Capítulo 6

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.
```

Cuando la máquina falla, le pide al usuario que diga una pregunta que discrimine entre el animal por el que apostó y el animal real que pensó. De esta forma, la aplicación aprende de sus errores e incorpora en su base de conocimiento el nuevo animal. Si por ejemplo el usuario pensó en el orangután, en la siguiente partida en vez de decir *serpiente*, seguiría

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

[2] Un clon de un antiguo juego de los ordenadores de 8 bits de la década de 1980, llamado *Animal, vegetal, mineral*, `http://www.amstrad.es/programas/amsdos/educativos/animalvegetalmineral.php`.

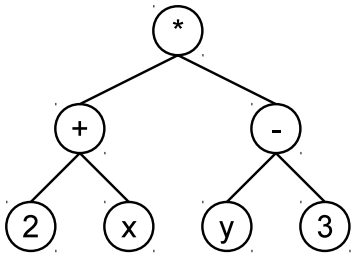
preguntando algo como `¿Es un reptil?` para en base a la respuesta, contestar *serpiente* u *orangután*.

¿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:

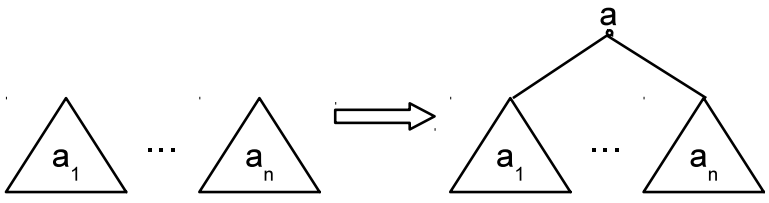
- Árboles genealógicos.
- Organización de un libro en capítulos, secciones, etc.
- Estructura de directorios y archivos de un sistema operativo.
- Árboles de análisis de expresiones aritméticas.



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 `a`. El nodo es la *raíz* del árbol.
- Dados n árboles $\mathtt{a_1}, \ldots, \mathtt{a_n}$, podemos construir un nuevo árbol `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 `a`.



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

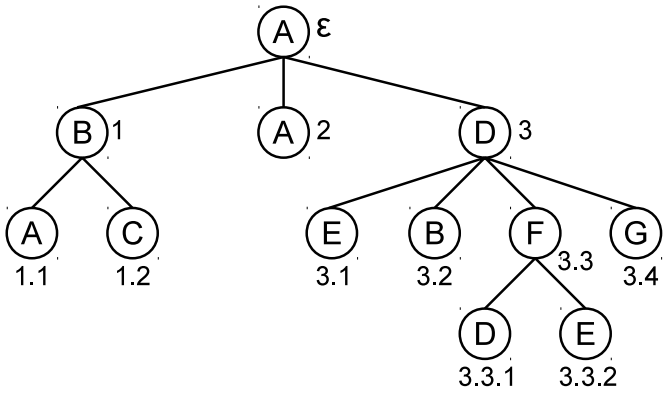


Figura 1: Posiciones y valores de cada nodo de un árbol.

- 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:

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

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

2.2. Terminología

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

- Cada *nodo* es una tupla $(\alpha, \mathtt{a}(\alpha))$ que contiene la posición y el valor asociado al nodo. Distinguimos 3 tipos de nodos:
  - La *raíz* es el nodo de posición ϵ.
  - Las *hojas* son los nodos de posición α tales que no existe i tal que $\alpha.i \in N$
  - Los *nodos internos* son los nodos que no son hojas.
- Un nodo $\alpha.i$ tiene como *padre* a α, y se dice que es *hijo* de α.
- Dos nodos de posiciones $\alpha.i$ y $\alpha.j$ $(i \neq j)$ se llaman *hermanos*.

- Un *camino* es una sucesión de nodos $\alpha_1, \alpha_2, \ldots \alpha_n$ en la que cada nodo es padre del siguiente. El camino anterior tiene *longitud* n.
- Una *rama* es cualquier camino que empieza en la raíz y acaba en una hoja.
- El *nivel* o *profundidad* de un nodo es la longitud del camino que va desde la raíz hasta al nodo. En particular, el nivel de la raíz es 1.
- La *talla* o *altura* de un árbol es el máximo de los niveles de todos los nodos del árbol.
- El *grado* o *aridad* de un nodo interno es su número de hijos. La *aridad de un árbol* es el máximo de las aridades de todos sus nodos internos.
- Decimos que α es *antepasado* de β (resp. β es *descenciente* de α) si existe un camino desde α hasta β.
- Cada nodo de un árbol `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 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.

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 *bintree*) tiene las siguientes operaciones:

- `Constructora sin argumentos`: operación generadora que construye un árbol vacío (un árbol sin ningún nodo).
- `Constructora con argumentos`: 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.
- `root`: operación observadora que devuelve el elemento almacenado en la raíz del árbol. Es parcial pues falla si el árbol es vacío.
- `left`, `right`: 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.
- `empty`: otra operación observadora para saber si un árbol tiene algún nodo o no.

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

4. Implementación de árboles binarios

Igual que ocurre con los TADs lineales, podemos implementar el TAD *bintree* 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 enlazados.

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 *bintree* con la estructura en memoria utilizada 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 `bintree`); 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 estructura `TreeNode` que, como no podría ser de otra manera, es una estructura interna a `bintree`. La clase `bintree` necesita únicamente almacenar un *puntero* a la raíz de la *estructura jerárquica de nodos* que representan el árbol[3].

