Capítulo 1

Diseño e implementación de TADs arborescentes[1]

Los ordenadores son buenos siguiendo instrucciones, no leyendo tu mente

Donald Knuth

Resumen: En este tema se presentan los TADs basados en árboles, prestando especial atención a los árboles binarios y a los árboles de búsqueda. Además, se introducen distintos tipos de recorridos usando tanto listas como iteradores.

1. Motivación

Podemos ver un texto como una secuencia (lista) de palabras, donde cada una de ellas es un `string`. El problema de las *concordancias* consiste en contar el número de veces que aparece cada palabra en ese texto.

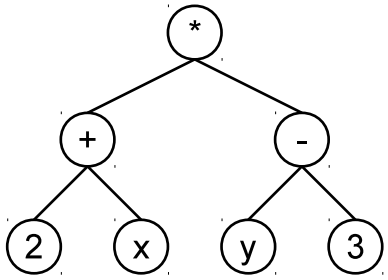
Implementar una función que reciba un texto como lista de las palabras (en forma de cadena) y escriba una línea por cada palabra distinta del texto donde se indique la palabra y el número de veces que aparece. La lista debe aparecer ordenada alfabéticamente.

2. Introducción

En el capítulo anterior estudiamos distintos TADs lineales para representar datos organizados de manera secuencial. En este capítulo usaremos árboles para representar de manera intuitiva datos organizados en jerarquías. Este tipo de estructuras jerárquicas surge de manera natural dentro y fuera de la Informática:

- Árboles genealógicos.
- Organización de un libro en capítulos, secciones, etc.
- Estructura de directorios y archivos de un sistema operativo.
- Árboles de análisis de expresiones aritméticas.

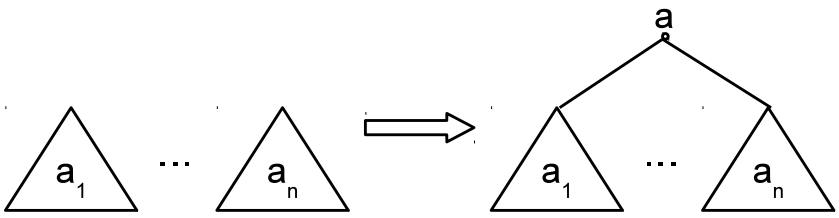
[1] Marco Antonio Gómez y Antonio Sánchez son los autores principales de este tema.



2.1. Modelo matemático

Desde un punto de vista matemático, los árboles son estructuras jerárquicas formadas por *nodos*, que se construyen de manera inductiva:

- Un solo nodo es un árbol $\mathtt{a}$. El nodo es la *raiz* del árbol.
- Dados n árboles $\mathtt{a_1}, \ldots, \mathtt{a_n}$, podemos construir un nuevo árbol $\mathtt{a}$ añadiendo un nuevo nodo como raíz y conectándolo con las raíces de los árboles $\mathtt{a_i}$. Se dice que los $\mathtt{a_i}$ son *subárboles* de $\mathtt{a}$.



Para identificar los distintos nodos de un árbol, vamos a usar una función que asigna a cada posición una cadena de números naturales con el siguiente criterio:

- La raíz del árbol tiene como posición la *cadena vacía* ϵ.
- Si un cierto nodo tiene como posición la cadena $\alpha \in \aleph^*$, el hijo i-ésimo de ese nodo tendrá como posición la cadena $\alpha.i$.

Por ejemplo, la figura 1 muestra un árbol y las cadenas que identifican las posiciones de sus nodos.

Un árbol puede describirse como una aplicación $\mathtt{a} : N \to V$ donde $N \subseteq \aleph^*$ es el conjunto de posiciones de los nodos, y V es el conjunto de valores posibles asociados a los nodos. Podemos describir el árbol de la figura 1 de la siguiente manera:

$$\begin{aligned} N &= \{\epsilon, 1, 2, 3, 1.1, 1.2, 3.1, 3.2, 3.3, 3.4, 3.3.1, 3.3.2\} \\ V &= \{A, B, C, D, E, F, G\} \end{aligned}$$

$$\begin{array}{llll} \mathtt{a}(\epsilon) = A & & & \\ \mathtt{a}(1) = B & \mathtt{a}(2) = A & \mathtt{a}(3) = D & \\ \mathtt{a}(1.1) = A & \mathtt{a}(1.2) = C & \mathtt{a}(3.1) = E & \text{etc.} \end{array}$$

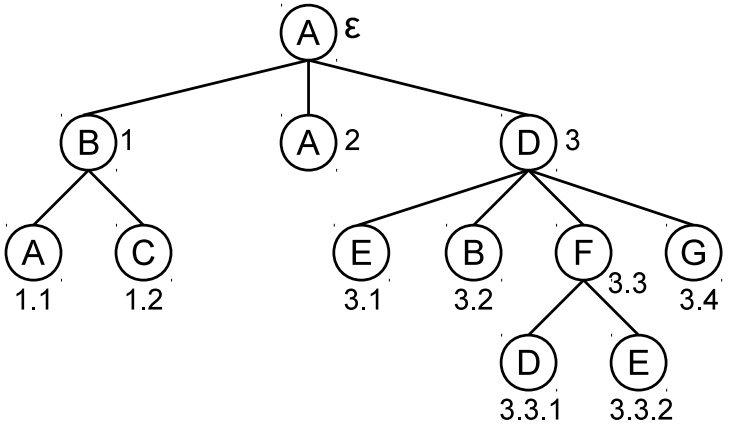


Figura 1: Posiciones y valores de cada nodo de un árbol.

2.2. Terminología

Antes de seguir adelante, debemos establecer un vocabulario común que nos permita describir árboles. Dado un árbol $\mathtt{a} : N \to V$

- Cada *nodo* es una tupla $(\alpha, \mathtt{a}(\alpha))$ que contiene la posición y el valor asociado al nodo. Distinguimos 3 tipos de nodos:
  - La *raíz* es el nodo de posición ϵ.
  - Las *hojas* son los nodos de posición α tales que no existe i tal que $\alpha.i \in N$
  - Los *nodos internos* son los nodos que no son hojas.
- Un nodo $\alpha.i$ tiene como *padre* a α, y se dice que es *hijo* de α.
- Dos nodos de posiciones $\alpha.i$ y $\alpha.j$ $(i \neq j)$ se llaman *hermanos*.
- Un *camino* es una sucesión de nodos $\alpha_1, \alpha_2, \ldots \alpha_n$ en la que cada nodo es padre del siguiente. El camino anterior tiene *longitud* n.
- Una *rama* es cualquier camino que empieza en la raíz y acaba en una hoja.
- El *nivel* o *profundidad* de un nodo es la longitud del camino que va desde la raíz hasta al nodo. En particular, el nivel de la raíz es 1.
- La *talla* o *altura* de un árbol es el máximo de los niveles de todos los nodos del árbol.
- El *grado* o *aridad* de un nodo interno es su número de hijos. La *aridad de un árbol* es el máximo de las aridades de todos sus nodos internos.
- Decimos que α es *antepasado* de β (resp. β es *descenciente* de α) si existe un camino desde α hasta β.
- Cada nodo de un árbol $\mathtt{a}$ determina un *subárbol* $\mathtt{a}_0$ con raíz en ese nodo.
- Dado un árbol $\mathtt{a}$, los subárboles de $\mathtt{a}$ (si existen) se llaman *árboles hijos* de $\mathtt{a}$.

2.3. Tipos de árboles

Distinguimos distintos tipos de árboles en función de sus características:

- Ordenados o no ordenados. Un árbol es ordenado si el orden de los hijos de cada nodo es relevante.
- Generales o n-ários. Un árbol es n-ário si el máximo numero de hijos de cualquier nodo es n. Un árbol es general si no existe una limitación fijada al número máximo de hijos de cada nodo.

3. Árboles binarios

Un árbol binario consiste en una estructura recursiva cuyos nodos tienen como mucho dos hijos, un hijo izquierdo y un hijo derecho. El TAD de los árboles binarios (lo llamaremos *Arbin*) tiene las siguientes operaciones:

- `ArbolVacio`: operación generadora que construye un árbol vacío (un árbol sin ningún nodo).
- `Cons(iz, elem, dr)`: segunda operación generadora que construye un árbol binario a partir de otros dos (el que será el hijo izquierdo y el hijo derecho) y de la información que se almacenará en la raíz.
- `raiz`: operación observadora que devuelve el elemento almacenado en la raíz del árbol. Es parcial pues falla si el árbol es vacío.
- `hijoIz`, `hijoDr`: dos operaciones observadoras (ambas parciales) que permiten obtener el hijo izquierdo y el hijo derecho de un árbol dado. Las operaciones no están definidas para árboles vacíos.
- `esVacio`: otra operación observadora para saber si un árbol tiene algún nodo o no.

Con sólo estas operaciones la utilidad del TAD está muy limitada. En apartados siguientes lo extenderemos con otras operaciones observadoras.

3.1. Implementación

Igual que ocurre con los TADs lineales, podemos implementar el TAD *Arbin* utilizando distintas estructuras en memoria. Sin embargo cuando la forma de los árboles no está restringida la única implementación factible es la que utiliza nodos.

La intuición detrás de la implementación es sencilla y sale directamente de las representaciones de los árboles que hemos utilizado en la sección anterior: cada nodo del árbol será representado como un nodo en memoria que contendrá tres atributos: el elemento almacenado y punteros al hijo izquierdo y al hijo derecho.

No obstante, *no* debe confundirse un elemento del TAD *Arbin* de la estructura en memoria utilizado para almacenarlo. Si bien existe una transformación directa entre uno y otro son conceptos distintos. Un árbol es un elemento del TAD construido utilizando las operaciones generadoras anteriores (y que, cuando lo programemos, será un objeto de la clase `Arbin`); los nodos en los que nos basamos forman una *estructura jerárquica de nodos* que tiene sentido únicamente *en la implementación*. Veremos en la implementación que hay métodos (privados o protegidos) que trabajan directamente con esta *estructura*

jerárquica a la que el usuario del TAD no tendrá acceso directo (en otro caso se rompería la barrera de abstracción impuesta por las operaciones del TAD).

