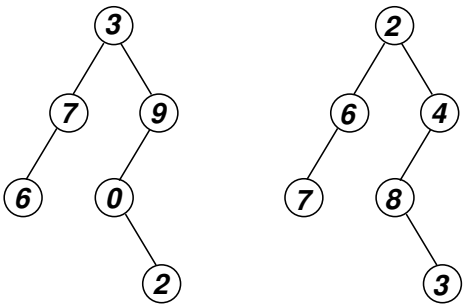

Figura 3.28: Dos árboles semejantes.

```
1.  bool semejante_p (btree &T,iterator_t nt,
2.            btree &Q,iterator_t nq) {
3.    if (nt==T.end() xor nq==Q.end()) return false;
4.    if (nt==T.end()) return true;
5.    return semejante_p(T,nt.right(),Q,nq.right()) &&
6.      semejante_p(T,nt.left(),Q,nq.left());
7.  }
8.  bool semejante_p(btree &T,btree &Q) {
9.    return semejante_p(T,T.begin(),Q,Q.begin());
10. }
```

Código 3.14: *Función predicado que determina si dos árboles son semejantes. [Archivo: semejantep.cpp]*

Modificando ligeramente este algoritmo verifica si dos árboles son *"semejantes"* es decir, son iguales en cuanto a su estructura, sin tener en cuenta el valor de los nodos. Por ejemplo, los árboles de la figura 3.28 son semejantes entre sí. En forma recursiva la semejanza se puede definir en forma casi igual que la igualdad pero no hace falta que las raíces de los árboles sea igual. Dos árboles son semejantes si

- Ambos son vacíos
- Ambos no son vacíos, y los hijos respectivos de su nodo raíz son semejantes.

Notar que la única diferencia es que no se comparan los valores de las raíces de los subárboles comparados. En el código 3.14 se muestra una función predicado que determina si dos árboles son semejantes. El código es igual al de **equal_p** sólo que se elimina la línea 5.

3.8.5.3. Ejemplo de uso. Hacer espejo "in place"

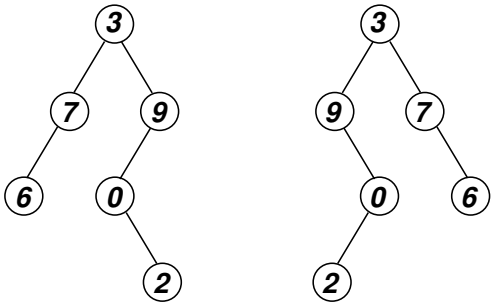

Figura 3.29: Copia espejo del árbol.

```
1.  void mirror(btree &T,iterator_t n) {
2.    if (n==T.end()) return;
3.    else {
4.      btree tmp;
5.      tmp.splice(tmp.begin(),n.left());
6.      T.splice(n.left(),n.right());
7.      T.splice(n.right(),tmp.begin());
8.      mirror(T,n.right());
9.      mirror(T,n.left());
10.   }
```

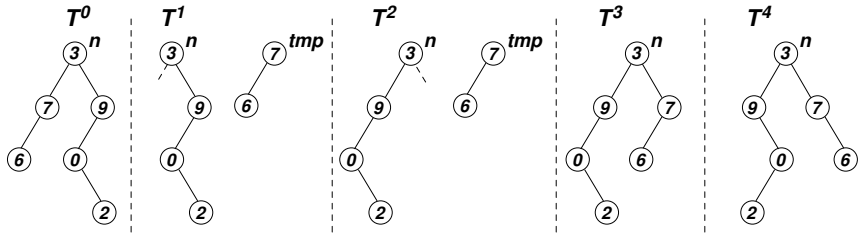

Figura 3.30: Descripción gráfica del procedimiento para copiar convertir *"in place"* un árbol en su espejo.

```
11. }
12. void mirror(btree &T) { mirror(T,T.begin()); }
```

Código 3.15: *Función para copiar convertir "in place" un árbol en su espejo. [Archivo: btmirror.cpp]*

Consideremos ahora una función **`void mirror(tree &T)`** que modifica el árbol **`T`**, dejándolo hecho igual a su espejo. Notar que esta operación es *"in place"*, es decir se hace en la estructura misma, sin crear una copia. El algoritmo es recursivo y se basa en intercambiar los subárboles de los hijos del nodo **`n`** y después aplicar recursivamente la función a los hijos (ver código 3.15). La operación del algoritmo sobre un nodo **`n`** del árbol se puede ver en la figura 3.30.

