ÁRBOLES

Jesús Pérez Melero

Contenido

Árboles Generales

La función de los árboles es organizar datos de manera jerárquica. Se utilizan en la implementación de otros TADs para mejorar su eficiencia.

Un **árbol general** es o bien vacío, o tiene dos componentes: un nodo raíz y un árbol/es o subárbol/es hijos.

Los árboles están formados por nodos. Estos nodos tienen un elemento y un conjunto de nodos que son la/las raíz/raíces de los subárboles hijos.

Un árbol libre es un grafo conectado, no dirigido ya acíclico.

Un árbol ordinario es un árbol libre en el que se elige un nodo como raíz.

Terminología

- **Raíz:** Nodo sin padre
- **Nodo interno (Internal node):** Nodo con al menos un hijo.
- **Nodo externo (External node):** Nodo sin hijos. Sinónimo de *leaf-node*.
- **Subárbol (subtree):** árbol formado por el nodo considerado como raíz junto a todos sus descendientes.
- **Ancestro:** Un nodo A es ancestro de otro nodo B únicamente cuándo A es B, o A es el padre de B, o A es ancestro del padre de B. Se dice que un nodo es **ancestro propio** de otro si un nodo es ancestro de otro pero no son el mismo nodo.
- **Descendiente:** Un nodo A es descendiente de otro nodo B únicamente si B es ancestro de A.
- **Hermano (sibling):** Nodos con el mismo padre.
- **Arista (edge):** Par de nodos con relación padre-hijo o hijo-padre.
- **Grado de un nodo (degree):** Número de hijos del nodo.
- **Grado de un árbol (degree):** Número máximo de los grados de todos los nodos
- **Camino de un árbol (path):** Secuencia de nodos tal que cada nodo consecutivo va formando una arista. La **longitud** del camino es el número de aristas.
- **Árbol ordenado (ordered tree):** existe un orden lineal (total) definido para los hijos de cada nodo: primer hijo, segundo hijo, etc. Se visualiza dibujando los hijos en orden de izquierda a derecha bajo el padre.
- **Profundidad de un nodo ("depth")**: Es la longitud del camino que va desde ese nodo hasta la raíz (o viceversa). La longitud del camino es cero si el nodo es la raíz. También de forma equivalente: la profundidad es el número de ancestros propios del nodo.
- **Altura de un nodo ("height")**: Es la longitud del mayor de todos los caminos que van desde el nodo hasta una hoja.
- **Altura de un árbol no vacío:** La altura de la raíz.
- Profundidad y altura se han definido como propiedades de nodos. Por tanto, según estas definiciones no tiene sentido hablar de la profundidad o la altura de un árbol vacío (sin nodos).
- La profundidad de un árbol podría definirse como la mayor de las profundidades de las hojas, pero ese valor es igual a la altura. De hecho, la altura de un árbol también se define como la máxima profundidad [CLR'90, p.94].

- **Nivel ('level'):** Conjunto de nodos con la misma profundidad. Así, tenemos desde el nivel 0 hasta el nivel 'h' donde 'h' es la altura del árbol.

Interfaz Tree<E>

Tree<E> es un interfaz pensado para trabajar directamente con posiciones (abstracciones de nodos).

Los TADs de árboles suelen usarse en implementaciones de otros TADs y para ello se necesita poder trabajar directamente (y eficientemente) con nodos. Esto explica, en parte, que no es un interfaz recursivo. Los métodos trabajan con posiciones y no con árboles.

Cada posición tendrá un elemento y un conjunto de posiciones (que pueden verse como posiciones raíz de subárboles).

Hay dos métodos que devuelven un *Iterable<E>:*

- El método *children* devuelve un iterable con los nodos hijo del nodo. Si el árbol está ordenado entonces los nodos en el iterable estarán ordenados.
- El método *positions* es un "*snapshot*".