A continuación aparece la definición de la clase `Nodo` que, como no podría ser de otra manera, es una clase interna a `Arbin`. La clase `Arbin` necesita únicamente almacenar un *puntero* a la raíz de la *estructura jerárquica de nodos* que representan el árbol[2].

```
template <class T>
class Arbin {
public:

   ...

protected:
   /**
    Clase nodo que almacena internamente el elemento (de tipo T),
    y los punteros al hijo izquierdo y al hijo derecho.
    */
   class Nodo {
   public:
      Nodo() : _iz(NULL), _dr(NULL) {}
      Nodo(const T &elem) : _elem(elem), _iz(NULL), _dr(NULL) {}
      Nodo(Nodo *iz, const T &elem, Nodo *dr) :
         _elem(elem), _iz(iz), _dr(dr) {}

      T _elem;
      Nodo *_iz;
      Nodo *_dr;
   };

   ...

private:
   /**
    Puntero a la raíz de la estructura jerárquica
    de nodos.
    */
   Nodo *_ra;
};
```

El *invariante de la representación* de esta implementación debe asegurarse de que:

- Todos los nodos contienen información válida y están ubicados correctamente en memoria.
- El subárbol izquierdo y el subárbol derecho no comparten nodos.
- No hay ciclos entre los nodos, o lo que es lo mismo, los nodos alcanzables desde al raíz no incluyen a la propia raíz.

Con estas ideas el invariante queda:

[2]Estas estructuras jerárquicas de nodos son útiles también para resolver otros problemas distintos a la implementación del TAD de los árboles binarios; ver por ejemplo el ejercicio 14.

$$R_{Arbin_T}(\mathsf{p}) \iff_{def} buenaJerarquia(\mathsf{p.\_ra})$$

donde

$$\begin{array}{lll}
buenaJerarquia(\mathsf{ptr}) = true & & \textbf{si}\ \ \mathsf{ptr} = \mathsf{null} \\
buenaJerarquia(\mathsf{ptr}) = \mathsf{ubicado(ptr)} \wedge R_T(\mathsf{ptr._elem}) \wedge & & \\
\quad nodos(\mathsf{ptr._iz}) \bigcap nodos(\mathsf{ptr._dr}) = \emptyset \wedge & & \\
\quad ptr \notin nodos(\mathsf{ptr._iz}) \wedge & & \\
\quad ptr \notin nodos(\mathsf{ptr._dr}) \wedge & & \\
\quad buenaJerarquia(\mathsf{ptr._iz}) \wedge & & \\
\quad buenaJerarquia(\mathsf{ptr._dr}) & & \textbf{si}\ \ \mathsf{ptr} \neq \mathsf{null}
\end{array}$$

$$\begin{array}{lll}
nodos(\mathsf{ptr}) = \emptyset & & \textbf{si}\ \ \mathsf{ptr} = \mathsf{null} \\
nodos(\mathsf{ptr}) = \{ptr\} \bigcup nodos(\mathsf{ptr._iz}) \bigcup nodos(\mathsf{ptr._dr}) & & \textbf{si}\ \ \mathsf{ptr} \neq \mathsf{null}
\end{array}$$

La definición de la relación de equivalencia que se utiliza para saber si dos árboles son o no iguales también sigue una definición recursiva:

$$\mathsf{p1} \equiv_{Arbin_T} \mathsf{p2} \iff_{def} iguales_T(\mathsf{p1.\_ra}, \mathsf{p2.\_ra})$$

$$\begin{array}{lll}
iguales(\mathsf{ptr1}, \mathsf{ptr2}) = \mathsf{true} & & \textbf{si}\ \ \mathsf{ptr1} = \mathsf{null} \wedge \mathsf{ptr2} = \mathsf{null} \\
iguales(\mathsf{ptr1}, \mathsf{ptr2}) = \mathsf{false} & & \textbf{si}\ \ (\mathsf{ptr1} = \mathsf{null} \wedge \mathsf{ptr2} \neq \mathsf{null}) \vee \\
 & & \quad (\mathsf{ptr1} \neq \mathsf{null} \wedge \mathsf{ptr2} = \mathsf{null}) \\
iguales(\mathsf{ptr1}, \mathsf{ptr2}) = \mathsf{ptr1._elem} \equiv_T \mathsf{ptr2._elem} \wedge & & \\
\quad iguales(\mathsf{ptr1._iz}, \mathsf{ptr2._iz}) \wedge & & \\
\quad iguales(\mathsf{ptr1._dr}, \mathsf{ptr2._dr}) & & \textbf{si}\ \ (\mathsf{ptr1} \neq \mathsf{null} \wedge \mathsf{ptr2} \neq \mathsf{null})
\end{array}$$

Antes de seguir con la implementación de las operaciones debemos crear algunos métodos que trabajan directamente con la *estructura jerárquica*, igual que en las implementaciones de los TADs lineales del tema anterior empezamos por las operaciones que trabajaban con las listas enlazadas y doblemente enlazadas. Dada la naturaleza recursiva de la estructura de nodos, todas esas operaciones serán recursivas; recibirán, al menos, un puntero a un nodo que debe verse como la raíz de la estructura de nodos, o al menos la raíz del "subárbol"[3] sobre el que debe operar.

Aunque pueda parecer empezar la casa por el tejado, comencemos con la operación que *libera* toda la memoria ocupada por la estructura jerárquica. El método recibe como parámetro el puntero al primer nodo y va eliminando de forma recursiva:

[3]La palabra "subárbol" la ponemos entre comillas para hacer notar que no estamos aquí haciendo referencia al concepto de subárbol como elemento del TAD sino más bien a la "subestructura" jerárquica de nodos.

```
static void libera(Nodo *ra) {
   if (ra != NULL) {
      libera(ra->_iz);
      libera(ra->_dr);
      delete ra;
   }
}
```

Algunas de las operaciones pueden trabajar no con una sino con *dos* estructuras jerárquicas. Por ejemplo el siguiente método compara dos estructuras, dados los punteros a sus raices:

```
static bool comparaAux(Nodo *r1, Nodo *r2) {
   if (r1 == r2)
      return true;
   else if ((r1 == NULL) || (r2 == NULL))
      // En el if anterior nos aseguramos de
      // que r1 != r2. Si uno es NULL, el
      // otro entonces no lo será, luego
      // son distintos.
      return false;
   else {
      return (r1->_elem == r2->_elem) &&
         comparaAux(r1->_iz, r2->_iz) &&
         comparaAux(r1->_dr, r2->_dr);
   }
}
```

Como veremos más adelante, este método estático nos resultará muy útil para implementar el operador de igualdad del TAD Arbin.

Por último, algunas de las operaciones pueden devolver otra información. Por ejemplo, el siguiente método hace una *copia* de una estructura jerárquica y devuelve el puntero a la raíz de la copia. Igual que todas las anteriores la implementación es recursiva. Como es lógico, la estructura recién creada deberá ser eliminada (utilizando el `libera` implementado anteriormente) posteriormente:

```
static Nodo *copiaAux(Nodo *ra) {
   if (ra == NULL)
      return NULL;

   return new Nodo(copiaAux(ra->_iz),
               ra->_elem,
               copiaAux(ra->_dr));
}
```

Tras esto, estamos en disposición de implementar las operaciones públicas del TAD. No obstante, antes de abordarla expliquemos una decisión de diseño relevante. Hay una operación generadora (que implementaremos como constructor) que recibe dos árboles ya construidos:

```
Arbin(const Arbin &iz, const T &elem, const Arbin &dr);
```

Una implementación ingenua crearía un nuevo nodo y "cosería" los punteros haciendo que el hijo izquierdo del nuevo nodo fuera la raíz de `iz` y el derecho la raíz de `dr`:

```
// IMPLEMENTACIÓN NO VALIDA
Arbin::Arbin(const Arbin &iz, const T &elem, const Arbin &dr) {
   _ra = new Nodo(iz._ra, elem, dr._ra);
}
```

Esta implementación, no obstante, no es válida porque tendríamos una poco deseable compartición de memoria: la estructura jerárquica de los nodos de `iz` y de `dr` formarían también parte de la estructura jerárquica del nodo recién construido. Eso tiene como consecuencia inmediata que al destruir `iz` se destruiría automáticamente los nodos del árbol más grande.

Para solucionar el problema existen dos alternativas (similares a las que se tienen cuando implementamos la operación de concatenación de las listas, ver ejercicio **??** del tema anterior)[4]:

- Hacer una copia de las estructuras jerárquicas de nodos de `iz` y `dr`.
- Utilizar esas estructuras jerárquicas para el nuevo árbol y *vaciar* `iz` y `dr` de forma que la llamada al constructor los *vacíe*[5].

En nuestra implementación nos decantaremos por la primera opción[6]:

```
Arbin(const Arbin &iz, const T &elem, const Arbin &dr) :
   _ra(new Nodo(copiaAux(iz._ra), elem, copiaAux(dr._ra))) {
}
```

Una copia similar hay que realizar con las operaciones `hijoIz` e `hijoDr` lo que automáticamente hace que el coste de estas operaciones sea $\mathcal{O}(n)$:

```
/**
 Devuelve un árbol copia del árbol izquierdo.
 Es una operación parcial (falla con el árbol vacío).

 hijoIz(Cons(iz, elem, dr)) = iz
 error hijoIz(ArbolVacio)
*/
Arbin hijoIz() const {
   if (esVacio())
      throw EArbolVacio();

   return Arbin(copiaAux(_ra->_iz));
}

/**
 Devuelve un árbol copia del árbol derecho.
 Es una operación parcial (falla con el árbol vacío).

 hijoDr(Cons(iz, elem, dr)) = dr
 error hijoDr(ArbolVacio)
*/
Arbin hijoDr() const {
```

[4] En realidad existe una tercera alternativa que veremos en la sección 3.3 más tarde en este mismo tema.

[5] Para eso en la cabecera deberían pasarse como parámetros de entrada/salida, es decir como referencias *no* constantes.