- La línea 5 extrae todo el subárbol del nodo izquierdo y lo inserta en un árbol vacío **`tmp`** con la operación **`splice(to,from)`**. Después de hacer esta operación el árbol se muestra como en el cuadro T^1.
- La línea 6 mueve todo el subárbol del hijo derecho al hijo izquierdo, quedando como en T^2.
- La línea 7 mueve todo el árbol guardado en **`tmp`** y que originariamente estaba en el hijo izquierdo al hijo derecho, quedando como en T^3.
- Finalmente en las líneas 8–9 la función se aplica recursivamente a los hijos derecho e izquierdo, de manera que el árbol queda como en T^4, es decir como el espejo del árbol original T^0.

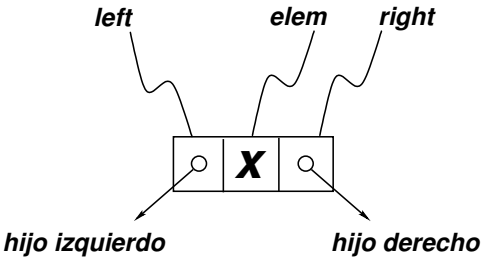

Figura 3.31: Celdas para representación de árboles binarios.

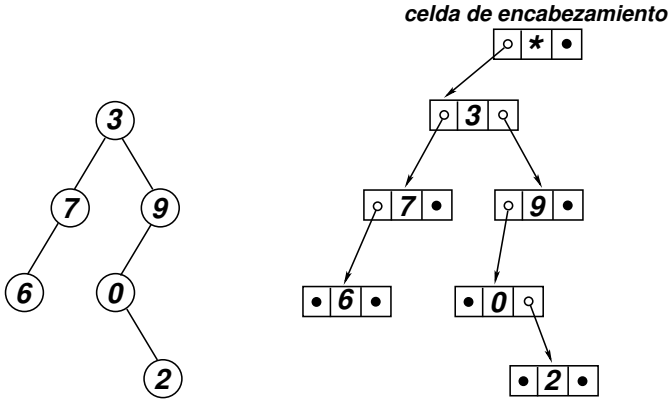

Figura 3.32: Representación de un árbol binario con celdas enlazadas.