```
template <class T>
class bintree {
protected:
   /*
    Nodo que almacena internamente el elemento (de tipo T),
    y punteros al hijo izquierdo y derecho, que pueden ser
    nullptr si el hijo es vacío (no existe).
    */
   struct TreeNode;

   using Link = TreeNode*;

   struct TreeNode {
      TreeNode(Link const& l, T const& e, Link const& r) : elem(e), left(l), right(r)
      T elem;
      Link left, right;
   };

   // puntero a la raíz del árbol
   Link raiz;

   ...

};
```

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

- Todos los nodos contienen información válida y están ubicados correctamente en

[3]Estas estructuras jerárquicas de nodos son útiles también para resolver otros problemas distintos a la implementación del TAD de los árboles binarios; ver por ejemplo el ejercicio 13.

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:

$$\begin{array}{l} R_{bintree_T}(\mathsf{p}) \\ \Longleftrightarrow_{def} \\ buenaJerarquia(\mathsf{p.raiz}) \end{array}$$

donde

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

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

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

$$\begin{array}{l} \mathsf{p1} \equiv_{bintree_T} \mathsf{p2} \\ \Longleftrightarrow_{def} \\ iguales_T(\mathsf{p1.raiz}, \mathsf{p2.raiz}) \end{array}$$

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

Antes de seguir con la implementación de las operaciones debemos crear algunos métodos que trabajan directamente con la *estructura jerárquica*, igual que en las implementaciones de los TADs lineales del tema anterior empezamos por las operaciones que trabajaban con las listas enlazadas y doblemente enlazadas. Dada la naturaleza recursiva de la estructura de nodos, todas esas operaciones serán recursivas; recibirán, al menos, un puntero a

un nodo que debe verse como la raíz de la estructura de nodos, o al menos la raíz del "subárbol"[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(Link ra) {
   if (ra != NULL) {
      libera(ra->left);
      libera(ra->right);
      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(Link r1, Link r2) {
   if (r1 == r2)
      return true;
   else if ((r1 == nullptr) || (r2 == nullptr))
      // En el if anterior nos aseguramos de
      // que r1 != r2. Si uno es nullptr, el
      // otro entonces no lo será, luego
      // son distintos.
      return false;
   else {
      return (r1->elem == r2->elem) &&
         comparaAux(r1->left, r2->left) &&
         comparaAux(r1->right, r2->right);
   }
}
```

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

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 Link copiaAux(Link ra) {
    if (ra == nullptr) return nullptr;
    return new TreeNode(copiaAux(ra->left),
                                ra->elem,
                                copiaAux(ra->right));
}
```

Tras esto, estamos en disposición de implementar las operaciones públicas del TAD. No obstante, antes de abordarlas expliquemos una decisión de diseño relevante. Hay una

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

operación generadora (que implementaremos como constructor) que recibe dos árboles ya construidos:

```
bintree(const bintree &iz, const T &elem, const bintree &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 (POR EL MOMENTO...)
bintree::bintree(const bintree &iz, const T &elem, const bintree &dr) {
   raiz = new Nodo(iz.raiz, elem, dr.raiz);
}
```

Esta implementación, no obstante, no es válida (por el momento) 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 podemos plantear 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]:

```
bintree(const bintree &iz, const T &elem, const bintree &dr) :
   raiz(new TreeNode(copiaAux(iz.raiz), elem, copiaAux(dr.raiz))) {
}
```

Una copia similar hay que realizar con las operaciones `left` e `right` 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).
*/
bintree left() const {
    if (empty())
        throw std::domain_error("El arbol vacio no tiene hijo izquierdo.");

    return bintree<T>(copiaAux(raiz->left));
}

/**
 Devuelve un árbol copia del árbol derecho.
 Es una operación parcial (falla con el árbol vacío).
*/
bintree right() const {
```

[5] En realidad existe una tercera alternativa que veremos en la sección 6 más tarde en este mismo tema.

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

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

```
    if (empty())
        throw std::domain_error("El arbol vacio no tiene hijo derecho.");

    return bintree<T>(copiaAux(raiz->right));
}
```

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
*/
const T& root() const {
    if (empty())
        throw std::domain_error("El arbol vacio no tiene raiz.");
    return raiz->elem;
}
```

Otra operación observadora sencilla de implementar es `empty`:

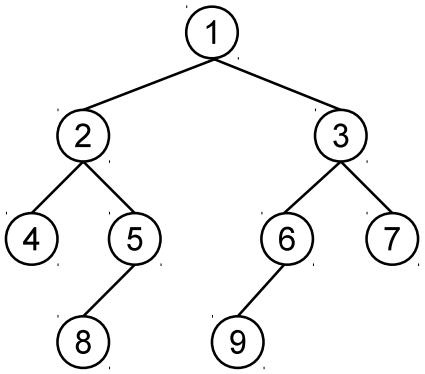
```
/**
 Operación observadora que devuelve si el árbol
 es vacío (no contiene elementos) o no.
 */
bool empty() const {
    return raiz == nullptr;
}
```

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 bintree
 son equivalentes.
 */
bool operator==(const bintree &a) const {
    return comparaAux(raiz, a.raiz);
}
```

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
`Constructora sin argumentos`	$\mathcal{O}(1)$
`Constructora con argumentos`	$\mathcal{O}(n)$
`left`	$\mathcal{O}(n)$
`right`	$\mathcal{O}(n)$
`empty`	$\mathcal{O}(1)$
`operator==`	$\mathcal{O}(n)$



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.

5. Recorridos

Además de las operaciones observadoras indicadas anteriormente podemos *enriquecer* el TAD `bintree` 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 vectores (`vector<T>`) con los elementos almacenados en el árbol donde cada elemento aparecerá una única vez. El orden concreto en el que aparecerán dependerá del recorrido concreto (ver figura 2).

Los tres primeros recorridos que consideraremos tienen definiciones recursivas:

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

Podemos hacer una implementación recursiva directa de la definición anterior si asumimos la existencia de una función que concatene vectores:

```
vector<T> bintree<T>::preorder() const {
   return preorder(raiz);
}

static vector<T> bintree<T>::preorder(Link p) {
   if (p == nullptr)
      return vector<T>(); // Vector vacío

   vector<T> ret;
   ret.push_back(ra->elem);
   concatena(ret, preorder(p->left));
```

```
   concatena(ret, preorder(p->right));

   return ret;
}
```

Sin embargo esa operación de concatenación no está disponible, y su implementación tendría coste lineal. Podemos implementarla por ejemplo basándonos en el método `insert` de la clase `vector` de esta forma:

```
template <class T>
void concatena(vector<T>& v1, const vector<T>& v2){
    v1.insert(v1.end(), v2.begin(), v2.end());
}
```

