Ejercicios del Tema 3

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

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

2. Implementa las mismas operaciones del ejercicio anterior pero como funciones externas al TAD.

3. Implementa una función recursiva:

```
template <class T>
void printArbol(const Arbin<T> &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
```

4. Los números de Fibonacci, definidos por (i) $F_0 = 0$, (ii) $F_1 = 1$, (iii) $F_{n+2} = F_{n+1} + F_n$ ($n \geq 0$), tienen un equivalente en el mundo de los árboles. Dado un número *n*, el árbol de Fibonacci para *n* es aquél que: (i) Consta únicamente del nodo 0 si n=0, (ii) consta únicamente del nodo 1 si n=1, (ii) si n > 1, es un árbol con F_n en la raíz, el árbol de Fibonacci para *n*-1 como hijo izquierdo, y el árbol de Fibonacci para *n*-2 como hijo derecho. Se pide:

- Implementar una función recursiva que, dado *n*, devuelva el correspondiente árbol de Fibonacci.
- Teniendo en cuenta que se está utilizando la implementación del TAD que comparte memoria, implementar otra versión de la función recursiva que maximice la compartición, en el sentido de que todos los subárboles para *k* compartan la misma estructura (pista: para n>2, ¿dónde aparece ya el hijo derecho?).
- Idear una versión iterativa para construir estos árboles, que maximice, además, la compartición de estructura (pista: básate en el cálculo iterativo de los números de Fibonacci).

5. Un árbol binario es *homogéneo* cuando todos los nodos interiores tienen dos hijos no vacíos. Escribe una función que determine cuándo un árbol es homogéneo.

6. Un árbol binario es *degenerado* cuando cada nodo tiene, a lo sumo, un hijo izquierdo o un hijo derecho. Implementa una función que determine si un árbol es degenerado. Implementa una versión recursiva y otra iterativa de dicha función.

7. Un árbol binario se dice *ordenado* si: (i) es vacío, o (ii) su raíz es mayor que todos los elementos del hijo izquierdo y menor que todos los elementos del hijo derecho, y además tanto el hijo izquierdo como el derecho son ordenados:

- Extiende la implementación del TAD con una operación `esOrdenado` que determine si el árbol es ordenado o no.
- Implementa la misma operación como una función externa al TAD, que utilice las operaciones básicas del mismo.

¿Podrías idear un algoritmo para estas operaciones que tenga complejidad lineal?

8. Dado un árbol binario de enteros:

I. Escribe una función que determine si existe un camino desde la raíz hasta una hoja cuyos nodos sumen un valor dado como parámetro.

II. Escribe una función que determine el número de caminos desde la raíz hasta las hojas tales que sus nodos suman un valor dado como parámetro.

9. Un árbol binario es *balanceado* cuando: (i) es vacío, o (ii) la diferencia de las tallas del hijo izquierdo y del derecho no es, en valor absoluto, mayor que 1 y ambos son balanceados. Escribe una función que determine si un árbol es o no balanceado. ¿Qué complejidad tiene dicha función? ¿Cómo modificarías la implementación del TAD para que dicha función tenga un coste O(1)?

10. Dado un árbol binario de enteros se dice que éste está *pareado* si, bien es vacío, o bien 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.

11. Implementa una función que construya un *string* con una representación *parentizada* de un árbol binario. En dicha representación:

- El árbol vacío se representa como `()`
- Un árbol con raíz A se representa como $(\tau_i\ A\ \tau_d)$, donde τ_i es la representación parentizada del hijo izquierdo, y τ_d la del derecho.

12. Implementa una función que a partir de un *string* que contiene una representación parentizada de un árbol, reconstruya el árbol representado.

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

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

El método debe construir 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. De esta forma, el valor de los nodos que aparecen en las mismas posiciones se obtiene aplicando el operador suma sobre los valores de los nodos correspondientes. El resto de nodos se preservan tal cual.

Implementa la misma operación mediante una función externa al TAD

14. Implementa una función `maxNivel` que reciba un árbol y obtenga el máximo número de nodos de un nivel del árbol dado, es decir, el número de nodos del "nivel más ancho". Analiza su complejidad.

15. Considérese la siguiente representación de expresiones aritméticas como árboles binarios:

- Si *n* es un número (representado en forma de *string*), `ArbolSimple`(*n*) es el árbol que representa dicho número.

- Si E_0 y E_1 son árboles que representan expresiones aritméticas, los árboles `Cons(`E_0`, "+", `E_1`)`, `Cons(`E_0`, "-", `E_1`)` , `Cons(`E_0`, "*", `E_1`)` y `Cons(`E_0`, "/", `E_1`)` son los árboles para, respectivamente, las correspondientes expresiones suma, resta, multiplicación y división

Implementa una función que, dado un árbol binario que representa una expresión aritmética en estos términos, devuelva su valor.

