Tema 6: Diseño e implementación de TADs arborescentes

Estructura de datos y Algoritmos
Curso 2020-2021

Grado en Desarrollo de Videojuegos
Facultad de Informática. UCM

(Slides adaptadas a partir de las originales de Marco Antonio Gómez etc.)

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:
  - Organización de un documento en capítulos, secciones, etc.
  - Estructura de directorios y archivos de un sistema operativo.
  - Árboles de sintaxis de programas.

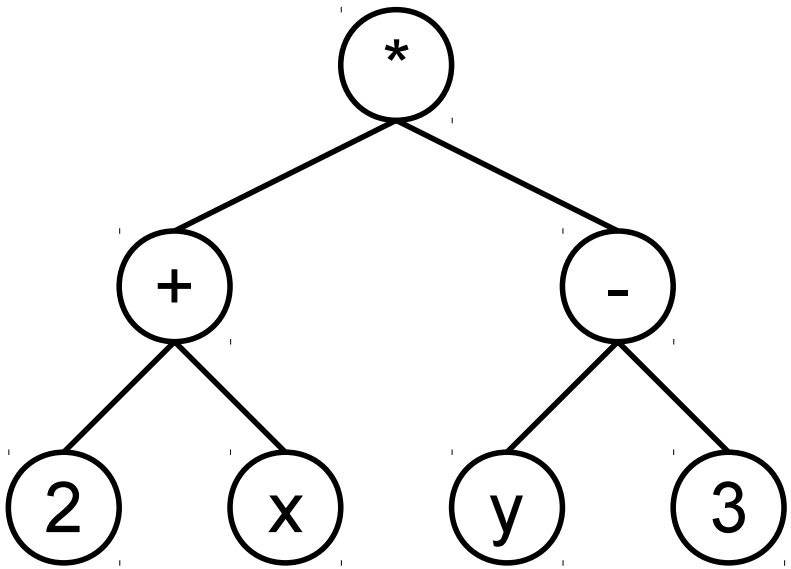

Figura 1: Aritmética con árboles

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

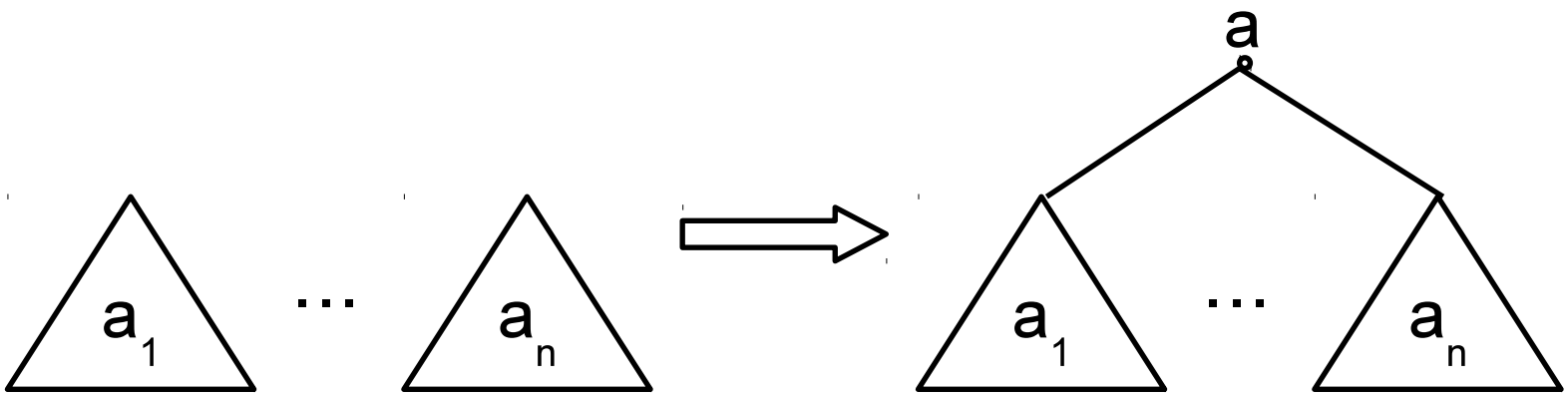

Figura 2: Construcción de un árbol

- 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 3 muestra un árbol y las cadenas que identifican las posiciones de sus nodos.