Esto da lugar a una implementación ineficiente. En concreto sería $\mathcal{O}(nlogn)$ de manera análoga a los algoritmos *quicksort* y *mergesort*. Observa que se hacen dos llamadas recursivas con la mitad de los datos y una llamada extra con coste lineal. Sin embargo, existe una implementación mejor que, siendo también recursiva, hace uso de un vector como parámetro de entrada/salida que va *acumulando* el resultado hasta que termina el recorrido. El método `preorder` auxiliar anterior se convierte en un método con dos parámetros: un puntero a la raíz de la estructura jerárquica de nodos que visitar, y un vector (no necesariamente vacío) al que se *añadirán* por la derecha los elementos del recorrido del resto del árbol.

```
   std::vector<T> preorder() const {
      std::vector<T> pre;
      preorder(raiz, pre);
      return pre;
   }

   static void preorder(Link a, std::vector<T> & pre) {
      if (a != nullptr) {
         pre.push_back(a->elem);
         preorder(a->left, pre);
         preorder(a->right, pre);
      }
   }
```

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:

```
   std::vector<T> inorder() const {
      std::vector<T> in;
      inorder(raiz, in);
      return in;
   }

   static void inorder(Link a, std::vector<T> & in) {
      if (a != nullptr) {
         inorder(a->left, in);
         in.push_back(a->elem);
         inorder(a->right, in);
      }
   }
```

```
std::vector<T> postorder() const {
   std::vector<T> post;
   postorder(raiz, post);
   return post;
}

static void postorder(Link a, std::vector<T> & post) {
   if (a != nullptr) {
      postorder(a->left, post);
      postorder(a->right, post);
      post.push_back(a->elem);
   }
}
```

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

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

```
std::vector<T> levelorder() const {
   std::vector<T> levels;
   if (!empty()) {
      std::queue<Link> pendientes;
      pendientes.push(raiz);
      while (!pendientes.empty()) {
         Link sig = pendientes.front();
         pendientes.pop();
         levels.push_back(sig->elem);
         if (sig->left != nullptr)
            pendientes.push(sig->left);
         if (sig->right != nullptr)
            pendientes.push(sig->right);
      }
   }
   return levels;
}
```

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.

6. Implementación eficiente de los árboles binarios

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

```
template <typename E>
unsigned int numNodos(const bintree<E> &arbol) {
    if (arbol.empty())
```

```
        return 0;
    else
        return 1 +
               numNodos(arbol.left()) +
               numNodos(arbol.right());
}
```

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 `left` 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 (`left`, `root`, 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:

```
bintree<int> arbol;

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

bintree<int> otro;
otro = arbol.left();
otro.cambiaRaiz(1 + otro.left());
```

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:

```
bintree<int> arbol;

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

bintree<int> otro;
otro = arbol.left();
```

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 vamos a implementar a continuación es utilizar lo que se llama *conteo de referencias*: cada nodo de la estructura jerárquica de nodos mantendrá un contador (entero) indicando *cuántos punteros lo referencian*. Así, si tengo un único árbol, todos sus nodos tendrán un 1 en ese contador. La operación `left` construirá un nuevo árbol cuya raíz apuntará al nodo del hijo izquierdo, por lo que su contador *se incrementará*. Cuando se invoque al destructor del árbol, se decrementará el contador y si llega a cero se eliminará él y recursivamente todos los hijos.

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

```
class TreeNode {
public:
   TreeNode() : left(nullptr), right(nullptr), numRefs(0) {}
   TreeNode(Link iz, const T &elem, Link dr) :
      elem(elem), left(iz), right(dr), numRefs(0) {

      if (left != nullptr) left->addRef();
      if (right != nullptr) right->addRef();
   }

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

   T elem;
   Link left;
   Link right;

   int numRefs;
};
```

Las operaciones como `left()` 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 bintree {
public:
   ...

    bintree left() const {
        if (empty())
            throw std::domain_error("El arbol vacio no tiene hijo izquierdo.");

        return bintree(copiaAux(raiz->left));
    }

protected:
    ...