3.8.5.4. Implementación con celdas enlazadas por punteros

Así como la diferencia entre las interfaces de AOO y AB difieren en las funciones `lchild()` y `right()` que son reemplazadas por `left()` y `right()`. (Recordar que `right()` tiene significado diferente en AOO y AB.) Esto induce naturalmente a considerar que en la celda haya dos punteros que apunten al hijo izquierdo y al hijo derecho, como se muestra en la figura 3.31. En la figura 3.32 se observa a la derecha el enlace de las celdas para representar el árbol binario de la izquierda.

```
1.  typedef int elem_t;
2.  class cell;
3.  class iterator_t;
4.
5.  class cell {
```

```
6.    friend class btree;
7.    friend class iterator_t;
8.    elem_t t;
9.    cell *right,*left;
10.   cell() : right(NULL), left(NULL) {}
11.  };
12.
13.  class iterator_t {
14.  private:
15.   friend class btree;
16.   cell *ptr,*father;
17.   enum side_t {NONE,R,L};
18.   side_t side;
19.   iterator_t(cell *p,side_t side_a,cell *f_a)
20.    : ptr(p), side(side_a), father(f_a) { }
21.
22.  public:
23.   iterator_t(const iterator_t &q) {
24.    ptr = q.ptr;
25.    side = q.side;
26.    father = q.father;
27.   }
28.   bool operator!=(iterator_t q) { return ptr!=q.ptr; }
29.   bool operator==(iterator_t q) { return ptr==q.ptr; }
30.   iterator_t() : ptr(NULL), side(NONE),
31.
32.              father(NULL) { }
33.
34.   iterator_t left() {
35.    return iterator_t(ptr->left,L,ptr);
36.   }
37.   iterator_t right() {
38.    return iterator_t(ptr->right,R,ptr);
39.   }
40.  };
41.
42.  class btree {
43.  private:
44.   cell *header;
45.   iterator_t tree_copy_aux(iterator_t nq,
46.                    btree &TT,iterator_t nt) {
47.    nq = insert(nq,TT.retrieve(nt));
48.    iterator_t m = nt.left();
49.    if (m != TT.end()) tree_copy_aux(nq.left(),TT,m);
50.    m = nt.right();
51.    if (m != TT.end()) tree_copy_aux(nq.right(),TT,m);
```

```
52.       return nq;
53.     }
54.   public:
55.     static int cell_count_m;
56.     static int cell_count() { return cell_count_m; }
57.     btree() {
58.       header = new cell;
59.       cell_count_m++;
60.       header->right = NULL;
61.       header->left = NULL;
62.     }
63.     btree(const btree &TT) {
64.       if (&TT != this) {
65.         header = new cell;
66.         cell_count_m++;
67.         header->right = NULL;
68.         header->left = NULL;
69.         btree &TTT = (btree &) TT;
70.         if (TTT.begin()!=TTT.end())
71.           tree_copy_aux(begin(),TTT,TTT.begin());
72.       }
73.     }
74.     ~btree() { clear(); delete header; cell_count_m--; }
75.     elem_t & retrieve(iterator_t p) { return p.ptr->t; }
76.     iterator_t insert(iterator_t p,elem_t t) {
77.       cell *c = new cell;
78.       cell_count_m++;
79.       c->t = t;
80.       if (p.side == iterator_t::R)
81.         p.father->right = c;
82.       else p.father->left = c;
83.       p.ptr = c;
84.       return p;
85.     }
86.     iterator_t erase(iterator_t p) {
87.       if(p==end()) return p;
88.       erase(p.right());
89.       erase(p.left());
90.       if (p.side == iterator_t::R)
91.         p.father->right = NULL;
92.       else p.father->left = NULL;
93.       delete p.ptr;
94.       cell_count_m--;
95.       p.ptr = NULL;
96.       return p;
97.     }
```

```
98.
99.     iterator_t splice(iterator_t to,iterator_t from) {
100.      cell *c = from.ptr;
101.      from.ptr = NULL;
102.      if (from.side == iterator_t::R)
103.        from.father->right = NULL;
104.      else
105.        from.father->left = NULL;
106.      if (to.side == iterator_t::R) to.father->right = c;
107.      else to.father->left = c;
108.      to.ptr = c;
109.      return to;
110.    }
111.    iterator_t find(elem_t t) { return find(t,begin()); }
112.    iterator_t find(elem_t t,iterator_t p) {
113.      if(p==end() || p.ptr->t == t) return p;
114.      iterator_t l = find(t,p.left());
115.      if (l!=end()) return l;
116.      iterator_t r = find(t,p.right());
117.      if (r!=end()) return r;
118.      return end();
119.    }
120.    void clear() { erase(begin()); }
121.    iterator_t begin() {
122.      return iterator_t(header->left,
123.                        iterator_t::L,header);
124.    }
125.    iterator_t end() { return iterator_t(); }
126.
127.    void lisp_print(iterator_t n) {
128.      if (n==end()) { cout << "."; return; }
129.      iterator_t r = n.right(), l = n.left();
130.      bool is_leaf = r==end() && l==end();
131.      if (is_leaf) cout << retrieve(n);
132.      else {
133.        cout << "(" << retrieve(n) << " ";
134.        lisp_print(l);
135.        cout << " ";
136.        lisp_print(r);
137.        cout << ")";
138.      }
139.    }
140.    void lisp_print() { lisp_print(begin()); }
141.  };
```

Código 3.16: *Implementación de árboles con celdas enlazadas por punteros. Declaraciones. [Archivo: btreebas.h]*

En el código 3.16 se muestra la implementación correspondiente.