Algunos métodos lanzan excepciones cuando los nodos son inválidos:

- ***BoundaryViolationException*** será lanzada por *parent* cuando el nodo argumento sea la raíz.
- ***EmptyTreeException*** será lanzada por *root* cuando se invoque dicho método sobre un objeto árbol vacío.
- ***InvalidPositionException*** será lanzada por los métodos que toman posiciones como argumento cuando la posición es inválida (p.ej., cuando no es un objeto de una clase que implementa los nodos del árbol).

Ejemplos de Métodos

Método ancestro

Indica si un nodo es ancestro de otro

```
/**
        * Method ancestro - Indica si un nodo w es ancestro de otro nodo v
        * @param tree - Arbol en el que se va a operar
        * @param w - Posible nodo ancestro
        * @param v - Posible nodo descendiente
        * @return true si w es ancestro de v y falso e.o.c
        * @throws InvalidPositionException - Si las posiciones son invalidas
        */
       public static <E> boolean ancestro (Tree<E> tree, Position<E> w, Position<E> v)
throws InvalidPositionException
       {
              return w == v ||
                           !tree.isRoot(v) && (w == tree.parent(v) ||
ancestro(tree,w,tree.parent(v)));
       }
```

Método ancestro propio

Indica si un nodo es ancestro propio de otro.

```
	/**
	 * Method ancestroPropio - Indica si un nodo w es ancestro propio de otro nodo v
	 * @param tree - Arbol en el que se va a operar
	 * @param w - Posible nodo ancestro
	 * @param v - Posible nodo descendiente
	 * @return true si w es ancestro propio de v y falso e.o.c
	 * @throws InvalidPositionException - Si las posiciones son invalidas o iguales
	 */
public static <E> boolean ancestroPropio (Tree<E> tree, Position<E> w, Position<E> v)
throws InvalidPositionException {

		if (w == v)
			throw new InvalidPositionException();

		return !tree.isRoot(v) && (w == tree.parent(v) ||
ancestro(tree,w,tree.parent(v)));
	}
```

Método Is Sibiling

Indica si dos nodos son hermanos

```
/**
	 * Method isSibiling - Indica si dos nodos son hermanos
	 * @param tree - Arbol en el que se va a operar
	 * @param w - Nodo 1
	 * @param v - Nodo 2
	 * @return true si Nodo 1 y Nodo 2 son hermanos y falso e.o.c
	 * @throws InvalidPositionException - Si las posiciones son invalidas o iguales
	 */
	public static <E> boolean isSibiling (Tree<E> tree, Position<E> w, Position<E> v)
throws InvalidPositionException {

		if (w == v || tree.isRoot(w) || tree.isRoot(v))
			return false;

		else
			return tree.parent(w) == tree.parent(v);
	}
```

Método Leaves

Devuelve una lista con los elementos de las hojas del árbol

```
/**
	 * Mehtod leaves - Devuelve una lista con los elementos de las hojas de un arbol
	 * @param tree - Arbol sobre el que se va a operar
	 * @return una nueva lista con los elementos de las hojas del arbol
	 */
	public static <E> PositionList<E> leaves (Tree<E> tree) {

		PositionList<E> newList = new NodePositionList<E>();

		if (tree.isEmpty()) return newList;

		Iterator <Position<E>> it = tree.positions().iterator();

		while (it.hasNext())
		{
			Position<E> v = it.next();
			if (tree.isExternal(v))
				newList.addLast(v.element());
		}
		return newList;
	}
```

También se puedeimplementar mediante ***for-each***

```
public static <E> PositionList<E> leavesForEach (Tree<E> tree) {

		PositionList<E> newList = new NodePositionList<E>();

		if (tree.isEmpty()) return newList;

		for (Position<E> v : tree.positions())
		{
			if (tree.isExternal(v))
				newList.addLast(v.element());
		}
		return newList;
	}
}
```