    bintree(Link raiz) : raiz(raiz) {
        if (raiz != nullptr)
            raiz->addRef();
    }
}
```

Tampoco se necesitaría la copia en la construcción de un árbol nuevo a partir de los dos hijos, pues el árbol grande compartiría la estructura. El constructor crearía el nuevo `TreeNode` (cuyo constructor incrementaría los contadores de los nodos izquierdo y derecho) y pondría el contador del nodo recién creado a uno:

```
bintree(const bintree &iz, const T &elem, const bintree &dr) :
   raiz(new TreeNode(copiaAux(iz.raiz), elem, copiaAux(dr.raiz))) {
   raiz->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(Link ra) {
   if (ra != nullptr) {
      ra->remRef();
      if (ra->numRefs == 0) {
         libera(ra->left);
         libera(ra->right);
         delete ra;
      }
   }
}
```

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

Operación	Complejidad
`ArbolVacio`	$\mathcal{O}(1)$
`Cons`	$\mathcal{O}(1)$
`left`	$\mathcal{O}(1)$
`right`	$\mathcal{O}(1)$
`empty`	$\mathcal{O}(1)$

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 `bintree`. Ten en cuenta que ahora podemos navegar por su estructura usando los métodos públicos `left` y `right` 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 con *punteros inteligentes*

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 `remRef` es incómodo y propenso a errores, pues es fácil olvidar llamarlas. Para nuestra pequeña implementación hemos preferido utilizar ese manejo explícito para que se vea más

claramente la idea, pero un olvido en un `remRef` de algún método habría tenido como consecuencia fugas de memoria pues el contador nunca recuperará el valor 0 para indicar que nadie lo está utilizando y que por tanto puede borrarse.

Para evitar este tipo de problemas se han implementado estrategias que gestionan de forma automática esos contadores. En particular, son muy utilizados lo que se conoce como *punteros inteligentes* (en inglés *smart pointers*) 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.

Así quedaría la implementación de la parte básica de la clase `bintree` haciendo uso de punteros inteligentes. En particular, usaremos un puntero inteligente de tipo `shared_ptr` (puntero compartido) en la definición del tipo `Link` y haremos uso de la función `make_shared` cuando haya que construirlo en las constructoras de `bintree`.

```
template <class T>
class bintree {
protected:
   struct TreeNode;

   // Uso de puntero inteligente en la definición del tipo Link
   using Link = std::shared_ptr<TreeNode>;
   struct TreeNode {
      TreeNode(Link const& l, T const& e, Link const& r)
                        : elem(e), left(l), right(r) {};
      T elem;
      Link left, right;
   };

   // puntero a la raíz del árbol
   Link raiz;

   // constructora privada
   bintree(Link const& r) : raiz(r) {}

public:
   // árbol vacío
   bintree() : raiz(nullptr) {}

   // árbol hoja
   bintree(T const& e) : // Uso de make_shared para la creación de nodos
      raiz(std::make_shared<TreeNode>(nullptr, e, nullptr)) {}

   // árbol con dos hijos
   bintree(bintree<T> const& l, T const& e, bintree<T> const& r) :
      // Uso de make_shared para la creación de nodos
      raiz(std::make_shared<TreeNode>(l.raiz, e, r.raiz)) {}

   // consultar si el árbol está vacío
   bool empty() const {
      return raiz == nullptr;
   }

   // consultar la raíz
   T const& root() const {
      if (empty())
         throw std::domain_error("El arbol vacio no tiene raiz.");
```

```
      else return raiz->elem;
   }

   // consultar el hijo izquierdo
   bintree<T> left() const {
      if (empty())
         throw std::domain_error("El arbol vacio no tiene hijo izquierdo.");
      else return bintree<T>(raiz->left);
   }

   // consultar el hijo derecho
   bintree<T> right() const {
      if (empty())
         throw std::domain_error("El arbol vacio no tiene hijo derecho.");
      else return bintree(raiz->right);
   }