[6] Ver, no obstante, el ejercicio 3.

```
   if (esVacio())
      throw EArbolVacio();

   return Arbin(copiaAux(_ra->_dr));
}
```

La otra operación observadora, utilizada para acceder al elemento que hay en la raíz, no tiene necesidad de copias:

```
/**
 Devuelve el elemento almacenado en la raiz

 raiz(Cons(iz, elem, dr)) = elem
 error raiz(ArbolVacio)
 @return Elemento en la raíz.
*/
const T &raiz() const {
   if (esVacio())
      throw EArbolVacio();
   return _ra->_elem;
}
```

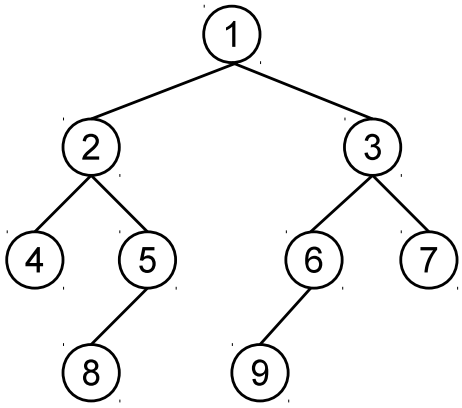
Otra operación observadora sencilla de implementar es `esVacio`:

```
/**
 Operación observadora que devuelve si el árbol
 es vacío (no contiene elementos) o no.

 esVacio(ArbolVacio) = true
 esVacio(Cons(iz, elem, dr)) = false
 */
bool esVacio() const {
   return _ra == NULL;
}
```

La última operación observadora que implementaremos será el operador de igualdad. En este caso delegamos en el método estático que compara las estructuras jerárquicas de nodos:

```
/**
 Operación observadora que indica si dos Arbin
 son equivalentes.

ArbolVacio == ArbolVacio
ArbolVacio != Cons(iz, elem, dr)
Cons(iz1, elem1, dr1) == Cons(iz2, elem2, dr2) sii
         elem1 == elem2, izq1 == izq2, dr1 == dr2
 */
bool operator==(const Arbin &a) const {
   return comparaAux(_ra, a._ra);
}
```

Con esta terminamos la implementación de las operaciones del TAD. Existen otras operaciones observadoras que suelen ser habituales en la implementación de los árboles binarios que permiten conocer algunas de las propiedades definidas en la sección 2. La implementación de la mayoría de ellas requiere la creación de métodos recursivos auxiliares que trabajen con la estructura jerárquica de los nodos. Ver por ejemplo el ejercicio 1.



Preorden: 1, 2, 4, 5, 8, 3, 6, 9, 7
Inorden: 4, 2, 8, 5, 1, 9, 6, 3, 7
Postorden: 4, 8, 5, 2, 9, 6, 7, 3, 1
Niveles: 1, 2, 3, 4, 5, 6, 7, 8, 9

Figura 2: Distintas formas de recorrer un árbol.

Operación	Complejidad
`ArbolVacio`	$\mathcal{O}(1)$
`Cons`	$\mathcal{O}(n)$
`hijoIz`	$\mathcal{O}(n)$
`hijoDr`	$\mathcal{O}(n)$
`esVacio`	$\mathcal{O}(1)$
`operator==`	$\mathcal{O}(n)$

3.2. Recorridos

Además de las operaciones observadoras indicadas anteriormente podemos *enriquecer* el TAD `Arbin` con una serie de operaciones que permitan recorrer todos los elementos del árbol. Si bien en el caso de las estructuras lineales el recorrido no ofrecía demasiadas posibilidades (solo había dos órdenes posibles, de principio al final o al contrario), en los árboles binarios hay distintas formas de recorrer el árbol.

Los cuatro recorridos que veremos procederán de la misma forma: serán operaciones observadoras que devolverán listas (`Lista<T>`) con los elementos almacenados en el árbol donde cada elemento aparecerá una única vez. El orden concreto en el que aparecerán dependerá del recorrido concreto (ver figura 2).

Los tres primeros recorridos que consideraremos tienen definiciones recursivas:

- Recorrido en *preorden*: se visita en primer lugar la *raíz* del árbol y, a continuación, se recorren en preorden el hijo izquierdo y el hijo derecho.
- Recorrido en *inorden*: la raíz se visita tras el recorrido en inorden del hijo izquierdo y antes del recorrido en inorden del hijo derecho.
- Recorrido en *postorden*: primero los recorridos en postorden del hijo izquierdo y derecho y al final la raíz.

La definición de todos ellos es similar. Por ejemplo:

```
preorden(ArbolVacio) = []
preorden(Cons(iz, elem, dr)) = [elem] ++ preorden(iz) ++ preorden(dr)
```

Podemos hacer una implementación recursiva directa de la definición anterior si admitimos la existencia de una operación de concatenación de listas:

```
Lista<T> Arbin<T>::preorden() const {
   return preordenAux(_ra);
}

static Lista<T> Arbin<T>::preordenAux(Nodo *p) {
   if (p == NULL)
      return Lista<T>();   // Lista vacía

   Lista<T> ret;
   ret.cons(p->_elem);
   ret.concatena(preordenAux(p->_iz));
   ret.concatena(preordenAux(p->_dr));

   return ret;
}
```

Sin embargo esa operación de concatenación no estaba disponible en la implementación básica de las listas; además su implementación puede tener coste lineal y puede tener como consecuencia directa tener una gran cantidad de nodos temporales. Existe una implementación mejor que, siendo también recursiva, hace uso de una lista como parámetro de entrada/salida que va *acumulando* el resultado hasta que termina el recorrido. El método `preordenAux` anterior se convierte en `preordenAcu` que tiene dos parámetros: un puntero a la raíz de la estructura jerárquica de nodos que visitar, y una lista (no necesariamente vacía) a la que se *añadirá* por la derecha los elementos del árbol.

```
preOrd(a) = preOrdAcu(a, [])
preOrdAcu(ArbolVacio, xs) = xs
preOrdAcu(Cons(iz, x, dr), xs) = preOrdAcu(dr, preOrdAcu(iz, xs++[x]))
```

```
Lista<T> preorden() const {
   Lista<T> ret;
   preordenAcu(_ra, ret);
   return ret;
}

// Método auxiliar (protegido o privado)
static void preordenAcu(Nodo *ra, Lista<T> &acu) {
   if (ra == NULL)
      return;

   acu.ponDr(ra->_elem);
   preordenAcu(ra->_iz, acu);
   preordenAcu(ra->_dr, acu);
}
```

La complejidad del recorrido es $\mathcal{O}(n)$, a lo que se puede llegar tras el análisis de recurrencias que utilizábamos en los algoritmos recursivos de los temas pasados. La misma complejidad tienen las implementaciones de los recorridos inorden y postorden que utilizan la misma idea:

```
inOrd(a) = inOrdAcu(a, [])
inOrdAcu(ArbolVacio, xs) = xs
inOrdAcu(Cons(iz, x, dr), xs) = inOrdAcu(dr, inOrdAcu(iz, xs)++[x])
```

```
Lista<T> inorden() const {
   Lista<T> ret;
   inordenAcu(_ra, ret);
   return ret;
}

// Métodos protegidos/privados
static void inordenAcu(Nodo *ra, Lista<T> &acu) {
   if (ra == NULL)
      return;

   inordenAcu(ra->_iz, acu);
   acu.ponDr(ra->_elem);
   inordenAcu(ra->_dr, acu);
}
```

```
postOrd(a) = postOrdAcu(a, [])
postOrdAcu(ArbolVacio, xs) = xs
postOrdAcu(Cons(iz, x, dr), xs) = postOrdAcu(dr, postOrdAcu(iz, xs))++[x]
```

```
Lista<T> postorden() const {
   Lista<T> ret;
   postordenAcu(_ra, ret);
   return ret;
}

// Métodos protegidos/privados
static void postordenAcu(Nodo *ra, Lista<T> &acu) {
   if (ra == NULL)
      return;

   postordenAcu(ra->_iz, acu);
   postordenAcu(ra->_dr, acu);
   acu.ponDr(ra->_elem);
}
```

El último tipo de recorrido que consideraremos es el recorrido *por niveles* (ver figura 2), que consiste en visitar primero la raíz, luego todos los nodos del nivel inmediatamente inferior de izquierda a derecha, a continuación todos los nodos del nivel tres, etc.

La implementación no puede ser recursiva sino iterativa. Hace uso de una cola que contiene todos los subárboles que aún quedan por visitar:

```
niveles(a) = nivelesCola(ponDetras(a, NuevaCola), NuevaLista)

nivelesCola(ColaVacia, xs) = xs
nivelesCola(as, xs) = nivelesCola(quitaPrim(as), xs)
        si NOT COLA.esVacia(as) AND ARBIN.esVacio(primero(as))

nivelesCola(as, xs) = nivelesCola(ponDetras(hijoDr(primero(as)),
                                    ponDetras(hijoIz(primero(as)),
                                      quitaPrim(as))),
                                  xs++raiz(primero(as)))
        si NOT COLA.esVacia(as) AND NOT ARBIN.esVacio(primero(as))
```

```
Lista<T> niveles() const {

   if (esVacio())
      return Lista<T>();

   Lista<T> ret;
   Cola<Nodo*> porProcesar;
   porProcesar.ponDetras(_ra);

   while (!porProcesar.esVacia()) {
      Nodo *visita = porProcesar.primero();
      porProcesar.quitaPrim();
      ret.ponDr(visita->_elem);
      if (visita->_iz != NULL)
         porProcesar.ponDetras(visita->_iz);
      if (visita->_dr != NULL)
         porProcesar.ponDetras(visita->_dr);
   }

   return ret;
}
```

La complejidad de este recorrido también es $\mathcal{O}(n)$ aunque para llegar a esa conclusión nos limitaremos, por una vez, a utilizar la intuición: cada una de las operaciones del bucle tienen coste constante, $\mathcal{O}(1)$. Ese bucle se repite una vez por cada nodo del árbol; para eso basta darse cuenta que cada subárbol (o mejor, cada subestructura) aparece una única vez en la cabecera de la cola.

3.3. Implementación eficiente de los árboles binarios

El coste de las operaciones observadoras `hijoIz` e `hijoDr` de los árboles binarios es lineal, ya que devuelven una *copia* nueva de los árboles. Eso hace que una función aparentemente inocente como la siguiente, que cuenta el número de nodos:

```
template <typename E>
unsigned int numNodos(const Arbin<E> &arbol) {
    if (arbol.esVacio())
        return 0;
    else
        return 1 +
               numNodos(arbol.hijoIz()) +
               numNodos(arbol.hijoDr());
}
```

no tenga un coste lineal.

La razón fundamental de esto es que no hemos permitido la *compartición* de la estructura jerárquica de nodos. Una forma (ingenua, como veremos en breve) de solucionar el problema de eficiencia de las operaciones observadoras sería que esa estructura fuera compartida por ambos árboles. La operación generadora devuelve un nuevo árbol cuya raíz *apunta al mismo nodo* que es apuntado por el puntero `_iz` de la raíz del árbol más grande.