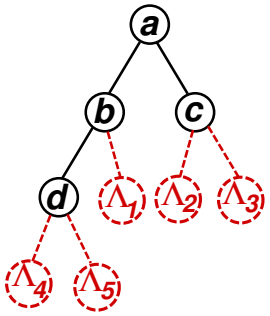

Figura 3.33: Iterators Λ en un árbol binario.

- **La clase iterator.** La clase **iterator_t** contiene un puntero a la celda, y otro al padre, como en el caso del AOO. Sin embargo el puntero **prev** que apuntaba al hermano a la izquierda, aquí ya no tiene sentido. Recordemos que el iterator nos debe permitir ubicar a las posiciones, incluso aquellas que son Λ. Para ello incluimos el iterator un miembro **side** de tipo tipo **enum side_t**, que puede tomar los valores **R** (right) y **L** (left). Por ejemplo consideremos el árbol de la figura 3.33. Además de los nodos $a-d$ existen 5 nodos Λ. Las posiciones de algunos nodos son representadas como sigue

  - nodo b: **ptr**=b, **father**=a, **side=L**
  - nodo c: **ptr**=c, **father**=a, **side=R**
  - nodo d: **ptr**=d, **father**=b, **side=L**
  - nodo Λ_1: **ptr=NULL**, **father**=b, **side=R**
  - nodo Λ_2: **ptr=NULL**, **father**=c, **side=L**
  - nodo Λ_3: **ptr=NULL**, **father**=c, **side=R**
  - nodo Λ_4: **ptr=NULL**, **father**=d, **side=L**
  - nodo Λ_5: **ptr=NULL**, **father**=d, **side=R**

- La comparación de iterators (líneas 28–29) compara sólo los campos **ptr**, de manera que todos los iterators Λ resultan iguales entre sí (ya

que tienen **ptr=NULL**). Como **end()** retorna un iterator Λ (ver más abajo), entonces esto habilita a usar los lazos típicos

[copy]

```
1.  while (c!=T.end()) {
2.    // . . . .
3.    c = c.right();
4.  }
```

- La clase **btree** incluye un contador de celdas **cell_count()** y constructor por copia **btree(const btree &)**, como para AOO.

- La diferencia principal está en **insert(p,x)** y **erase(p)**. En insert se crea la celda (actualizando el contador de celdas) y se inserta el dato (líneas 77–79. Recordar que los campos punteros de la celda quedan en **NULL**, porque así se inicializan en el constructor de celdas. El único campo de celdas que se debe actualizar es, o bien el campo **left** o **right** de la celda padre. Cuál de ellos es el que debe apuntar a la nueva celda se deduce de **p.side** en el iterator. Finalmente se debe actualizar el iterator de forma que **ptr** apunte a la celda creada.

- **erase(p)** elimina primero recursivamente todo el subárbol de los hijos izquierdo y derecho de **p**. Después libera la celda actualizando el campo correspondiente del padre (dependiendo de **p.side**). También se actualiza el contador **cell_count_m** al liberar la celda. Notar la actualización del contador por la liberación de las celdas en los subárboles de los hijos se hace automáticamente dentro de la llamada recursiva, de manera que en **erase(p)** sólo hay que liberar explícitamente a la celda **p.ptr**.

- El código de **splice(to,from)** es prácticamente un erase de **from** seguido de un **insert** en **to**.

- La posición raíz del árbol se elige como el hijo izquierdo de la celda de encabezamiento. Esto es una convención, podríamos haber elegido también el hijo derecho.

- El constructor por defecto de la clase **iterator** retorna un iterator no dereferenciable que no existe en el árbol. Todos sus punteros son nulos y **side** es un valor especial de **side_t** llamado **NONE**. Insertar en este iterator es un error.

- **end()** retorna un iterator no dereferenciable dado por el constructor por defecto de la clase **iterator_t** (descripto previamente). Este iterator debería ser usado sólo para comparar. Insertar en este iterator es un error.

