Diseño e implementación de TADs arborescentes

Marco Antonio Gómez y Antonio Sánchez son los autores principales de este tema

Facultad de Informática - UCM

23 de marzo de 2014

Introducción

Los TADs arborescentes se utilizan para representar datos organizados en jerarquías.

- 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.

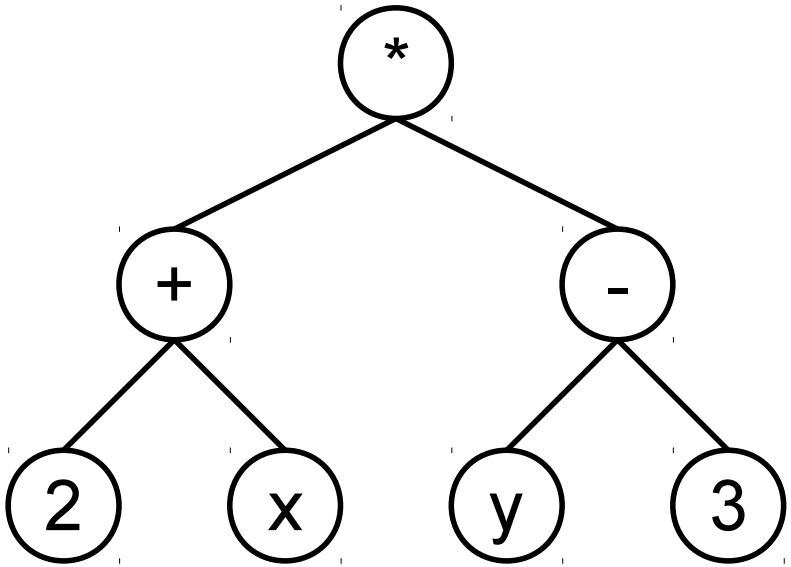


Figura 1 : Aritmética con árboles

Construcción de árboles

Los árboles están formados por *nodos*. Se construyen de manera inductiva:

- Un solo nodo es un árbol `a`. El nodo es la *raiz* 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`.