La compartición de memoria es posible gracias a que los árboles son objetos *inmutables*. Efectivamente, una vez que hemos construido un árbol, éste *no* puede ser modificado ya

que todas las operaciones disponibles (`hijoIz`, `raiz`, etc.) son *observadoras* y nunca modifican el árbol. Gracias a eso sabemos que la devolución del hijo izquierdo compartiendo nodos *no* es peligrosa en el sentido de que modificaciones en un árbol afecten al contenido de otro, pues esas modificaciones son imposibles por definición del TAD. Si hubieramos tenido disponible una operación como `cambiaRaiz` la compartición no habría sido posible pues el siguiente código:

```
Arbin<int> arbol;

// ...
// aquí construimos el árbol con varios nodos
// ...

Arbin<int> otro;
otro = arbol.hijoIz();
otro.cambiaRaiz(1 + otro.raiz());
```

al cambiar el contenido de la raíz del árbol `otro` está también cambiando un elemento de `arbol`.

Desgraciadamente, sin embargo, esta solución de compartición de memoria no funciona en lenguajes como C++. Pensemos en el siguiente aparentemetne inocente:

```
Arbin<int> arbol;

// ...
// aquí construimos el árbol con varios nodos
// ...

Arbin<int> otro;
otro = arbol.hijoIz();
```

Cuando el código anterior termina y las dos variables salen de ámbito, el compilador llamará a sus destructores. La destrucción de la variable `arbol` eliminará todos sus nodos; cuando el detructor de `otro` vaya a eliminarlos, se encontrará con que ya no existían, generando un error de ejecución.

Por lo tanto, el principal problema es la destrucción de la estructura de memoria compartida. En lenguajes como Java o C# con recolección automática de basura la solución anterior es perfectamente válida. En el caso de C++ hay que buscar otra aproximación. En concreto, se pueden traer ideas del mundo de la recolección de basura a nuestra implementación de los árboles. La solución que adoptamos es utilizar lo que se llama *conteo de referencias*: cada nodo de la estructura jerárquica de nodos mantiene un contador (entero) que indica *cuántos punteros lo referencian*. Así, si tengo un único árbol, todos sus nodos tendrán un 1 en ese contador. La operación `hijoIz` construye un nuevo árbol cuya raíz apunta al nodo del hijo izquierdo, por lo que su contador *se incrementa*. Cuando se invoca al destructor del árbol, se decrementa el contador y si llega a cero se elimina él y recursivamente todos los hijos.

La implementación de la clase nodo con el contador queda así (aparecen subrayados los cambios):

```
class Nodo {
public:
   Nodo() : _iz(NULL), _dr(NULL), _numRefs(0) {}
   Nodo(Nodo *iz, const T &elem, Nodo *dr) :
```

```
      _elem(elem), _iz(iz), _dr(dr), _numRefs(0) {

      if (_iz != NULL) _iz->addRef();
      if (_dr != NULL) _dr->addRef();
   }

   void addRef() { assert(_numRefs >= 0); _numRefs++; }
   void remRef() { assert(_numRefs >0); _numRefs-; }

   T _elem;
   Nodo *_iz;
   Nodo *_dr;

   int _numRefs;
};
```

Las operaciones como `hijoIz()` ya no necesitan hacer la copia profunda de la estructura jerárquica de nodos; el constructor especial que recibe el puntero al nodo raíz simplemente incrementa el contador de referencias:

```
class Arbin {
public:
   ...

   Arbin hijoIz() const {
      if (esVacio())
         throw EArbolVacio();

      return Arbin(copiaAux(_ra->_iz));
   }

private:
   ...

   Arbin(Nodo *raiz) : _ra(raiz) {
      if (_ra != NULL)
         _ra->addRef();
   }
}
```

Tampoco se necesita la copia la construcción de un árbol nuevo a partir de los dos hijos, pues el árbol grande compartirá la estructura. El constructor crea el nuevo `Nodo` (cuyo constructor incrementará los contadores de los nodos izquierdo y derecho) y pone el contador del nodo recién creado a uno:

```
Arbin(const Arbin &iz, const T &elem, const Arbin &dr) :
   _ra(new Nodo(copiaAux(iz._ra), elem, copiaAux(dr._ra))) {
   _ra->addRef();
}
```

Por último, la liberación de la estructura jerárquica de nodos sólo se realiza si nadie más referencia el nodo. Por lo tanto el método de liberación recursivo cambia:

```
static void libera(Nodo *ra) {
   if (ra != NULL) {
      ra->remRef();
      if (ra->_numRefs == 0) {
```

```
            libera(ra->_iz);
            libera(ra->_dr);
            delete ra;
        }
    }
}
```

Gracias a estas modificaciones la complejidad de todas las operaciones pasa a ser constante, y el consumo de memoria se reduce pues no desperdiciamos nodos con información repetida.

Operación	Complejidad
`ArbolVacio`	$\mathcal{O}(1)$
`Cons`	$\mathcal{O}(1)$
`hijoIz`	$\mathcal{O}(1)$
`hijoDr`	$\mathcal{O}(1)$
`esVacio`	$\mathcal{O}(1)$

Ahora que disponemos de operaciones eficientes, podríamos implementar todos los recorridos que vimos en el apartado anterior (preorden, inorden, etc) como funciones externas al TAD Arbin. Ten en cuenta que ahora podemos navegar por su estructura usando los métodos públicos `hijoIz` e `hijoDr` sin necesidad de hacer copias.

Finalmente, te proponemos que realices el ejercicio 5 para terminar de ver la diferencia entre las dos implementaciones del TAD.

3.4. En el mundo real...

La técnica del conteo de referencias utilizada en la sección previa es muy común y anterior a la existencia de recolectores de basura (en realidad los primeros recolectores de basura se implementaron utilizando esta técnica, por lo que podríamos incluso decir que lo que hemos implementado aquí es un pequeño recolector de basura para los nodos de los árboles).

En un desarrollo más grande el manejo explícito de los contadores invocando al `addRef` y `quitaRef` es incómodo y propenso a errores, pues es fácil olvidar llamarlas. Para nuestra pequeña implementación hemos preferido utilizar ese manejo explícito para que se vea más claramente la idea, pero un olvido en un `quitaRef` de algún método habría tenido como consecuencia fugas de memoria pues el contador nunca recuperará el valor 0 para indicar que nadie lo está utilizando y que por tanto puede borrarse.

Para evitar este tipo de problemas se han implementado estrategias que gestionan de forma automática esos contadores. En particular, son muy utilizados lo que se conoce como *punteros inteligentes* que son variables que se comportan igual que un puntero pero que, además, cuando cambian de valor incrementan o decrementan el contador del nodo al que comienzan o dejan de apuntar.

4. Árboles de búsqueda

Cuando los elementos que se almacenan en el árbol binario pueden ordenarse (en C++ eso se traduce a que tienen implementada la comparación mediante el operador $<$) se puede hablar de la existencia de *árboles ordenados*. Entenderemos que un árbol está ordenado

si su recorrido en inorden está ordenado en orden estrictamente mayor[7]. De lo anterior se deduce inmediatamente que la raíz del árbol es mayor que todos los elementos del hijo izquierdo y menor que todos los elementos del hijo derecho; además tanto el hijo izquierdo como el hijo derecho deben estar, a su vez, ordenados (ver ejercicio 9).

Parece fácil ver que si suponemos que los elementos están distribuidos uniformemente y por tanto su talla es del orden de $log(n)$[8], establecer si un elemento está en el árbol tiene coste logarítmico (igual que en búsquedas binarias sobre vectores ordenados), pues cada vez que descendemos por uno de los lados del árbol dejamos atrás la mitad de los elementos. La ventaja de los árboles frente a los vectores ordenados, además, es que la inserción de un nuevo elemento *no* tiene coste lineal, pues no necesitaremos desplazar todos los elementos para hacer hueco en el vector; basta con encontrar en qué lugar del árbol debe ir el elemento para mantener el árbol ordenado y crear el nuevo nodo. Por su parte el borrado, aunque más difícil de ver de forma intuitiva, también tiene coste logarítmico.

Los árboles de búsqueda consisten en utilizar esta misma idea en nuestro beneficio. Si almacenamos como valores elementos de tipo `Persona` utilizando como orden el DNI, podremos encontrarlos rápidamente sin penalizar las inserciones ni borrados.

Los árboles de búsqueda que trataremos aquí van un paso más allá: separaremos por un lado el valor que utilizaremos para ordenar (que llamaremos de ahora en adelante *clave*) y por otro la información adicional asociada a ella (y que llamaremos *valor*). En cada nodo guardaremos, pues, una *pareja*: la clave (por la que se ordena) y su valor asociado. Eso implica automáticamente que los árboles de búsqueda están parametrizados *por dos tipos distintos* en vez de sólo por uno: el tipo de la clave (que debe poderse ordenar) y el tipo del valor.

Las operaciones del TAD `Arbus` son las siguientes:

- `ArbusVacio`: generadora. Construye un árbol de búsqueda vacío.
- `Inserta`: generadora. Añade una nueva clave con su valor asociado. Si la clave ya existía en el árbol se sobreescribe el valor.
- `borra`: modificadora, borra del árbol una clave.
- `esta`: observadora. Permite preguntar si una clave aparece almacenada en el árbol.
- `consulta`: observadora. Dada una clave cualquiera devuelve su valor asociado. Es parcial (falla si la clave no `esta`).
- `esVacio`: observadora que permite ver si el árbol guarda algo de información.

Las operaciones generadoras presentan una peculiaridad que no ha aparecido hasta ahora: un `Inserta` puede *anular* el resultado de un `Inserta` anterior. Eso implica que hay *más de una forma* de construir el mismo árbol de búsqueda. Por ejemplo, los siguientes árboles de búsqueda (con enteros como claves y caracteres como valor) son equivalentes:

```
Inserta(3, 'a', ArbusVacio)

Inserta(3, 'a', Inserta(3, 'b', ArbolVacio))

Inserta(3, 'a', Inserta(3, 'c', Inserta(3, 'b', ArbolVacio)))
```

[7] Eso implica que no permitimos la aparición de elementos repetidos; existen otras variaciones de estos árboles que sí lo hacen.

[8] Ver ejercicio 12.