3.8.5.5. Interfaz avanzada

```
1.  template<class T>
2.  class btree {
3.    /* . . . */
4.   public:
5.    class iterator {
6.    /* . . . */
7.    public:
8.      T &operator*();
9.      T *operator->();
10.     bool operator!=(iterator q);
11.     bool operator==(iterator q);
12.     iterator left();
13.     iterator right();
14.   };
15.   iterator begin();
16.   iterator end();
17.   iterator insert(iterator p,T t);
18.   iterator erase(iterator p);
19.   iterator splice(iterator to,iterator from);
20.   void clear();
21. };
```

Código 3.17: *Interfaz avanzada para árboles binarios. [Archivo: btreeh.h]*

En el código 3.17 vemos una interfaz para árboles binarios incluyendo templates, clases anidadas y sobrecarga de operadores. Las diferencias principales son (ver también lo explicado en la sección §3.6)

- La clase es un template sobre el tipo contenido en el dato (**class T**) de manera que podremos declarar **btree<int>**, **btree<double>** ...
- La dereferenciación de nodo se hace sobrecargando los operadores ***** y **->**, de manera que podemos hacer

[copy]

```
1.  x = *n;
```

```
2.  *n = w;
3.  y = n->member;
4.  n->member = v;
```

donde `n` es de tipo iterator, `x,w` con de tipo `T` y `member` es algún campo de la clase `T` (si es una clase compuesta). También es válido hacer `n->f(...)` si `f` es un método de la clase `T`.

3.8.5.6. Ejemplo de uso. El algoritmo apply y principios de programación funcional.

A esta altura nos sería fácil escribir algoritmos que modifican los valores de un árbol, por ejemplo sumarle a todos los valores contenidos en un árbol un valor, o duplicarlos. Todos estos son casos particulares de un algoritmo más general `apply(Q,f)` que tiene como argumentos un árbol `Q` y una *"función escalar"* `T f(T)`. y le aplica a cada uno de los valores nodales la función en cuestión. Este es un ejemplo de *"programación funcional"*, es decir, programación en los cuales los datos de los algoritmos pueden ser también funciones.

C++ tiene un soporte básico para la programación funcional en la cual se pueden pasar *"punteros a funciones"*. Un soporte más avanzado se obtiene usando clases especiales que sobrecargan el operador `()`, a tales funciones se les llama *"functors"*. Nosotros vamos a escribir ahora una herramienta simple llamada `apply(Q,f)` que aplica a los nodos de un árbol una función escalar `t f(T)` pasada por puntero.

```
1.  template<class T>
2.  void apply(btree<T> &Q,
3.         typename btree<T>::iterator n,
4.         T(*f)(T)) {
5.    if (n==Q.end()) return;
6.    *n = f(*n);
7.    apply(Q,n.left(),f);
8.    apply(Q,n.right(),f);
9.  }
10. template<class T>
11. void apply(btree<T> &Q,T(*f)(T)) {
12.   apply(Q,Q.begin(),f);
```

Código 3.18: *Herramienta de programación funcional que aplica a los nodos de un árbol una función escalar. [Archivo: apply.cpp]*

La función se muestra en el código 3.18. Recordemos que para pasar funciones como argumentos, en realidad se pasa *el puntero a la función*. Como las funciones a pasar son funciones que toman como un argumento un elemento de tipo `T` y retornan un elemento del mismo tipo, su *"signatura"* (la forma como se declara) es `T f(T)`. La declaración de punteros a tales funciones se hace reemplazando en la signatura el nombre de la función por `(*f)`. De ahí la declaración en la línea 11, donde el segundo argumento de la función es de tipo `T(*f)(T)`.

Por supuesto `apply` tiene una estructura recursiva y llama a su vez a una función auxiliar recursiva que toma un argumento adicional de tipo iterator. Dentro de esta función auxiliar el puntero a función `f` se aplica como una función normal, como se muestra en la línea 6.