Recorridos de árboles generales

Método Depth

Indica la profundidad de un nodo

```
/**
	 * Method depth - Indica la profundidad de un nodo
	 * @param tree - Arbol en el que se va a operar
	 * @param v - Nodo cuya profundidad se desea obtener
	 */
	public static <E> int depth (Tree<E> tree, Position<E> v) throws
InvalidPositionException {

		int count = 0;
		while (!tree.isRoot(v))
		{
			count++;
			v = tree.parent(v);
		}
		return count;
	}


	//Version con for

	public static <E> int depthFor (Tree<E> tree, Position<E> v) throws
InvalidPositionException {

		int count;
		for (count = 0; !tree.isRoot(v);count++,v = tree.parent(v));
		return count;
	}

	//Version recursiva

	public static <E> int depthRec (Tree<E> tree, Position<E> v) throws
InvalidPositionException {

		if (tree.isRoot(v))
			return 0;
		else
		{
			return 1+ depthRec (tree,tree.parent(v));
		}
	}
```

Método Height

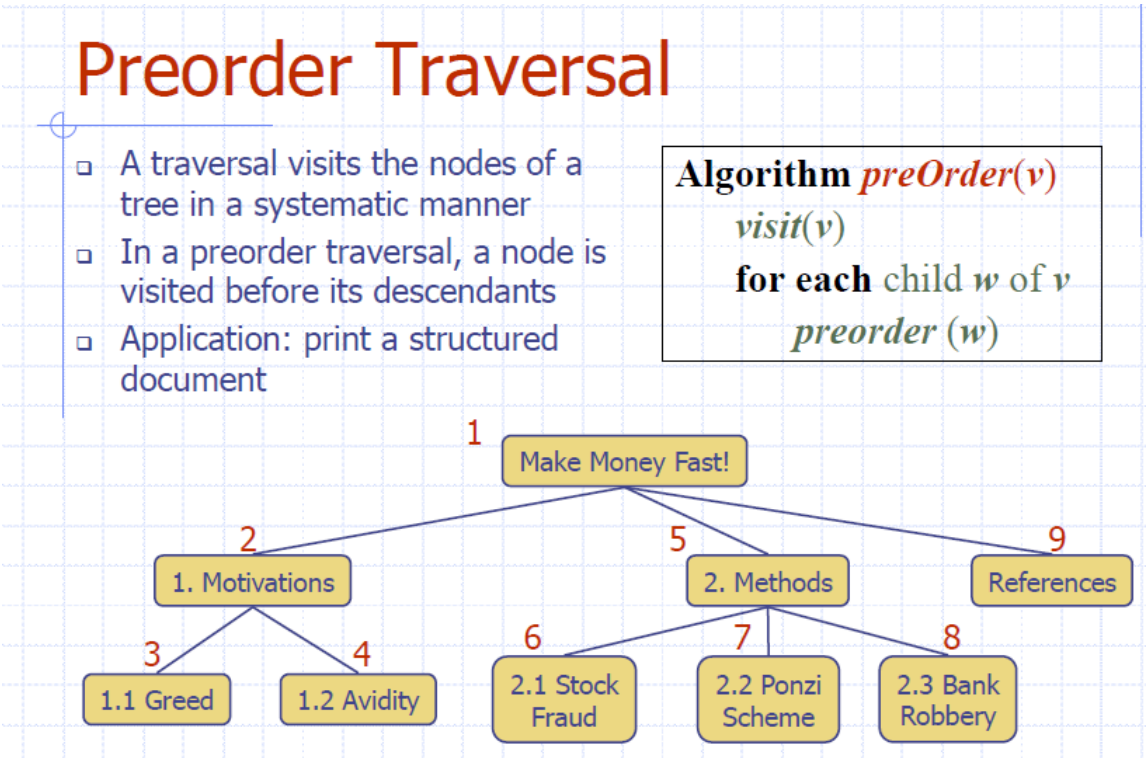
Indica la altura de un nodo

```
/**
	 * Mehtod height - Indica la altura de un nodo
	 * @param tree - Arbol sobre el que se va a operar
	 * @param v - Nodo del que se desea saber la altura
	 * @return la altura del nodo
	 * @throws InvalidPositionException - Si la posicion indicada es invalida
	 */
	public static <E> int height (Tree<E> tree, Position<E> v ) throws
InvalidPositionException {

		int h = 0;
		if (tree.isExternal(v))
			return 0;
		else
			for (Position<E> w : tree.children(v))
				h = Math.max(h, height (tree,w));
		return 1 +h;
	}
```

Recorrido en PreOrden de un árbol

Cuando hablamos de recorrer un árbol en preorden, estamos diciendo que vamos a visitar un nodo y después, sus descendientes.