4.1. Implementación

La implementación del TAD anterior se hará utilizando, igual que con los árboles binarios, una estructura jerárquica de nodos. La diferencia fundamental es que en esta ocasión esos nodos guardan dos elementos:

```
template <class Clave, class Valor>
class Arbus {
public:

   ...

protected:

   /**
    Clase nodo que almacena internamente la pareja (clave, valor)
    y los punteros al hijo izquierdo y al hijo derecho.
    */
   class Nodo {
   public:
      Nodo() : _iz(NULL), _dr(NULL) {}
      Nodo(const Clave &clave, const Valor &valor)
         : _clave(clave), _valor(valor),
           _iz(NULL), _dr(NULL) {}
      Nodo(Nodo *iz, const Clave &clave,
           const Valor &valor, Nodo *dr)
         : _clave(clave), _valor(valor),
           _iz(iz), _dr(dr) {}

      Clave _clave;
      Valor _valor;
      Nodo *_iz;
      Nodo *_dr;
   };

   ...

private:
   /**
    Puntero a la raíz de la estructura jerárquica
    de nodos.
    */
   Nodo *_ra;
};
```

Las operaciones generales que vimos para los árboles binarios siguen siendo válidas (la liberación, copia, etc.), pero extendiéndolas cuando corresponda para que tenga en cuenta la existencia de dos valores en vez de sólo uno.

El invariante de la representación del árbol de búsqueda exige lo mismo que en el caso de los árboles binarios (añadiendo que se cumpla el invariante de la representación tanto de la clave como del valor), y que se mantenga el orden de las claves[9]:

[9] La definición podría haberse hecho también en base al recorrido en inorden.

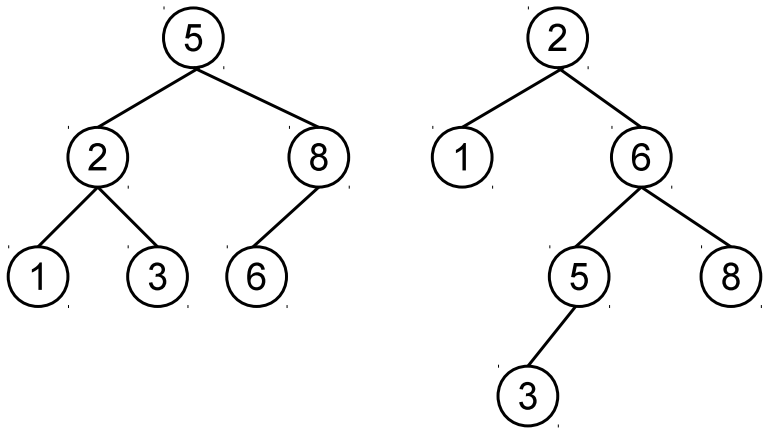


Figura 3: Dos árboles de búsqueda distintos pero equivalentes.

$$
\begin{aligned}
& R_{Arbus_{(C,V)}}(\mathsf{p}) \\
\Longleftrightarrow_{def} \\
& R_{Arbin_{Pareja(C,V)}}(\mathsf{p})^{10} \wedge \\
& ordenado(\mathsf{p})
\end{aligned}
$$

donde

$$
\begin{aligned}
ordenado(\mathsf{ptr}) &= true && \textbf{si} \;\; \mathsf{ptr} = \mathsf{null} \\
ordenado(\mathsf{ptr}) &= \forall c \in claves(\mathsf{ptr.\_iz}) : c < \mathsf{ptr.\_clave} \wedge \\
&\quad \forall c \in claves(\mathsf{ptr.\_dr}) : \mathsf{ptr.\_clave} < c && \textbf{si} \;\; \mathsf{ptr} \neq \mathsf{null}
\end{aligned}
$$

$$
\begin{aligned}
claves(\mathsf{ptr}) &= \emptyset && \textbf{si} \;\; \mathsf{ptr} = \mathsf{null} \\
claves(\mathsf{ptr}) &= \{\mathsf{ptr\_.clave}\} \bigcup claves(\mathsf{ptr.\_iz}) \bigcup claves(\mathsf{ptr.\_dr}) && \textbf{si} \;\; \mathsf{ptr} \neq \mathsf{null}
\end{aligned}
$$

La relación de equivalencia, sin embargo, no sigue la misma idea que en los árboles binarios. Para entender por qué basta ver los dos árboles de búsqueda de la figura 3, que guardan la misma información pero tienen una estructura arbórea totalmente distinta.

En realidad dos árboles de búsqueda son equivalentes si almacenan los mismos elementos. Los árboles de búsqueda tienen la misma interfaz que los conjuntos, y su relación de equivalencia también es la misma:

$$
\begin{aligned}
& \mathsf{a1} \equiv_{Arbin_T} \mathsf{a2} \\
\Longleftrightarrow_{def} \\
& elementos(\mathsf{a1}) \equiv_{Conjunto_{Pareja(C,V)}} elementos(\mathsf{a2})
\end{aligned}
$$

donde *elementos* devuelve un conjunto con todos los pares (clave, valor) contenidos en el árbol.

[10] Extendido para considerar claves y valores; el $Pareja(C, V)$ viene a indicarlo.

Implementación de la observadora `esVacio`

El árbol de búsqueda se implementa, igual que los árboles binarios, guardando un puntero a la raíz. Por lo tanto el constructor sin parámetros para generar un árbol de búsqueda vacío lo inicializa a `NULL`, y la operación observadora `esVacio` comprueba si esa condición se sigue cumpliendo:

```
/** Constructor; operacion ArbolVacio */
Arbus() : _ra(NULL) {
}

/**
 Operación observadora que devuelve si el árbol
 es vacío (no contiene elementos) o no.

 esVacio(ArbusVacio) = true
 esVacio(inserta(c, v, arbol)) = false
 */
bool esVacio() const {
   return _ra == NULL;
}
```

Implementación de `esta` y `consulta`

Las operaciones observadoras que permiten averiguar si una clave aparece en el árbol o acceder al valor asociado a una clave se basan en un método auxiliar que trabaja con la estructura de nodos y busca el nodo que contiene una clave dada, y que se implementa de manera trivial con recursión:

```
// Método protegido/privado
/**
 Busca una clave en la estructura jerárquica de
 nodos cuya raíz se pasa como parámetro, y devuelve
 el nodo en la que se encuentra (o NULL si no está).
 @param p Puntero a la raíz de la estructura de nodos
 @param clave Clave a buscar
*/
static Nodo *buscaAux(Nodo *p, const Clave &clave) {
   if (p == NULL)
      return NULL;

   if (p->_clave == clave)
      return p;

   if (clave < p->_clave)
      return buscaAux(p->_iz, clave);
   else
      return buscaAux(p->_dr, clave);
}
```

Con ella las operaciones `esta` y `consulta` son casi inmediatas:

```
/**
 Operación observadora que devuelve el valor asociado
 a una clave dada.
```

```
 consulta(e, inserta(c, v, arbol)) = v si e == c
 consulta(e, inserta(c, v, arbol)) = consulta(e, arbol) si e != c
 error consulta(ArbusVacio)

 @param clave Clave por la que se pregunta.
*/
const Valor &consulta(const Clave &clave) {
   Nodo *p = buscaAux(_ra, clave);
   if (p == NULL)
      throw EClaveErronea();

   return p->_valor;
}

/**
 Operación observadora que permite averiguar si una clave
 determinada está o no en el árbol de búsqueda.

 esta(e, ArbusVacio) = false
 esta(e, inserta(c, v, arbol)) = true si e == c
 esta(e, inserta(c, v, arbol)) = esta(e, arbol) si e != c

 @param clave Clave por la que se pregunta.
*/
bool esta(const Clave &clave) {
   return buscaAux(_ra, clave) != NULL;
}
```

Implementación de la inserción

La inserción debe garantizar que:

- Si la clave ya aparecía en el árbol, el valor antiguo se sustituye por el nuevo.
- Tras la inserción, el árbol de búsqueda sigue cumpliendo el invariante de la representación, es decir, sigue estando ordenado por claves.

La implementación debe "encontrar el hueco" en el que se debe crear el nuevo nodo, y si por el camino descubre que la clave ya existía, sustituir su valor.

Lo importante de la implementación es darse cuenta de que la raíz de la estructura jerárquica *puede cambiar*. En concreto, si el árbol de búsqueda estaba vacío, en el momento de insertar el elemento se crea un nuevo nodo que pasa a ser la raíz. Teniendo esto en cuenta la implementación (con un método auxiliar recursivo) sale casi sola:

```
/**
 Operación generadora que añade una nueva clave/valor
 a un árbol de búsqueda.
 @param clave Clave nueva.
 @param valor Valor asociado a esa clave. Si la clave
 ya se había insertado previamente, sustituimos el valor
 viejo por el nuevo.
*/
void inserta(const Clave &clave, const Valor &valor) {
   _ra = insertaAux(clave, valor, _ra);
```

```
}

// Método protegido/privado
/**
 Inserta una pareja (clave, valor) en la estructura
 jerárquica que comienza en el puntero pasado como parámetro.
 Ese puntero se admite que sea NULL, por lo que se creará
 un nuevo nodo que pasará a ser la nueva raíz de esa
 estructura jerárquica. El método devuelve un puntero a la
 raíz de la estructura modificada. En condiciones normales
 coincidirá con el parámetro recibido; sólo cambiará si
 la estructura era vacía.

 @param clave Clave a insertar. Si ya aparecía en la
 estructura de nodos, se sobreescribe el valor.
 @param valor Valor a insertar.
 @param p Puntero al nodo raíz donde insertar la pareja.
 @return Nueva raíz (o p si no cambia).
*/
static Nodo *insertaAux(const Clave &clave,
                        const Valor &valor, Nodo *p) {

   if (p == NULL) {
      return new Nodo(clave, valor);
   } else if (p->_clave == clave) {
      p->_valor = valor;
      return p;
   } else if (clave < p->_clave) {
      p->_iz = insertaAux(clave, valor, p->_iz);
      return p;
   } else { // (clave > p->_clave)
      p->_dr = insertaAux(clave, valor, p->_dr);
      return p;
   }
}
```

Si suponemos que el árbol está equilibrado, la complejidad del método anterior es logarítmica pues en cada llamada recursiva se divide el tamaño de los datos entre dos.