Si `n` es Λ la función simplemente retorna. Si no lo está, entonces aplica la función al valor almacenado en `n` y después llama `apply` recursivamente a sus hijos izquierdo y derecho.

Otro ejemplo de programación funcional podría ser una función `reduce(Q,g)` que toma como argumentos un árbol `Q` y una función asociativa `T g(T,T)` (por ejemplo la suma, el producto, el máximo o el mínimo) y devuelve el resultado de aplicar la función asociativa a todos los valores nodales, hasta llegar a un único valor. Por ejemplo, si hacemos que `g(x,y)` retorne `x+y` retornará la suma de todas las etiquetas del árbol y si hacemos que retorne el máximo, entonces retornará el máximo de todas las etiquetas del árbol. Otra aplicación pueden ser *"filtros"*, como la función `prune_odd` discutida en la sección §3.3.3. Podríamos escribir una función `remove_if(Q,pred)` que tiene como argumentos un árbol `Q` y una función predicado `bool pred(T)`. La función `remove_if` elimina todos los nodos `n` (y sus subárboles) para cuyos valores la función `pred(*n)` retorna verdadero. La función `prune_odd` se podría obtener entonces simplemente pasando a `remove_if` una función predicado que retorna verdadero si el argumento es impar.

3.8.5.7. Implementación de la interfaz avanzada

```
1. #ifndef AED_BTREE_H
2. #define AED_BTREE_H
3.
4. #include <iostream>
5. #include <cstddef>
6. #include <cstdlib>
```

```
7.  #include <cassert>
8.  #include <list>
9.
10. using namespace std;
11.
12. namespace aed {
13.
14.  //---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>---:
15.  template<class T>
16.  class btree {
17.  public:
18.    class iterator;
19.  private:
20.    class cell {
21.      friend class btree;
22.      friend class iterator;
23.      T t;
24.      cell *right,*left;
25.      cell() : right(NULL), left(NULL) {}
26.    };
27.    cell *header;
28.    enum side_t {NONE,R,L};
29.  public:
30.    static int cell_count_m;
31.    static int cell_count() { return cell_count_m; }
32.    class iterator {
33.    private:
34.      friend class btree;
35.      cell *ptr,*father;
36.      side_t side;
37.      iterator(cell *p,side_t side_a,cell *f_a)
38.        : ptr(p), side(side_a), father(f_a) { }
39.    public:
40.      iterator(const iterator &q) {
41.        ptr = q.ptr;
42.        side = q.side;
43.        father = q.father;
44.      }
45.      T &operator*() { return ptr->t; }
46.      T *operator->() { return &ptr->t; }
47.      bool operator!=(iterator q) { return ptr!=q.ptr; }
48.      bool operator==(iterator q) { return ptr==q.ptr; }
49.      iterator() : ptr(NULL), side(NONE), father(NULL) { }
50.
51.      iterator left() { return iterator(ptr->left,L,ptr); }
52.      iterator right() { return iterator(ptr->right,R,ptr); }
53.
```

```
54.     };
55.
56.     btree() {
57.       header = new cell;
58.       cell_count_m++;
59.       header->right = NULL;
60.       header->left = NULL;
61.     }
62.     btree<T>(const btree<T> &TT) {
63.       if (&TT != this) {
64.         header = new cell;
65.         cell_count_m++;
66.         header->right = NULL;
67.         header->left = NULL;
68.         btree<T> &TTT = (btree<T> &) TT;
69.         if (TTT.begin()!=TTT.end())
70.           copy(begin(),TTT,TTT.begin());
71.       }
72.     }
73.     btree &operator=(btree<T> &TT) {
74.       if (this != &TT) {
75.         clear();
76.         copy(begin(),TT,TT.begin());
77.       }
78.       return *this;
79.     }
80.     ~btree() { clear(); delete header; cell_count_m--; }
81.     iterator insert(iterator p,T t) {
82.       assert(p==end());
83.       cell *c = new cell;
84.       cell_count_m++;
85.       c->t = t;
86.       if (p.side==R) p.father->right = c;
87.       else p.father->left = c;
88.       p.ptr = c;
89.       return p;
90.     }
91.     iterator erase(iterator p) {
92.       if(p==end()) return p;
93.       erase(p.right());
94.       erase(p.left());
95.       if (p.side==R) p.father->right = NULL;
96.       else p.father->left = NULL;
97.       delete p.ptr;
98.       cell_count_m--;
99.       p.ptr = NULL;
100.      return p;
```

```
101.     }
102.
103.     iterator splice(iterator to,iterator from) {
104.       if (from==end()) return to;
105.       cell *c = from.ptr;
106.       from.ptr = NULL;
107.       if (from.side==R) from.father->right = NULL;
108.       else from.father->left = NULL;
109.
110.       if (to.side==R) to.father->right = c;
111.       else to.father->left = c;
112.       to.ptr = c;
113.       return to;
114.     }
115.     iterator copy(iterator nq,btree<T> &TT,iterator nt) {
116.       nq = insert(nq,*nt);
117.       iterator m = nt.left();
118.       if (m != TT.end()) copy(nq.left(),TT,m);
119.       m = nt.right();
120.       if (m != TT.end()) copy(nq.right(),TT,m);
121.       return nq;
122.     }
123.     iterator find(T t) { return find(t,begin()); }
124.     iterator find(T t,iterator p) {
125.       if(p==end() || p.ptr->t == t) return p;
126.       iterator l = find(t,p.left());
127.       if (l!=end()) return l;
128.       iterator r = find(t,p.right());
129.       if (r!=end()) return r;
130.       return end();
131.     }
132.     void clear() { erase(begin()); }
133.     iterator begin() { return iterator(header->left,L,header); }
134.
135.     void lisp_print(iterator n) {
136.       if (n==end()) { cout << "."; return; }
137.       iterator r = n.right(), l = n.left();
138.       bool is_leaf = r==end() && l==end();
139.       if (is_leaf) cout << *n;
140.       else {
141.         cout << "(" << *n << " ";
142.         lisp_print(l);
143.         cout << " ";
144.         lisp_print(r);
145.         cout << ")";
146.       }
147.     }
```

```
148.     void lisp_print() { lisp_print(begin()); }
149.
150.     iterator end() { return iterator(); }
151.   };
152.
153.   template<class T>
154.   int btree<T>::cell_count_m = 0;
155. }
156. #endif
```