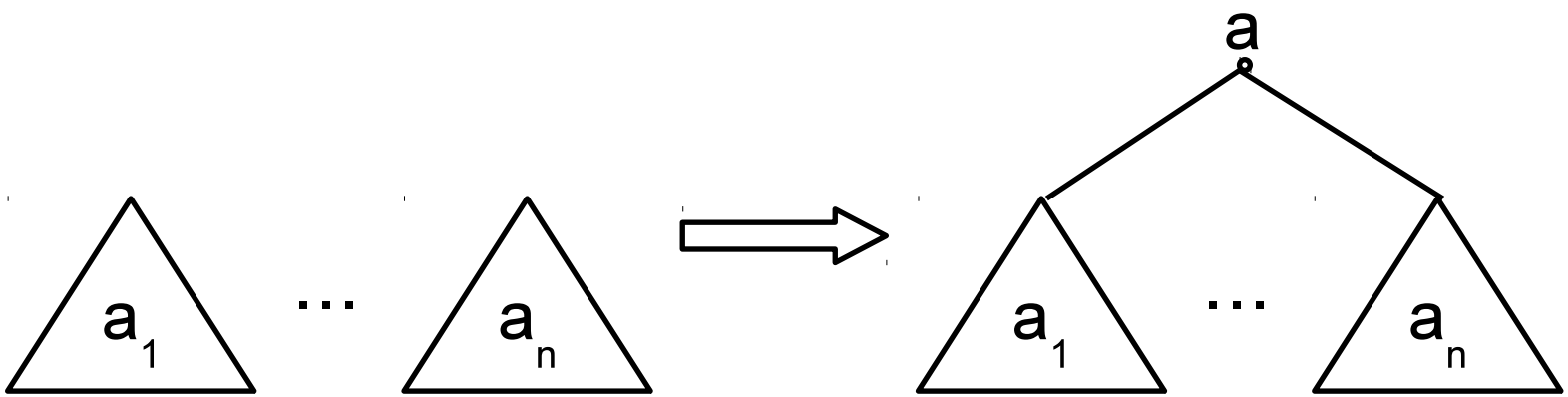
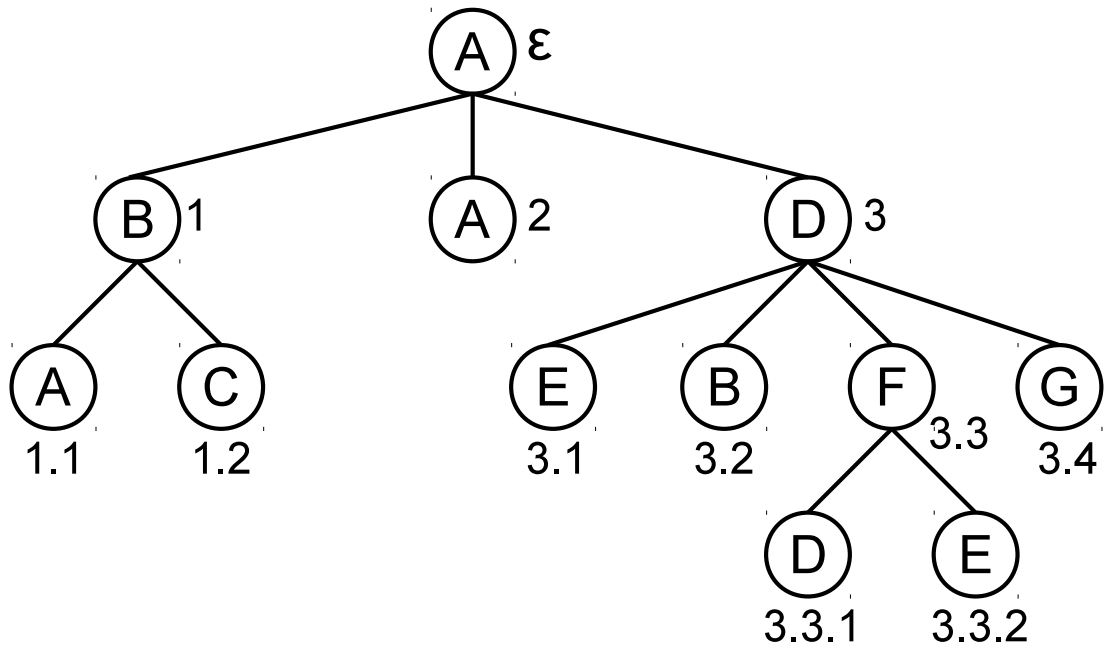


Figura 2 : Construcción de un árbol

Identificación de los nodos de un árbol.

- A cada nodo del árbol le asignamos una cadena de números:
  - La raíz del árbol tiene como posición la *cadena vacía* ϵ.
  - Si un nodo tiene como posición la cadena $\alpha \in \aleph^*$, el hijo i-ésimo de ese nodo tendrá como posición la cadena $\alpha.i$.
- Ejemplo:



Descripción de un árbol

- Un árbol viene dado por 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.
- Ejemplo: el árbol anterior se describe como:

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

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

Terminología

Dado un árbol $a : N \to V$

- 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 *descendiente* 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 número 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 es un árbol ordenado cuyos nodos tienen como mucho dos hijos, un hijo izquierdo y un hijo derecho.
- TAD *Arbin*: Operaciones básicas:
  - `ArbolVacio`: Construye un árbol sin ningún nodo.
  - `Cons(iz, elem, dr)`: Construye un árbol a partir de otros dos y de la información de la raíz.
  - `raiz`: Observadora parcial. Devuelve el elemento almacenado en la raíz del árbol.
  - `hijoIz`, `hijoDr`: Observadoras parciales. Obtienen el hijo izquierdo y el hijo derecho de un árbol dado.
  - `esVacio`: Observadora.

Implementación

- Tipo representante: Un vector y un índice con el número de nodos. v[0] es el nodo raíz.
  Hijos del nodo i: $2i + 1$ y $2i + 2$.
  Padre del nodo i: $(i - 1)/2$
  Solo es factible si el árbol es muy *denso* (árboles completos y semi-completos)
- Tipo representante: 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.
  Un *puntero* a la raíz de la *estructura jerárquica de nodos* que representan el árbol.

```
template <class T>
class Arbin {
public:    ...
protected:
     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
      Nodo *_ra;
};
```

Invariante de la representación:

- 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.