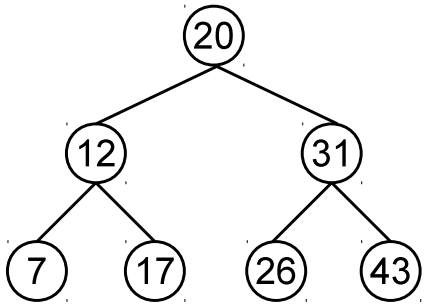
Implementación del borrado

La operación de borrado es más complicada que la de la inserción, porque puede exigir reestructurar los nodos para mantener la estructura ordenada por claves. Si nos piden eliminar la clave c, se busca el nodo que la contiene y:

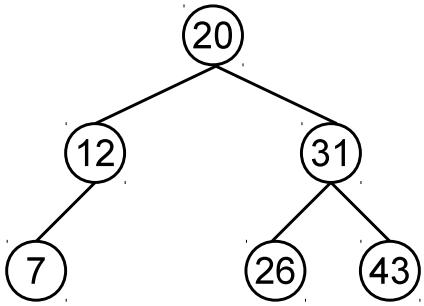
- Si la búsqueda fracasa (la clave no está), se termina sin modificar el árbol.
- Si la búsqueda tiene éxito, se localiza un nodo α, que es el que hay que borrar. Para hacerlo:
  - Si α es hoja, se puede eliminar directamente (actualizando el puntero del padre).
  - Si α tiene un sólo hijo, se elimina el nodo α y se coloca en su lugar el subárbol hijo cuya raíz quedará en el lugar del nodo α.

- Si α tiene dos hijos la estrategia que utilizaremos será "subir" el elemento más pequeño del hijo derecho (que no tendrá hijo izquierdo, pues de otra forma no sería el más pequeño) a la raíz. Para eso:
  - Se busca el nodo α' más pequeño del hijo derecho.
  - Si ese nodo tiene hijo derecho, éste pasa a ocupar su lugar.
  - El nodo α' pasa a ocupar el lugar de α, de forma que su hijo izquierdo y derecho cambian a los hijos izquierdo y derecho de la raíz antigua.

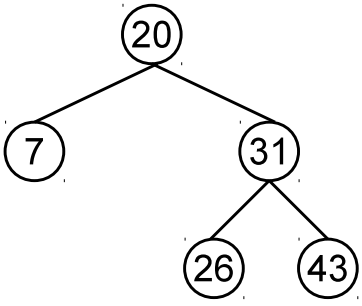
A modo de ejemplo, a continuación mostramos la evolución de un árbol de búsqueda a cuando vamos borrando sus nodos:



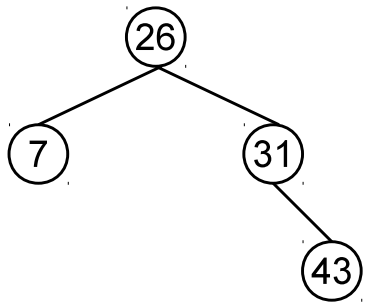
- `a.borrar(17)`



- `a.borrar(12)`



- `a.borrar(20)`



Igual que ocurría con la inserción, la implementación utiliza métodos auxiliares; como la raíz puede cambiar éstos devuelven la nueva raíz (que puede ser igual que la vieja si no hay cambios):

```
/**
 Operación modificadora que elimina una clave del árbol.
 Si la clave no existía la operación no tiene efecto.

 borra(elem, ArbusVacio) = ArbusVacio
 borra(e, inserta(c, v, arbol)) =
                   inserta(c, v, borra(e, arbol)) si c != e
 borra(e, inserta(c, v, arbol)) = borra(e, arbol) si c == e

 @param clave Clave a eliminar.
*/
void borra(const Clave &clave) {
   _ra = borraAux(_ra, clave);
}

protected:

/**
 Elimina (si existe) la clave/valor de la estructura jerárquica
 de nodos apuntada por p. Si la clave aparecía en la propia raíz,
 ésta cambiará, por lo que se devuelve la nueva raíz. Si no cambia
 se devuelve p.

 @param p Raíz de la estructura jerárquica donde borrar la clave.
 @param clave Clave a borrar.
 @return Nueva raíz de la estructura, tras el borrado. Si la raíz
 no cambia, se devuelve el propio p.
*/
static Nodo *borraAux(Nodo *p, const Clave &clave) {

   if (p == NULL)
      return NULL;

   if (clave == p->_clave) {
      return borraRaiz(p);
   } else if (clave < p->_clave) {
      p->_iz = borraAux(p->_iz, clave);
      return p;
   } else { // clave > p->_clave
      p->_dr = borraAux(p->_dr, clave);
      return p;
   }
```

```
}


/**
 Borra la raíz de la estructura jerárquica de nodos
 y devuelve el puntero a la nueva raíz que garantiza
 que la estructura sigue siendo válida para un árbol de
 búsqueda (claves ordenadas).
 */
static Nodo *borraRaiz(Nodo *p) {

   Nodo *aux;

   // Si no hay hijo izquierdo, la raíz pasa a ser
   // el hijo derecho
   if (p->_iz == NULL) {
      aux = p->_dr;
      delete p;
      return aux;
   } else
   // Si no hay hijo derecho, la raíz pasa a ser
   // el hijo izquierdo
   if (p->_dr == NULL) {
      aux = p->_iz;
      delete p;
      return aux;
   } else {
   // Convertimos el elemento más pequeño del hijo derecho
   // en la raíz.
      return mueveMinYBorra(p);
   }
}

/**
 Método auxiliar para el borrado; recibe un puntero a la
 raíz a borrar. Busca el elemento más pequeño del hijo derecho
 que se convertirá en la raíz (que devolverá), borra la antigua
 raíz (p) y "cose" todos los punteros, de forma que ahora:

   - El mínimo pasa a ser la raíz, cuyo hijo izquierdo y
     derecho eran los hijos izquierdo y derecho de la raíz
     antigua.
   - El hijo izquierdo del padre del elemento más pequeño
     pasa a ser el antiguo hijo derecho de ese mínimo.
 */
static Nodo *mueveMinYBorra(Nodo *p) {

   // Vamos bajando hasta que encontramos el elemento
   // más pequeño (aquel que no tiene hijo izquierdo).
   // Vamos guardando también el padre (que será null
   // si el hijo derecho es directamente el elemento
   // más pequeño).
   Nodo *padre = NULL;
   Nodo *aux = p->_dr;
   while (aux->_iz != NULL) {
      padre = aux;
```

```
      aux = aux->_iz;
   }

   // aux apunta al elemento más pequeño.
   // padre apunta a su padre (si el nodo es hijo izquierdo)

   // Dos casos dependiendo de si el padre del nodo con
   // el mínimo es o no la raíz a eliminar
   // (=> padre != NULL)
   if (padre != NULL) {
      padre->_iz = aux->_dr;
      aux->_iz = p->_iz;
      aux->_dr = p->_dr;
   } else {
      aux->_iz = p->_iz;
   }

   delete p;
   return aux;
}
```

4.2. Recorrido de los elementos mediante un iterador

En los árboles de búsqueda podríamos también añadir operaciones para el recorrido de sus elementos. Sin embargo, dado que el único recorrido que parece tener sentido es el recorrido en inorden (pues las claves salen en orden creciente), vamos a extender el TAD añadiendo la posibilidad de recorrerlo mediante un iterador. El iterador permitirá acceder tanto a la clave como al valor del elemento visitado. Eso sí, en este caso *no* permitirá cambiar los elementos pues pondría en peligro el invariante de la representación (si se cambia la clave directamente el árbol podría dejar de estar ordenado)[11].

El recorrido, sin embargo, no es tan fácil como en el caso de las listas, porque averiguar el siguiente elemento a un nodo dado no es directo. En concreto, hay dos casos:

- Cuando el nodo visitado actualmente tiene hijo derecho, el siguiente nodo a visitar será el elemento más pequeño del hijo derecho. Éste se obtiene bajando siempre por la rama de la izquierda hasta llegar a un nodo que no tiene hijo izquierdo.
- Si el nodo visitado no tiene hijo derecho, el siguiente elemento a visitar es el *ascendiente más cercano* que aún no ha sido visitado. Dado que la estructura jerárquica mantiene punteros hacia los hijos pero no hacia los padres, necesitamos una estructura de datos auxiliar. En particular, el iterador mantendrá una *pila* con todos los ascendientes que aún quedan por recorrer. En la búsqueda del hijo más pequeño descrita anteriormente vamos descendiendo por una rama, vamos apilando todos esos descendientes para poder visitarlos después.

El recorrido termina cuando el nodo actual no tiene hijo derecho y la pila queda vacía. En ese caso se cambia el puntero interno a `NULL` para indicar que estamos "fuera" del recorrido.

```
/**
 Clase interna que implementa un iterador sobre
```

[11] En realidad podríamos permitir cambiar únicamente el valor; lo dejamos como ejercicio.

```
 la lista que permite recorrer la lista e incluso
 alterar el valor de sus elementos.
 */
class Iterador {
public:
   const Clave &clave() const {
      if (_act == NULL) throw EAccesoInvalido();
      return _act->_clave;
   }

   const Valor &valor() const {
      if (_act == NULL) throw EAccesoInvalido();
      return _act->_valor;
   }

   void avanza() {
      if (_act == NULL) throw EAccesoInvalido();

      // Si hay hijo derecho, saltamos al primero
      // en inorden del hijo derecho
      if (_act->_dr)
         _act = primeroInOrden(_act->_dr);
      else {
         // Si no, vamos al primer ascendiente
         // no visitado. Para eso consultamos
         // la pila; si ya está vacía, no quedan
         // ascendientes por visitar
         if (_ascendientes.esVacia())
            _act = NULL;
         else {
            _act = _ascendientes.cima();
            _ascendientes.desapila();
         }
      }
   }

   bool operator==(const Iterador &other) const {
      return _act == other._act;
   }

   bool operator!=(const Iterador &other) const {
      return !(this->operator==(other));
   }
protected:
   // Para que pueda construir objetos del
   // tipo iterador
   friend class Arbus;

   Iterador() : _act(NULL) {}
   Iterador(Nodo *act) {
      _act = primeroInOrden(act);
   }

   /**
    Busca el primer elemento en inorden de
    la estructura jerárquica de nodos pasada
```

```
    como parámetro; va apilando sus ascendientes
    para poder "ir hacia atrás" cuando sea necesario.
    @param p Puntero a la raíz de la subestructura.
    */
   Nodo *primeroInOrden(Nodo *p) {
      if (p == NULL)
         return NULL;

      while (p->_iz != NULL) {
         _ascendientes.apila(p);
         p = p->_iz;
      }
      return p;
   }

   // Puntero al nodo actual del recorrido
   // NULL si hemos llegado al final.
   Nodo *_act;

   // Ascendientes del nodo actual
   // aún por visitar
   Pila<Nodo*> _ascendientes;
};

/**
 Devuelve el iterador al principio de la lista.
 @return iterador al principio de la lista;
 coincidirá con final() si la lista está vacía.
 */
Iterador principio() {
   return Iterador(_ra);
}

/**
 @return Devuelve un iterador al final del recorrido
 (fuera de éste).
 */
Iterador final() const {
   return Iterador(NULL);
}
```

Las operaciones funcionan de forma similar al visto en las listas: la operación `principio` devuelve un iterador al principio del recorrido mientras que `final` devuelve un iterador que queda *fuera* del recorrido.