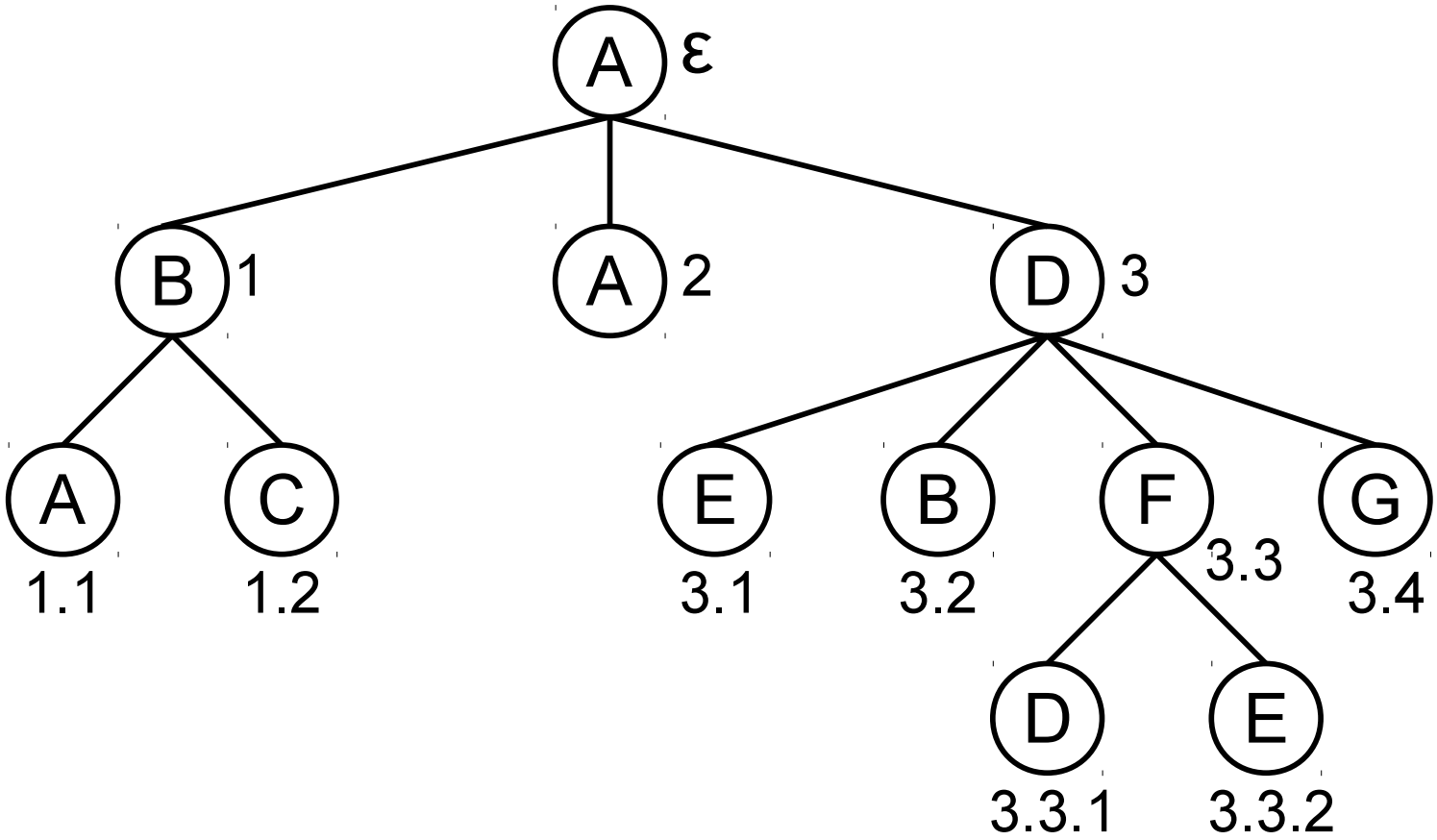

Figura 3: Modelo matemático de un árbol

- 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 3 de la siguiente manera:

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

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

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 $\mathtt{a}$.

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.

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 una serie de operaciones básicas con una utilidad muy limitada.
- En apartados siguientes lo extenderemos con otras operaciones.
- Las operaciones básicas son las siguientes:

- `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.

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.
- Representación: 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* debe confundirse un elemento del TAD *Arbin* con 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*.