Código 3.19: *Implementación de la interfaz avanzada de árboles binarios por punteros. [Archivo: btree.h]*

En el código 3.19 vemos una posible implementación de interfaz avanzada de AB con celdas enlazadas por punteros.

3.8.6. Arboles de Huffman

Los árboles de Huffman son un ejemplo interesante de utilización del TAD AB. El objetivo es comprimir archivos o mensajes de texto. Por simplicidad, supongamos que tenemos una cadena de N caracteres compuesta de un cierto conjunto reducido de caracteres C. Por ejemplo si consideramos las letras $C = \{a, b, c, d\}$ entonces el mensaje podría ser $abdcdacabbcdba$. El objetivo es encontrar un representación del mensaje en bits (es decir una cadena de 0's y 1's) lo más corta posible. A esta cadena de 0's y 1's la llamaremos *"el mensaje encodado"*. El algoritmo debe permitir recuperar el mensaje original, ya que de esta forma, si la cadena de caracteres representa un archivo, entonces podemos guardar el mensaje encodado (que es más corto) con el consecuente ahorro de espacio.

Una primera posibilidad es el código provisto por la representación binaria ASCII de los caracteres. De esta forma cada caracter se encoda en un código de 8 bits. Si el mensaje tiene N caracteres, entonces el mensaje encodado tendrá una longitud de $l = 8N$ bits, resultando en una longitud promedio de

$$\langle l \rangle = \frac{l}{N} = 8 \text{ bits/caracter} \tag{3.21}$$

Pero como sabemos que el mensaje sólo está compuesto de las cuatro letras a, b, c, d podemos crear un código de dos bits como el $C1$ en la Tabla 3.2, de manera que un mensaje como $abcdcba$ se encoda en 00011011100100.