Relación de equivalencia:

$$\text{p1} \equiv_{Arbin_T} \text{p2} \Longleftrightarrow_{def} \mathit{iguales}_T(\text{p1.\_ra}, \text{p2.\_ra})$$

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

Métodos sobre la *estructura jerárquica*

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 toda la estructura:

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

Copia de una estructura jerárquica, devuelve el puntero a la raíz de la copia.

```
static Nodo *copiaAux(Nodo *ra) {
    if (ra == NULL)  return NULL;
    return new Nodo(copiaAux(ra->_iz), ra->_elem,
                    copiaAux(ra->_dr));
}
```

Métodos sobre la *estructura jerárquica*

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);
    }
}
```

Implementación de las operaciones públicas del TAD

Constructor que recibe dos árboles ya construidos:

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

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

- Compartición de memoria: la estructura jerárquica de los nodos de `iz` y de `dr` formará 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.

Implementación de las operaciones públicas del TAD

- Alternativas para solucionar el problema:
  - 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*.
- La siguiente implementación utiliza la primera opción.

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

Constructor sin parámetros.

```
Arbin() : _ra(NULL){}
```

Implementación de las operaciones públicas del TAD

- Operaciones `hijoIz` e `hijoDr` se realiza una copia de la estructura. El coste de las operaciones es $\mathcal{O}(n)$:

```
//Devuelve un árbol copia del árbol izquierdo (parcial).
Arbin hijoIz() const {
    if (esVacio()) throw EArbolVacio();
    return Arbin(copiaAux(_ra->_iz));
}

//Devuelve un árbol copia del árbol derecho (parcial).
Arbin hijoDr() const {
    if (esVacio()) throw EArbolVacio();
    return Arbin(copiaAux(_ra->_dr));
}
```

Es necesario declarar un constructor privado con parámetro un puntero al nodo raíz.

Implementación de las operaciones públicas del TAD

```
//Devuelve el elemento almacenado en la raiz
const T &raiz() const {
    if (esVacio())  throw EArbolVacio();
    return _ra->_elem;
}

// Operación observadora que devuelve si el árbol
// es vacío (no contiene elementos) o no.
bool esVacio() const {
    return _ra == NULL;
}

//Operación observadora que indica si dos Arbin
// son equivalentes.
bool operator==(const Arbin &a) const {
    return comparaAux(_ra, a._ra);
}
```

- La complejidad de las operaciones de los árboles implementados es:

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==`	$\mathcal{O}(n)$

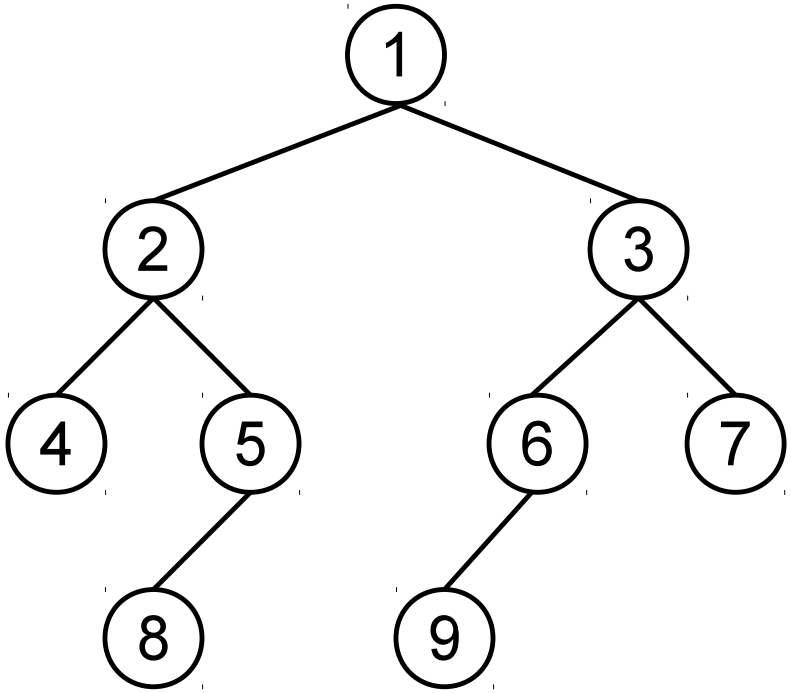
Instrucciones para crear un árbol