- Habrá 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.

```
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:

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

- donde

$$
\begin{aligned}
\mathit{buenaJerarquia}(\text{ptr}) &= \mathit{true} && \texttt{si} \quad \text{ptr} = \text{null} \\
\mathit{buenaJerarquia}(\text{ptr}) &= \text{ubicado}(\text{ptr}) \wedge R_T(\text{ptr.\_elem}) \wedge \\
&\quad \mathit{nodos}(\text{ptr.\_iz}) \bigcap \mathit{nodos}(\text{ptr.\_dr}) = \emptyset \wedge \\
&\quad \mathit{ptr} \notin \mathit{nodos}(\text{ptr.\_iz}) \wedge \\
&\quad \mathit{ptr} \notin \mathit{nodos}(\text{ptr.\_dr}) \wedge \\
&\quad \mathit{buenaJerarquia}(\text{ptr.\_iz}) \wedge \\
&\quad \mathit{buenaJerarquia}(\text{ptr.\_dr}) && \texttt{si} \quad \text{ptr} \neq \text{null}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{nodos}(\text{ptr}) &= \emptyset && \texttt{si} \quad \text{ptr} = \text{null} \\
\mathit{nodos}(\text{ptr}) &= \{\mathit{ptr}\} \bigcup \mathit{nodos}(\text{ptr.\_iz}) \bigcup \mathit{nodos}(\text{ptr.\_dr}) \\
& && \texttt{si} \quad \text{ptr} \neq \text{null}
\end{aligned}
$$

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

$$\text{p1} \equiv_{Arbin_T} \text{p2}$$

$$\Longleftrightarrow_{def}$$

$$iguales_T(\text{p1.\_ra}, \text{p2.\_ra})$$

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

- 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" 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:

```
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(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):
  - 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*.
- En nuestra implementación nos decantaremos por la primera opción.

```
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 {
    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 1.
- La implementación de la mayoría de ellas requiere la creación de métodos recursivos auxiliares que trabajen con la estructura jerárquica de los nodos.

- Así, la complejidad de las operaciones de los árboles queda como sigue:

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

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 5).

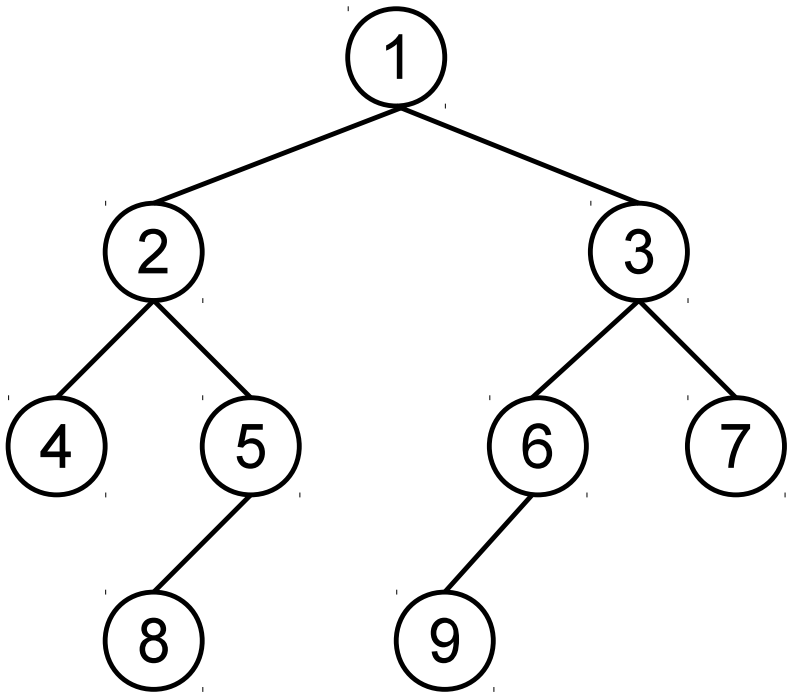

Figura 4: Aritmética con árboles

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 5: Distintas formas de recorrer un árbol.

- 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:

$$\begin{aligned}&\text{preorden}(\text{ArbolVacio}) = [\,]\\&\text{preorden}(\text{Cons}(\text{iz}, \text{elem}, \text{dr})) =\\&\qquad\qquad [\text{elem}] +\!+ \text{preorden}(\text{iz}) +\!+ \text{preorden}(\text{dr})\end{aligned}$$

- 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.

$$\begin{aligned}
&\text{preOrd(a)} = \text{preOrdAcu(a, [ ])}\\
&\text{preOrdAcu(ArbolVacio, xs)} = \text{xs}\\
&\text{preOrdAcu(Cons(iz, x, dr), xs)} =\\
&\qquad\qquad \text{preOrdAcu(dr, preOrdAcu(iz, xs++[x]))}
\end{aligned}$$

```
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 5), 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.

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, no tenga coste lineal.

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

- 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*.
- 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 código, aparentemente 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 destructor 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 en 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 y no desperdiciamos nodos con información repetida.

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

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 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 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.

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.