16. Las expresiones aritméticas en notación *prefija*, *postfija*, y *totalmente parentizada* se definen como sigue:

- **c***n*, con *n* un número es una expresión aritmética en notación prefija, *n***c** lo es en notación postfija, y (**c***n*) lo es en notación parentizada.
- Si E_0 y E_1 son expresiones aritméticas en notación prefija, también lo son las siguientes: +E_0E_1, -E_0E_1, *E_0E_1 y /E_0E_1. Si son expresiones en notación posfija, también lo son las siguientes: E_0E_1+, E_0E_1-, E_0E_1* y E_0E_1/. Si lo son en notación parentizada, también lo son las siguientes: (E_0+E_1), (E_0-E_1), (E_0*E_1) y (E_0/E_1)

Implementa funciones que, dado un árbol que representa una expresión aritmética, impriman, respectivamente, sus representaciones prefijas, posfijas, y parentizadas.

17. Implementa una función que, dado un *string* que representa una expresión en notación prefija, construya la representación como árbol binario de dicha expresión. Implementa otra función que haga lo propio con una expresión en notación parentizada, y otra que haga lo propio con una expresión posfija (en este último caso, utiliza una pila de árboles como estructura auxiliar: cada vez que leas un número, apila el correspondiente árbol, cada vez que leas un operador, desapila dos árboles, construye el correspondiente árbol utilizando estos como operandos, y apila el árbol resultante).

18. Un árbol es:

- *Semicompleto* cuando: (i) todos los nodos de niveles anteriores al penúltimo tienen exactamente dos hijos, (ii) los nodos del penúltimo nivel, vistos de izquierda a derecha, son tales que, primero aparecen nodos que tienen exactamente dos hijos hoja, y después aparecen nodos hoja.
- *Completo* cuando todos los nodos del mismo nivel, bien son hojas, bien tienen dos hijos no vacíos.

Escribe funciones que determinen cuándo un árbol es semi-completo, y cuándo es completo.

19. Implementar una función que reconstruya un árbol binario a partir de dos listas que representan, respectivamente, sus recorridos en preorden y en inorden. Suponer que todos los valores de los nodos son distintos entre sí. Resolver el problema suponiendo que las listas representan los recorridos en postorden y en inorden.

20. Implementar una función que reconstruya un árbol a partir de su recorrido por niveles, suponiendo que dicho árbol está ordenado.

21. Dado el recorrido en preorden de un árbol ordenado, implementa una función que devuelva el recorrido en postorden.

22. **(Junio 2016)** La compañía que gestiona la cuenca hidrográfica del río "Aguas Limpias" nos ha pedido que calculemos los tramos navegables de dicho río. Un tramo del río es navegable si contiene un caudal mayor o igual a 3 m3/s. Como todos los ríos, el Aguas Limpias está formado por una serie de afluentes. Los afluentes que nacen de los manantiales llevan un caudal de un 1 m3/s. Se considera que siempre confluyen exactamente dos afluentes. Cada vez que dos afluentes confluyen se calcula el caudal del tramo resultante como la suma de los caudales de sus afluentes. Adicionalmente, a lo largo del río se han construido varios embalses que hacen decrecer en una determinada cantidad el caudal del tramo saliente. En un embalse pueden confluir dos afluentes o solamente un tramo del río. La figura 1 es un ejemplo de un río que contiene 2 tramos navegables.

La estructura de afluentes se representa como un árbol binario cuyos nodos internos son o bien puntos de encuentro de dos afluentes, en cuyo caso el nodo contiene un 0, o bien un embalse, en cuyo caso el nodo contiene un número negativo que representa cuanto caudal puede absorber el embalse. Nótese que el caudal de un tramo no puede ser negativo pero sí 0. Implementa una función que dado un árbol que representa la estructura del río, calcule el número de tramos navegables.

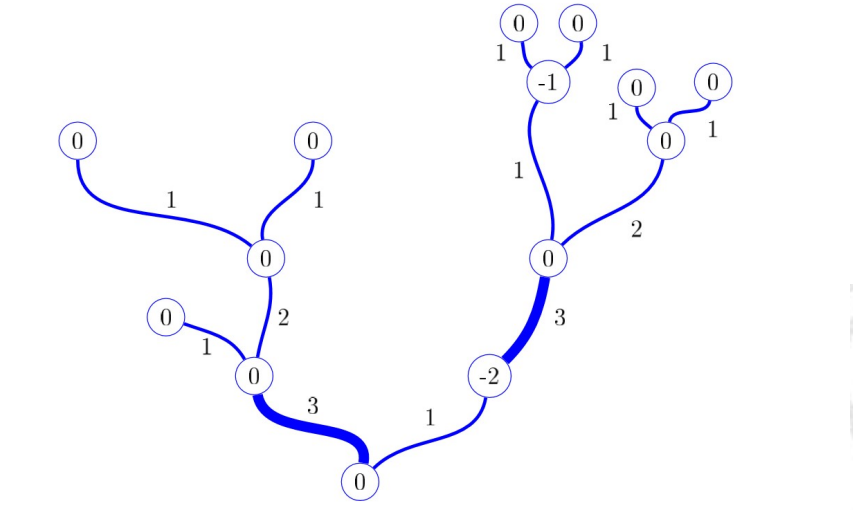


Figura 1

23. **(Junio 2015)** Podemos utilizar los árboles binarios para representar los caminos en la falda de una montaña. La raíz del árbol representa la cima de la que salen una o dos rutas. Las distintas rutas según se va ensanchando la falda de la montaña se dividen en dos formando caminos que nunca se volverán a conectar. Un escalador está en la cima de la montaña (raíz del árbol) y se da cuenta de que en distintas intersecciones (marcadas en el árbol con 'X') hay

