Capítulo 9

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. Además, se introducen distintos tipos de recorridos usando tanto listas como iteradores.

1. Motivación

Queremos hacer un pequeño juego en el que la máquina pide al usuario que piense un animal y ésta trata de adivinarlo haciendo distintas preguntas[2]:

```
¿Tiene cuatro patas? (Si/No)
no
¿Vive en el agua? (Si/No)
no
Mmmmmm, dejame que piense.....
Creo que el animal en que estabas pensando era...: ¡Serpiente!
¿Acerté? (Si/No)
si
¡Estupendo! Gracias por jugar conmigo.
```

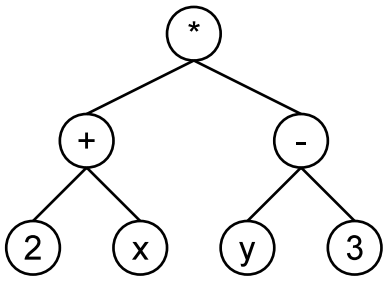
preguntando algo como `¿Es un reptil?` para en base a la respuesta, contestar *serpiente* u *orangután*.

¿Cómo implementarías la aplicación?

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.



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 $\mathtt{a}$.

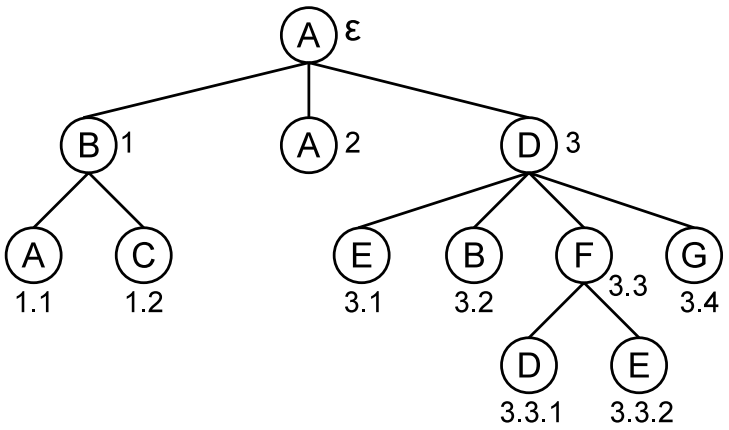


Figura 1: Posiciones y valores de cada nodo de un árbol.

- 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:

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

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

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:

- 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 `a` determina un *subárbol* $\texttt{a}_0$ con raíz en ese nodo.
- Dado un árbol `a`, los subárboles de `a` (si existen) se llaman *árboles hijos* de `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: operaciones

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.

4. Implementación de árboles binarios

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[3].

```
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;
   };
```

```
    */
   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 la raíz no incluyen a la propia raíz.

Con estas ideas el invariante queda:

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

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:

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

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"[4] 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:

```
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:

```
                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 15 del tema anterior)[5]:

- 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*[6].

En nuestra implementación nos decantaremos por la primera opción[7]:

```
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
```

```
      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 {
   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:

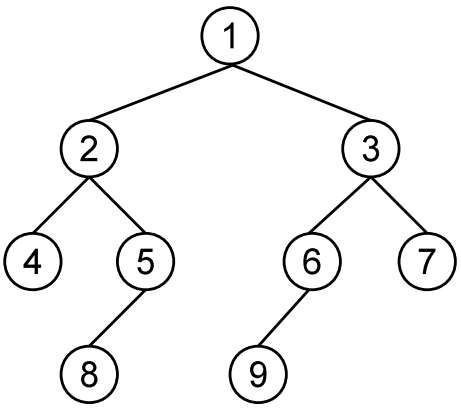
```
/**
 Devuelve el elemento almacenado en la raiz

 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;
}
```



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.

```
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.

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

5. Recorridos

Además de las operaciones observadoras indicadas anteriormente podemos *enriquecer* el TAD `Arbin` con una serie de operaciones que permitan recorrer todos los elementos

- 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:

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

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

   List<T> ret;
   ret.push_front(ra->_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]))
```

```
   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])
```

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

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

   inordenAcu(ra->_iz, acu);
   acu.push_back(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]
```

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

// Métodos protegidos/privados
```

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))
```

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

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

   List<T> ret;
   Queue<Nodo*> porProcesar;
   porProcesar.push_back(_ra);

   while (!porProcesar.empty()) {
      Nodo *visita = porProcesar.front();
      porProcesar.pop_front();
      ret.push_back(visita->_elem);
      if (visita->_iz)
         porProcesar.push_back(visita->_iz);
      if (visita->_dr)
         porProcesar.push_back(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.

```
    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 aparentemente inocente:

```
Arbin<int> arbol;

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

Arbin<int> otro;
```

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));
   }
```

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.

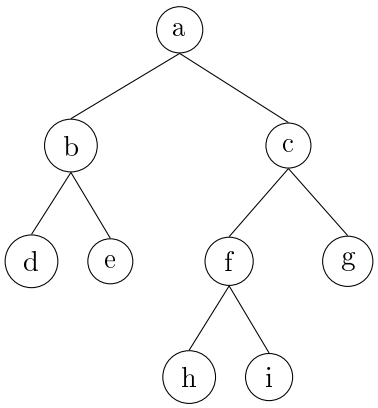
Finalmente, te proponemos que realices el ejercicio 5 para terminar de ver la diferencia entre las dos implementaciones del TAD.

7. Implementación estática ad-hoc de árboles binarios

Otra alternativa que puede utilizarse es la de almacenar el árbol en un vector en donde cada posición del vector contiene la información de un nodo: elemento almacenado en el nodo y los índices en donde se encuentran el hijo izquierdo y el hijo derecho.