```
Arbin<int> a11;
Arbin<int> a12;
Arbin<int> a1(a11,1,a12);
Arbin<int> a2(a11,2,a12);
Arbin<int> a0(a1,0,a2);
cout << "La raiz del arbol es " << a0.raiz() << "\n";
Arbin<int> h1 = a0.hijoIz();
cout << "Raiz hijo izquierdo " << h1.raiz() << "\n";
Arbin<int> h2 = a0.hijoDr();
cout << "Raiz  hijo derecho " << h2.raiz() << "\n";
```

Recorridos

Podemos *enriquecer* el TAD `Arbin` con operaciones que permitan recorrer todos los elementos del árbol.
Formas de recorrer el árbol:

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



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

- Los cuatro recorridos se implementan 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.
- La definición de los recorridos en altura es recursiva. Por ejemplo:

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

Implementación basada en la definición recursiva (admitimos la existencia de una operación de concatenación en las 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;
}
```

La implementación de la operación de concatenación puede tener coste lineal. Con árboles equilibrados el coste del recorrido sería $\mathcal{O}(n \log n)$. Con árboles desequilibrados $\mathcal{O}(n^2)$.

Para mejorar el coste de la operación se utilizan parámetros acumuladores en la llamada recursiva.

```
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);
}
```

El coste del recorrido es $\mathcal{O}(n)$.

Recorrido en inorden.

```
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);
}
```

Recorrido en postorden.

```
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 recorrido por niveles se implementa con un algoritmo iterativo que hace uso de una cola que contiene todos los subárboles que aún quedan por visitar.

```
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

- Las operaciones `hijoIz` e `hijoDr` tienen coste lineal, ya que devuelven una *copia* nueva de los árboles.
- La función siguiente, que cuenta el número de nodos de un árbol, tiene coste $\mathcal{O}(n \log n)$ o $\mathcal{O}(n^2)$ cuando puede hacerse en 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());
}
```

- Podemos permitir compartir memoria porque los árboles son objetos *inmutables*. Todas las operaciones son *observadoras*.

- Aunque los objetos no cambien si se destruyen. Al compartir memoria si un árbol sale del ámbito en que se creó se llamará a su destructor y se liberará la memoría, destruyéndose también el árbol que comparta memoria con él.
- Por ejemplo:

```
Arbin<int> arbol;
// aquí construimos el árbol con varios nodos
Arbin<int> otro;
otro = arbol.hijoIz();
```

- Cuando el código termina y las dos variables salen de ámbito, se 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.

- La solución que adoptamos simula la recolección automática de basura de lenguajes como Java o C#.
- Se utiliza 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*.
- Mientras no se realiza ninguna copia del árbol, todos sus nodos tendrán un 1 en ese contador.
- La operación `hijoIz` al construir un nuevo árbol cuya raíz apunta al nodo del hijo izquierdo, incrementa el contador del nodo.
- Cuando se invoca al destructor del árbol, se decrementa el contador y si llega a cero se elimina él y recursivamente todos los hijos.

Implementación de la clase Nodo con contador

```
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;
};
```

La operación `hijoIz()` ya no necesitan hacer la copia profunda de la estructura jerárquica de nodos; el constructor especial que recibe el puntero al nodo raíz 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.

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

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.

La liberación de la estructura jerárquica de nodos sólo se realiza si nadie más referencia el nodo.

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

- La complejidad de todas las operaciones con esta nueva implementación 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`	$\mathcal{O}(1)$

- Con esta nueva implementación se pueden implementar los recorridos (preorden, inorden, etc) como funciones externas al TAD Arbin.
- Podemos navegar por la estructura del árbol usando los métodos públicos `hijoIz` e `hijoDr` sin necesidad de hacer copias.

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.