amigos que necesitan su ayuda para subir. Tiene que bajar a cada una de las 'X' y ayudarles a subir de uno en uno. Implementa una función con la cabecera

```
int tiempoAyuda(const Arbin<char> &a);
```

que, dado uno de estos árboles binarios, devuelva el tiempo que tardará el escalador en ayudar a todos esos amigos si cada tramo de intersección a intersección lleva una hora en ser recorrido (en cada uno de los dos sentidos).

24. **(Septiembre 2015)** Implementa la siguiente función:

```
template <class T>
bool coinciden(const Arbin<T> &a, const Lista<T> &pre);
```

que diga si la lista `pre` coincide con el recorrido en preorden del árbol `a`. No puedes hacer uso de estructuras de datos auxiliares (arrays, pilas, colas, listas, árboles etc.) ni pedir/liberar memoria adicional (hacer **new** o **delete**).

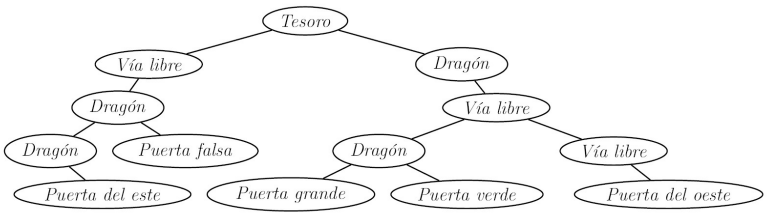
25. **(Junio 2014)** Dado un árbol binario, en cuya raíz se encuentra situado un tesoro y cuyos nodos internos pueden contener un dragón o no contener nada, se pide diseñar un algoritmo que nos indique la hoja del árbol cuyo camino hasta la raíz tenga el menor número de dragones. En caso de que existan varios caminos con el mismo número de dragones, el algoritmo devolverá el que se encuentre más a la izquierda de todos ellos. Para ello implementar una función que reciba un árbol binario cuyos nodos almacenan cadenas de caracteres:

- La raíz tiene la cadena `Tesoro`
- Los nodos internos tienen la cadena `Dragón` para indicar que en el nodo hay un dragón o la cadena `Vía libre` para indicar que no hay dragón.
- En cada hoja se almacena un identificador que no puede estar repetido.

La función debe devolver el identificador de la hoja del camino seleccionado. El árbol tiene como mínimo un nodo raíz y un nodo hoja diferente de la raíz. La operación no se implementa como parte de ningún TAD.

El coste de la operación implementada debe ser O(n).

Por ejemplo, dado el siguiente árbol el algoritmo devolverá la cadena de caracteres `Puerta falsa`.



26. **(Septiembre 2014)** Dado un árbol binario de enteros, se entiende que es un *árbol genealógico correcto* si cumple las siguientes reglas, producto de interpretar el entero de cada nodo como el año de nacimiento del individuo, el

hijo izquierdo como su primer hijo de un máximo de dos, y el hijo derecho como el segundo hijo:

- La edad del padre siempre supera en al menos 18 años las edades de cada uno de los hijos.
- La edad del segundo hijo (si existe) es al menos dos años menos que la del primer hijo (no hay hermanos gemelos/mellizos en estos árboles).
- Los árboles genealógicos de ambos hijos son también correctos.

Implementa una función que reciba un árbol binario que permita averiguar si un árbol binario es o no árbol genealógico correcto y, en caso de serlo, el número de generaciones distintas que hay en la familia.

27. **(Junio 2013)** Dado un árbol binario, diremos que un nodo es *hijo único* si su nodo padre solamente tiene un hijo (él mismo). Dado un árbol binario, diseña una función que calcule el número de hijos únicos con una complejidad O(n).

28. **(Septiembre 2013)** Dado un árbol binario, se llama *altura mínima* del árbol a la menor de sus distancias desde la raíz a un subárbol vacío. Decimos que un árbol binario es *zurdo* si o, bien es vacío, o si no lo es, ambos hijos son zurdos y la altura mínima del hijo izquierdo nunca es menor que la del hijo derecho. Implementa una función que determine si un árbol es zurdo o no.

29. **(Septiembre 2012)** El tipo *Personaje* es un tipo con tres valores: `dragon`, `caballero` o `princesa`. En un árbol binario de personajes, un nodo se dice *accesible* si en el camino que lo une con la raíz no hay ningún nodo `dragon`. Implementa una función que, dado un árbol de personajes, devuelva el número de nodos `princesa` accesibles.