5. Árboles generales

Los árboles generales no imponen una limitación *a priori* sobre el número de hijos que puede tener cada nodo. Por tanto, para implementarlos debemos buscar mecanismos generales que nos permitan almacenar un número de hijos variable en cada nodo.

Dos posibles soluciones:

- Cada nodo contiene una lista de punteros a sus nodos hijos. Si el número de hijos se conoce en el momento de la creación del nodo y no se permite añadir hijos a un

nodo ya creado, en lugar de una lista podemos utilizar un array.

- Cada nodo contiene un puntero al primer hijo y otro puntero al hermano derecho. De esa forma, podemos acceder al hijo i-ésimo de un nodo accediendo al primer hijo y luego recorriendo $i-1$ punteros al hermano derecho.

6. Para terminar...

Terminamos el tema con la solución a la motivación dada al principio del tema. La implementación utiliza un árbol de búsqueda como variable local que va almacenando, para cada palabra encontrada en el texto, el número de veces que ha aparecido.

```
void refsCruzadas(Lista<string> &texto) {

   Lista<string>::Iterador it(texto.principio());
   Arbus<string, int> refs;

   while (it != texto.final()) {
      if (!refs.esta(it.elem()))
         refs.inserta(it.elem(), 1);
      else
         refs.inserta(it.elem(), 1 + refs.consulta(it.elem()));
      it.avanza();
   }

   // Y ahora escribimos
   Arbus<string, int>::Iterador ita = refs.principio();
   while (ita != refs.final()) {
      cout << ita.clave() << " " << ita.valor() << endl;
      ita.avanza();
   }
}
```

Notas bibliográficas

Gran parte de este capítulo se basa en el capítulo correspondiente de (**?**) y de (**?**); algunos ejercicios son copias casi directas de los encontrados allí.

Es interesante ver las librerías de lenguajes como C++, donde la clase `std::set` y `std::map` están implementadas utilizando árboles ordenados y de búsqueda respectivamente. En esas implementaciones no se deja *al azar* la altura del árbol, sino que para garantizar que la altura siempre se mantiene dentro de unos límites, éste se reestructura en las operaciones de inserción y borrado. Existen numerosas estrategias para conseguir este comportamiento; puede consultarse (**?**) para más detalles.

Ejercicios

1. Extiende la implementación de los árboles binarios que no comparten memoria con las siguientes operaciones:

   - `numNodos`: devuelve el número de nodos del árbol.

- `esHoja`: devuelve cierto si el árbol es una hoja (los hijos izquierdo y derecho son vacíos).
- `numHojas`: devuelve el número de hojas del árbol.
- `talla`: devuelve la talla del árbol.
- `frontera`: devuelve una lista con todas las hojas del árbol de izquierda a derecha.
- `espejo`: devuelve un árbol nuevo que es la imagen especular del original.
- `minElem`: devuelve el elemento más pequeño de todos los almacenados en el árbol. Es un error ejecutar esta operación sobre un árbol vacío.

2. Implementa las mismas operaciones del ejercicio anterior pero como funciones *externas* al TAD. Estas nuevas funciones tendrá sentido utilizarlas si el TAD está implementado con compartición de memoria. ¿Por qué?

3. Implementa una nueva operación generadora de los árboles binarios que no comparten memoria similar a la ofrecida por el constructor con tres parámetros que construya un árbol a partir de los subárboles izquierdo y derecho pero que, para evitar la sobrecarga de las copias, deje vacíos los árboles que recibe como parámetro.

```
// Pre: iz y dr son dos árboles binarios válidos
Arbin<T> construyeYVacia(Arbin<T> &iz,
                         const T &elem,
                         Arbin<T> &dr);
// Post: devuelve el árbol Cons(iz, elem, dr), y
// altera iz y dr, de forma que esVacio(iz) y esVacio(dr).
```

4. Crea una función recursiva:

```
template <typename E>
void printArbol(const Arbin<E> &arbol);
```

que escriba por pantalla el árbol que recibe como parámetro, según las siguientes reglas:

- Si el árbol es vacío, escribirá `<vacio>` y después un retorno de carro.
- Si el árbol es un "árbol hoja", escribirá el contenido de la raíz y un retorno de carro.
- Si el árbol tiene algún hijo, escribirá el contenido del nodo raíz, y recursivamente en las siguientes líneas el hijo izquierdo y después el hijo derecho. Los hijos izquierdo y derecho aparecerán *tabulados*, dejando tres espacios.

Como ejemplo, el dibujo del árbol

```
Cons( Cons (vacio, 3, vacio),
      5,
      Cons (vacio, 6, Cons(vacio, 7, vacio)))
```

sería el siguiente (se han sustituido los espacios por puntos para poder ver más claramente cuántos hay):

```
5
...3
...6
......<vacio>
......7
```

5. Dibuja cómo queda la memoria de la máquina tras la ejecución del siguiente código con las dos implementaciones del TAD de los árboles binarios: la que no comparte memoria y la que sí lo hace:

   ```
   Arbin<int> vacio;
   Arbin<int> iz(vacio, 3, vacio);
   Arbin<int> dr(vacio, 4, vacio);
   Arbin<int> a(iz, 5, dr);
   Arbin<int> o = a.hijoIz();
   ```

   Dibuja el proceso de destrucción de los árboles si éste se realizara en el siguiente orden:

   - `a`, `iz`, `dr`, `o`.
   - `o`, `iz`, `a`, `dr`.

6. Los famosos números de Fibonacci (cuya serie es `0, 1, 1, 2, 3, 5, 8, 13, ...`) tienen un equivalente en el mundo de los árboles. Dado un n, entendemos por un árbol de Fibonacci de ese n aquel cuya raíz contiene el número de Fibonacci `fib(n)` y cuyo hijo izquierdo representa el árbol de fibonacci de $n-2$ y el derecho el de $n-1$. Evidentemente, cuando $n = 0$ el árbol es un árbol hoja con un 0 en la raíz y cuando $n = 1$ su raíz será 1.

   Implementa una función que, dado un n, devuelva el árbol de Fibonacci. ¿Cuántos nodos tiene el árbol? ¿Podrías encontrar una versión mejorada de la función anterior que maximice la compartición de nodos entre los distintos subárboles? Pinta el árbol de fibonacci de $n = 4$ con y sin compartición de estructura.

7. Decimos que un árbol binario es *homogéneo* cuando todos sus subárboles, excepto las hojas, tienen dos hijos no vacíos. Implementa una función que devuelva si un árbol dado es o no homogéneo.

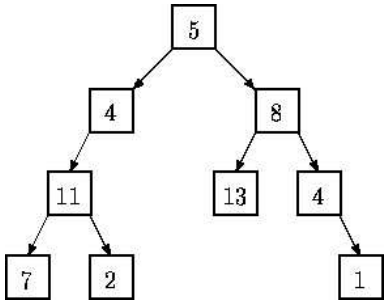
8. Escribe una operacion `degenerado()` que devuelva si un árbol es *degenerado*. Se entiende por árbol degenerado aquel en el que cada nodo tiene a lo sumo un hijo izquierdo o un hijo derecho.

9. Extiende la implementación de los árboles binarios con la siguiente operación:

   ```
   /**
   Devuelve true si el árbol binario cumple las propiedades
   de los árboles ordenados: la raiz es mayor que todos los elementos
   del hijo izquierdo y menor que los del hijo derecho y tanto el
   hijo izquierdo como el derecho son ordenados.
   */
   template <class T>
   bool Arbin::esOrdenado() const;
   ```

   Implementa la misma operación como función externa al TAD.

10. Dado un árbol binario de enteros, escribe una función que determine si existe un camino desde la raíz hasta una hoja cuyos nodos sumen un valor dado. Por ejemplo, en el siguiente árbol, las sumas de los distintos caminos son 27, 22, 26 y 18, por lo que si se utiliza la función con esos números devolverá `true`, devolviendo `false` en cualquier otro caso[12].



11. Diremos que un árbol binario es de *altura mínima* si no existe otro árbol binario con el mismo número de nodos con una altura menor. Dado un número de nodos n, ¿cuál es la altura mínima?

12. Los árboles de búsqueda se comportan mejor si están balanceados. Se entiende por árbol balanceado aquel en el que la talla del hijo izquierdo y la del hijo derecho no difieren en más de una unidad y ambos subárboles están a su vez balanceados. Extiende la implementación de los árboles binarios para que incorpore una nueva operación que diga si el árbol binario está balanceado. ¿Qué complejidad tiene? ¿Podrías idear una forma de conseguir la operación en coste $\mathcal{O}(1)$?

13. Implementa una operación en los árboles de búsqueda que *balancee* el árbol. Se permite el uso de estructuras de datos auxiliares.

14. ¿Notas algo extraño en la siguiente lista de teléfonos?

    - Emergencias: 911
    - Alicia: 97 625 999
    - Bob: 91 12 54 26

    Efectivamente, la lista es *incosistente*: el número de emergencias hace imposible poder llamar a Bob: cuando se marca el 911 la central telefónica dirige la llamada directamente hacia ellas aunque luego sigamos tecleando el resto del número de Bob.

    Implementa una función que reciba una lista de números de teléfono e indique si la lista es o no consistente según la explicación anterior[13].

15. Un árbol de codificación es un árbol binario que almacena en cada una de sus hojas un carácter diferente. La información almacenada en los nodos internos se considera irrelevante. Si un cierto carácter c se encuentra almacenado en la hoja de posición α definida como secuencias de 1's y 2's donde un 1 implica descender por el hijo izquierdo y un 2 por el hijo derecho, se considera que α es el código asignado a c por

[12] Este ejercicio es una traducción casi directa de `http://uva.onlinejudge.org/index.php?option=com_onlinejudg` la imagen del enunciado es una copia directa.