```
class InfoNode {
public:
   ...
private:
   T _elem;
   int indexIz; // -1 si no hay hijo izquierdo
   int indexDr;
};
```

Así por ejemplo, si tenemos el siguiente árbol binario de caracteres:



Lo podemos almacenar en un vector de 9 posiciones[8]. Si el orden de inserción en el árbol anterior fue a, b, c, f, h, g, d, e, i, los nodos estarían colocados en ese mismo orden en el vector, y su contenido sería:

a		b		c		f		h		g		d		e		i		...
1	2	6	7	3	5	4	8	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	

8. 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`

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.

9. 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.

10. Para terminar...

Terminamos el tema con una aproximación inicial a la solución a la motivación dada al principio del tema.

La implementación puede hacerse con un árbol binario de cadenas. En los nodos hoja se almacena el nombre del animal al que representa, mientras que en cada nodo interno del árbol se almacena la pregunta que discrimina entre los animales almacenados en el hijo izquierdo y en el hijo derecho. En concreto, los animales almacenados en el hijo izquierdo son todo aquellos con los que se responde afirmativamente a la pregunta, mientras que los del hijo derecho son los que no la cumplen.

Así, el árbol con el que se puede conseguir la partida mostrada al principio del tema podría ser:

```
Arbin<string> bd =
   Arbin<string>(
      Arbin<string>("Tigre"),
      "¿Tiene cuatro patas?",
      Arbin<string>(
         Arbin<string>("Ballena"),
         "¿Vive en el agua?",
         Arbin<string>("Serpiente")
      )
```

```
   while (!esHoja(current)) {

      cout << current.raiz() << " (Si/No)" << endl;
      string line;
      cin >> line;
      if (line == "Si")
         current = current.hijoIz();
      else if (line == "No")
         current = current.hijoDr();
   }

   cout << "Mmmmmm, dejame que piense....." << endl;
   cout << "Creo que el animal en que estabas pensando era...: ¡"
        << current.raiz() << "!" << endl;

   cout << "Espero haber acertado." << endl;
}
```

Implementar el aprendizaje consiste en susituir la hoja en la que se termina por un nodo interno con la pregunta que discrimina y dos hijos hoja con los dos animales.

El TAD `Arbin<T>` no permite añadir nodos al árbol (pues éstos son inmutables), por lo que para poder implementar ese aprendizaje habría que extender el TAD o utilizar una implementación distinta, como la que hace uso de vectores estáticos de nodos vista en el apartado 7.

Notas bibliográficas

Gran parte de este capítulo se basa en el capítulo correspondiente de (Rodríguez Artalejo et al., 2011) y de (Peña, 2005); algunos ejercicios son copias casi directas de los encontrados allí.

Ejercicios

Algunos de los ejercicios puedes probarlos en el portal y juez en línea "Acepta el reto" (`https://www.aceptaelreto.com`). Son los que aparecen marcados con el icono seguido del número de problema dado en el portal.

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).

- `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
```

```
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. (ACR200) 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. (ACR201)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.

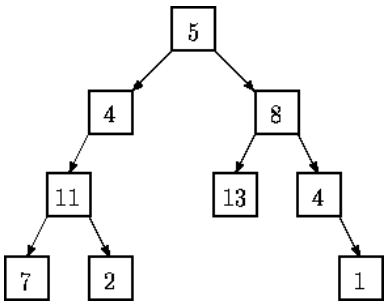
8. (ACR201)Escribe una operacion `degenerado()` que devuelva si un árbol es *degenerado*. Se entiende por árbol degenerado aquel en el que cada nodo tiene a lo sumo un hijo izquierdo o un hijo derecho.

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. [illegible]



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. (ACR275) 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. ¿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[10].

14. 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

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.

15. (ACR204)Dado un árbol binario de enteros se dice que éste está *pareado* si la diferencia entre el número de números pares del hijo izquierdo y del hijo derecho no excede la unidad y, además, tanto el hijo izquierdo como el derecho es *pareado*. Implementa una función que diga si un árbol binario de enteros está o no pareado.

16. 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.

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

18. (ACR203)Extiende la implementación de los árboles binarios implementando el siguiente constructor (y todas las funciones auxiliares que necesites):

19. 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.

20. Dado un árbol binario de enteros, implementa una función que determine cuál es la rama (camino desde la raíz hasta la hoja) cuya suma sea *mínima*, entendiendo ésta como la suma de todos los nodos por los que pasa.

21. Dado el *dibujo* de un árbol binario que representa una expresión aritmética con los operadores binarios de suma, resta, multiplicación y división y con operandos entre 0 y 9, implementa una función que devuelva el valor de la expresión.

Recorridos

22. Las implementaciones de los recorridos vistos en la sección 5 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 $\mathcal{O}(1)$.

23. (ACR215)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.

24. (ACR218)Repite el ejercicio anterior sustityuendo la lista del preorden por el recorrido en postorden.

25. Implementa una función que reciba el recorrido de cada nivel de un árbol ordenado y resconstruya el árbol.

26. Dado el recorrido en preorden de un árbol ordenado sin elementos repetidos, escribe su recorrido en postorden.

27. En el ejercicio 9 del tema anterior nos pedían que evaluáramos una expresión determinada utilizando una cola como estructura de datos auxiliar. Implementa una función que, dada una expresión en notación postfija, la traduzca al formato utilizado por el algoritmo del ejercicio 9.

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

    - `esCompleto`: observadora, dado un árbol devuelve cierto si el árbol es com-

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 elemento (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, convierte lo 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