Explicación gráfica:



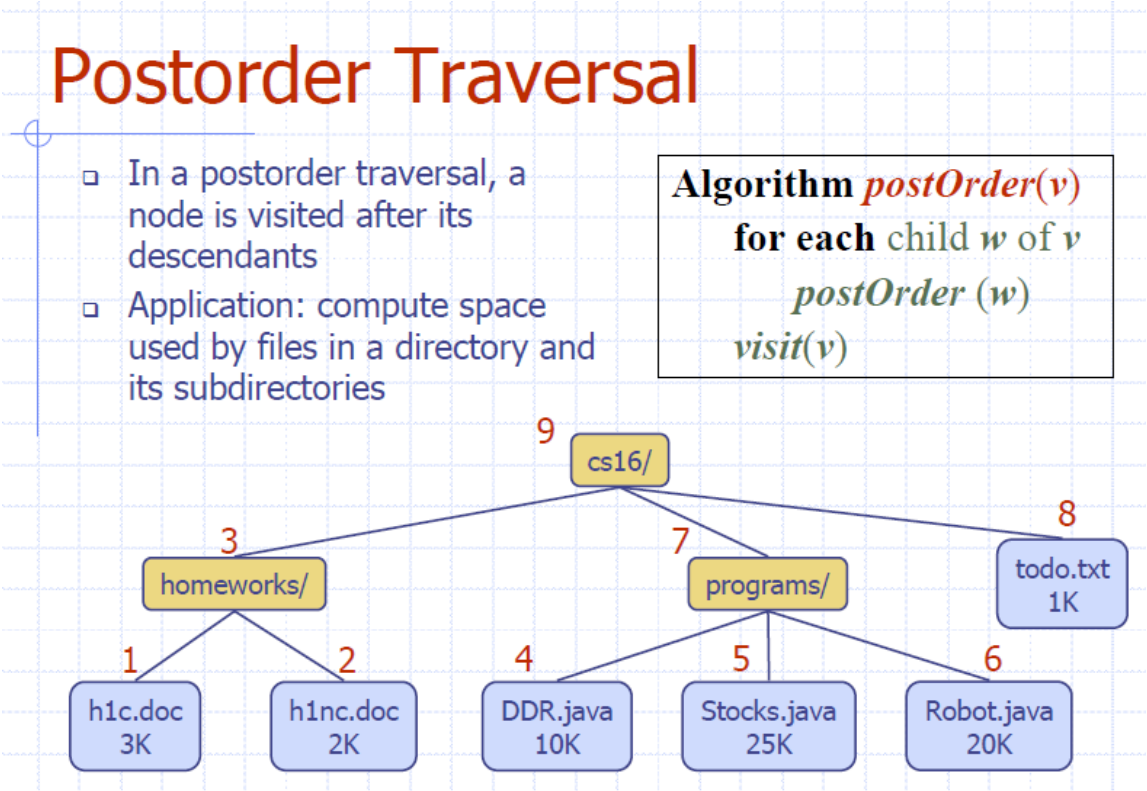
En Java se puede usar un recorrido en preorden para, por ejemplo, mostrar los elementos de un árbol.

```
/**
	 * Method lookPreorder - Muestra el arbol desde el punto seleccionado hasta el
final EN PREORDEN
	 * @param tree - Arbol a mostrar
	 * @param v - Nodo (Posicion) en la que se comienza
	 * @throws InvalidPositionException - Si la posicion indicada es invalida
	 */
	public static <E> void lookPreorder (Tree<E> tree, Position<E> v) throws
InvalidPositionException {

		System.out.println(v.toString());

		for (Position<E> w : tree.children(v))
			lookPreorder (tree, w);
	}
```

Recorrido en PostOrden de un árbol

Cuando hablamos de recorrer un árbol en postorden, estamos diciendo que antes de visitar a un nodo, visitaremos sus descendientes.