   ...
};
```

8. Implementación estática ad-hoc de árboles binarios

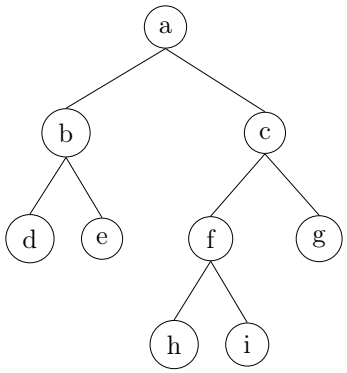
Las implementaciones anteriores de los árboles binarios utilizando una estructura jerárquica de nodos con memoria compartida dificulta (o mejor dicho, hace imposible) construir árboles "de arriba a abajo", es decir añadiendo nodos adicionales como hijos de otros nodos ya existentes.

Para poder implementarlo, se puede prescindir de la memoria compartida y dotar al árbol binario de algún tipo de iterador complejo de forma que se extienda el TAD con operaciones de inserción y borrado de nodos en el lugar marcado por el iterador.

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:
   ...
protected:
   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	

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:

[8]En realidad lo almacenaríamos en un vector más grande, llevando la cuenta de cuántas posiciones hay ocupadas, igual que se hizo con las implementaciones estáticas de algunos TADs lineales el tema anterior.

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

El juego que dirige la partida recibe el árbol y en un bucle va descendiendo por él hasta que llega a una hoja, momento en el que hace la apuesta final:

```
void juegaPartida(const bintree<string> &bd) {

   bintree<string> current = bd;

   while (!esHoja(current)) {
      cout << current.root() << " (Si/No)\n";
      string line;
      cin >> line;
      if (line == "Si")
         current = current.left();
      else if (line == "No")
         current = current.right();
   }

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

   cout << "Espero haber acertado.\n";
}
```

Implementar el aprendizaje consiste en sustituir 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 `bintree<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 8.

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).
   - `numHojas`: devuelve el número de hojas del árbol.
   - `talla`: devuelve la talla del árbol.
   - `frontera`: devuelve una lista con todas las hojas del árbol de izquierda a derecha.
   - (ACR228)`espejo`: devuelve un árbol nuevo que es la imagen especular del original.
   - `minElem`: devuelve el elemento más pequeño de todos los almacenados en el árbol. Es un error ejecutar esta operación sobre un árbol vacío.

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

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

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

4. Crea una función recursiva:

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

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

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

   Como ejemplo, el dibujo del árbol

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

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

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

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

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

   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.

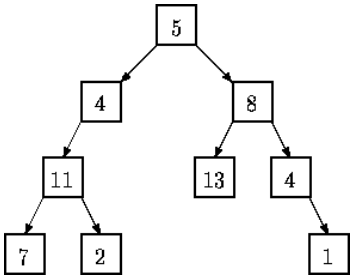
8. (ACR201)Escribe una operacion `degenerado()` que devuelva si un árbol es *degenerado*. Se entiende por árbol degenerado aquel en el que cada nodo tiene a lo sumo un hijo izquierdo o un hijo derecho.

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 bintree::esOrdenado() const;
   ```

Implementa la misma operación como función externa al TAD.

10. Dado un árbol binario de enteros, escribe una función que determine si existe un camino desde la raíz hasta una hoja cuyos nodos sumen un valor dado. Por ejemplo, en el siguiente árbol, las sumas de los distintos caminos son 27, 22, 26 y 18, por lo que si se utiliza la función con esos números devolverá `true`, devolviendo `false` en cualquier otro caso[9].



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

[9] Este ejercicio es una traducción casi directa de `http://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=3&page=show_problem&problem=48`; la imagen del enunciado es una copia directa.

[10] El ejercicio es traducción casi directa de `http://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=2347`.

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

- Dibuja el árbol de codificación correspondiente al código siguiente:

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

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

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):

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

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

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

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 sustituyendo 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 completo.
    - `esSemicompleto`: igual que la anterior, pero para averiguar si un árbol es semicompleto.

29. Dadas dos listas de enteros A y B, implementa una función que diga cuántos árboles binarios pueden construirse que tengan a A como recorrido en preorden y a B como recorrido en inorden.

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, conviertelo en un montículo utilizando la operación `flotar` sin utilizar espacio adicional.

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