[13] El ejercicio es traducción casi directa de `http://uva.onlinejudge.org/index.php?option=com_onlinejudge&It`

ese árbol de codificación. Más en general, el código de cualquier cadena de caracteres dada se puede construir concatenando los códigos de los caracteres que la forman, respetando su orden de aparición.

- Dibuja el árbol de codificación correspondiente al código siguiente:

Carácter	Código
`A`	`1.1`
`T`	`1.2`
`G`	`2.1.1`
`R`	`2.1.2`
`E`	`2.2`

- Construye el resultado de codificar la cadena de caracteres "RETA" utilizando el código representado por el árbol de codificación anterior.
- Descifra `1.2.1.1.2.1.2.1.2.1.1` usando el código que estamos utilizando en estos ejemplos, construyendo la cadena de caracteres correspondiente.
- Desarrolla un módulo que implemente la clase `ArbCod` que representa a los árboles de codificación descritos, equipada con las siguientes operaciones:
  - `Nuevo`: genera un árbol de codificación vacío.
  - `Inserta`: dado un carácter y un código representado como lista de enteros en $[1, 2]$, añade el carácter al árbol en el lugar indicado por la secuencia. La operación no está definida si el carácter ya formaba parte del árbol.
  - `codifica`: dado un mensaje, devuelve su codificación. La operación no está definida si la cadena contiene algún carácter que no se encuentra codificado en el árbol.
  - `decodifica`: dado un código, devuelve la cadena que oculta. La operación no está definida si no es pobile decodificar toda la entrada.

16. Las implementaciones de los recorridos vistos en la sección 3.2 se hicieron como métodos de la clase (es decir, extendiendo el TAD de los árboles). Se hizo así porque en ese momento el TAD no compartía las estructuras de nodos por lo que la implementación de otra forma habría sido muy ineficiente. Implementa los tres tipos de recorrido utilizando funciones externas, contando con que todas las operaciones de los árboles tienen complejidad $\mathcal{O}(1)$.

17. Diseñar una función que reconstruya un árbol binario a partir de dos listas que contienen respectivamente su recorrido en preorden y en inorden. Suponer, si es necesario, que todos los elementos son distintos.

18. Repite el ejercicio anterior sustityuendo la lista del preorden por el recorrido en postorden.

19. Extiende los árboles binarios añadiendo una función que cree un recorrido en inorden con paréntesis anidados que identifiquen cada subárbol (incluidos los vacíos). Por ejemplo:

    - `()` representa el árbol vacío.
    - `(()A())` representa un árbol hoja con la `A` en la raíz.

- `(((`)`B(`)`)A(`)`C(`)`)` representa un árbol con la `A` en la raíz y en cada hijo una hoja con la `B` y la `C` respectivamente.

20. Implementa una función que a partir del recorrido en forma de lista anterior reconstruya el árbol de partida.

21. Extiende la implementación de los árboles binarios implementando el siguiente constructor (y todas las funciones auxiliares que necesites):

```
template <class T>
Arbin::Arbin(const Arbin &a1, const Arbin &a2);
```

    El método construye un nuevo árbol mezclando (o "sumando") los dos árboles que se reciben como parámetro. Se entiende por mezcla de dos árboles a "solapar" los dos árboles uno encima del otro y guardar el resultado de aplicar el operador suma a los contenidos de los nodos que aparezcan en ambos árboles.

    ¿Podrías implementar la misma idea con una función externa al TAD?

22. Añade las siguientes operaciones a los árboles de búsqueda y analiza su complejidad:

    - `consultaK`: recibe un entero k y devuelve la k-ésima clave del árbol de búsqueda, considerando que en un árbol con n elementos, $k = 0$ corresponde a la menor clave y $k = n - 1$ a la mayor.
    - `recorreRango`: dadas dos claves, a y b, devuelve una lista con los valores asociados a las claves que están en el intervalo $[a..b]$.

23. Extiende la implementación de los árboles binarios con una nueva operación, `maxNivel` que obtenga el máximo número de nodos de un nivel del árbol, es decir, el número de nodos del "nivel más ancho". Analiza su complejidad.

24. ¿Qué cambios realizarías en la implementación de los árboles de búsqueda para permitir almacenar distintos valores para la misma clave? Es decir:

    - Al insertar un par (clave, valor), si la clave ya se encontrase en el árbol, en lugar de sustituir el valor antiguo por el nuevo, se asociaría el valor adicional con la clave.
    - La operación de consulta en vez de devolver un único valor, devuelve una lista con todos ellos en el mismo orden en el que fueron insertados.
    - La operación de borrado elimina todos los valores asociados con la clave dada.

    Para la implementación no debes utilizar otros TADs.

25. En la sección de motivación veíamos que un texto puede venir como una lista de palabras. Otra alternativa es que venga como una lista de *líneas*, donde cada línea es a su vez una lista de palabras (es decir, el tipo sería `Lista<Lista<string>>`). El problema de las *referencias cruzadas* consiste en crear el listado en orden alfabético de todas las palabras que aparecen en el texto, indicando, para cada una de las palabras, el número de líneas en las que aparece (si una palabra aparece varias veces en una línea, el número de línea aparecerá repetido). Implementa en C++ un método que reciba un texto y escriba la lista de palabras ordenada alfabéticamente; cada línea contendrá una palabra seguida de todas las líneas en las que ésta aparece. Analiza su complejidad.

26. Extiende la implementación de los árboles binarios con las siguientes operaciones:

    - `esCompleto`: observadora, dado un árbol devuelve cierto si el árbol es completo.
    - `esSemicompleto`: igual que la anterior, pero para averiguar si un árbol es semicompleto.

27. Implementa una función que reciba una lista de enteros, algunos de ellos repetidos, y devuelva una lista con esos enteros ordenados de menor a mayor y eliminando las repeticiones.

28. Los árboles binarios hilvanados son una estructura dinámica más sofisticada de los árboles binarios que "aprovecha" los punteros que quedan vacíos (a `NULL`) en la representación dinámica ordinaria, de modo que puedan programarse eficientemente los algoritmos iterativos de recorrido.

    Para ello, cuando un nodo tiene un hijo derecho vacío, se hace apuntar al siguiente nodo en inorden. Cuando un nodo tiene un hijo izquierdo vacío, se hace apuntar al nodo anterior en el recorrido en inorden.

    Cambia el tipo representante de los árboles de búsqueda, para poder utilizar árboles hilvanados. El nuevo tipo representante deberá incluir dos valores binarios que indicarán si cada uno de los punteros a sus hijos son hilvanes o punteros a hijos.

    Modifica la implementación de las operaciones necesarias para mantener esos hilvanes. Por último, modifica la implementación del iterador para eliminar la necesidad de una pila.

    Extiendo, por último, los árboles de búsqueda para que permiten recorridos al contrario de forma similar al ejercicio **??** del tema anterior.

29. La evaluación contínua se le ha ido de las manos al profesor. Les pide a los alumnos que no lo dejen todo para el final sino que vayan estudiando día a día, pero él no predica con el ejemplo. Ahora tiene todos los ejercicios que los alumnos han ido entregando durante todo el año en una pila de folios y le toca revisarlos. Los ejercicios o están bien (y entonces puntúan positivamente) o están mal (y entonces restan).

    Al final del día quiere tener imprimida una lista con los nombres de todos los alumnos ordenados alfabéticamente y su puntuación en la evaluación contínua (resultado de sumar todos los ejercicios que tienen bien menos los que tienen mal). Si un alumno tiene un 0 como balance no debería aparecer en la lista.

    Aunque sea abusar un poco del alumnado... ¿puedes ayudar al profesor? Lo que se pide es que implementes una función que reciba dos listas de cadenas con los nombres de los alumnos. La primera lista tiene un elemento por cada ejercicio correcto entregado y contiene el nombre del alumno que lo entregó (por lo tanto un mismo alumno puede aparecer varias veces, si entregó varios ejercicios bien). De forma similar, la segunda lista contiene los ejercicios incorrectos. La función debe imprimir por pantalla el resultado final de la evaluación de los alumnos cuyo balance es distinto de 0 por orden alfabético.

Montículos

30. Cuando se trabaja con árboles semicompletos, las operaciones generadoras que se plantean son distintas: basta con poder crear un árbol vacío y añadir un nuevo ele-

mento (al final del último nivel, o "abriendo" un nivel nuevo). Con estas operaciones es posible implementar los árboles utilizando vectores en lugar de estructuras jerárquicas de nodos, de forma que los elementos aparecen en el mismo orden en el que aparecerían en un recorrido por niveles.

Dado un índice i del vector que representa un nodo del árbol, ¿en qué índice aparecerá si hijo izquierdo? ¿y el derecho? ¿y el padre?

31. Un árbol binario se dice que es un *montículo* (en inglés *heap*) si:

    - Es un árbol semicompleto.
    - La raíz del árbol contiene al elemento *más pequeño*.
    - El hijo izquierdo y el hijo derecho son, a su vez, montículos.

    Extiende la implementación de los árboles binarios para añadir una nueva operación `esMonticulo` que devuelva si el árbol es o no montículo.

32. Los montículos anteriores, al ser árboles binarios semicompletos, se almacenan más cómodamente en un vector.

    Dado un montículo almacenado en un vector en la que se ha añadido un nuevo elemento en el último nivel (que rompe la propiedad de *ser montículo*), implementa una función `flotar` que, con complejidad logarítmica, modifique el vector para que vuelva a ser un montículo.

33. Imaginemos que tenemos un montículo almacenado en un vector. Si cambiamos el valor de la raíz a un valor distinto, el árbol resultante puede dejar de ser montículo. Implementa una función `hundir` que, en ese escenario, modifique el árbol para conseguir recuperar la propiedad de ser montículo en tiempo logarítmico.

34. Utilizando la idea de los montículos (y las operaciones `flotar` y `hundir` de los ejercicios anteriores) implementa el TAD con las operaciones: `min`, `insertar` y `borraMin` que guardan una colección de objetos (no repetidos) y permiten: acceder al elemento más pequeño en tiempo constante, e insertar un nuevo elemento y borrar el más pequeño en tiempo logarítmico.

35. Dado un vector de elementos no repetidos, conviertelo en un montículo utilizando la operación `flotar` sin utilizar espacio adicional.

36. Utilizando el resultado de los ejercicios anteriores, implementa el método de inserción *heapsort* que consiste en ordenar un vector de mayor a menor convirtiéndolo primero en montículo y extrayendo el elemento más pequeño n veces colocándolo al final.