Explicación gráfica:



En Java se puede usar un recorrido en postorden para, por ejemplo, mostrar los elementos de un árbol.

```
/**
	 * Method lookPostOrder - Muestra el arbol desde el punto seleccionado hasta el
final EN POSTORDEN
	 * @param tree - Arbol a mostrar
	 * @param v - Nodo (Posicion) en la que se comienza
	 * @throws InvalidPositionException - Si la posicion indicada es invalida
	 */
	public static <E> void lookPostorder (Tree<E> tree, Position<E> v) throws
InvalidPositionException {

		for (Position<E> w : tree.children(v))
		{
			lookPostorder (tree,w);
			System.out.println(w.element());
		}
		System.out.println(v.element());
	}
```

Árboles Binarios

Los árboles binarios son casos especiales de árboles generales, en los que todo nodo tiene **como máximo** dos hijos, el izquierdo, *left child,* y el derecho, *right child.*

Tienen orden!: El izquierdo precede al derecho.

Son árboles ordinarios con grado 2.

Los nodos de estos árboles tienen referencias al padre, al elemento, al hijo izquierdo y al hijo derecho. Se suelen representar los nodos internos como círculos y los externos como cuadrados.

Terminología

- **Definición recursiva:** Un árbol binario es o bien vacío, o bien consiste en un nodo raíz, un subárbol izquierdo y un subárbol derecho.
- **Árbol binario propio o árbol estrictamente binario:** Todo nodo interno tiene dos hijos. Es decir, que todo nodo del árbol tiene o bien 0 hijos (si es externo) o bien 2 hijos (si es interno).
- **Árbol binario impropio:** Árbol binario que no es propio.
- **Un árbol binario perfecto** es aquel que cuenta con el máximo número de nodos posible.
- **Árbol binario equilibrado (balanced binary tree):** para todo nodo, el valor absoluto de la diferencia de altura entre los dos subárboles hijos es como máximo 1. Es decir, para todo nodo 'n' con hijo izquierdo 'i' e hijo derecho 'd' se tiene |h (i)-h (d)| <= 1.

Usos de árboles binarios

Los árboles binarios se pueden usar para implementar TADs, crear árboles de decisión, árboles aritméticos…

Interfaz BinaryTree <E>

Extiende el interfaz Tree <E> con métodos getters e interrogadores para nodos hijos que pueden lanzar excepciones:

- ***BoundaryViolationException*** si un nodo no tiene hijo izquierdo o derecho y se intenta acceder al hijo izquierdo o al derecho, respectivamente.
- ***InvalidPositionException*** si los nodos (posiciones) introducidos son inválidos

Ejemplos de Métodos

Método Height

Indica la altura de un nodo

```
/**
	 * Method height - Indica la altura de un nodo
	 * @param tree - Arbol en el que se va a operar
	 * @param v - Nodo del que se desea saber la altura
	 * @return la altura del nodo
	 * @throws InvalidPositionException - Si la posicion indicada es invalida
	 */
	public static <E> int height (BinaryTree<E> tree, Position<E> v) throws
InvalidPositionException {

		int hLeft = 0,hRight = 0;

		if (tree.isExternal(v)) return 0;
		else
		{
			if (tree.hasLeft(v)) hLeft = height(tree,tree.left(v));
			if (tree.hasRight(v)) hRight = height (tree,tree.right(v));
			return 1 + Math.max(hLeft, hRight);
		}
	}
```

Método Depth

Indica la profundidad de un nodo

```
/**
	 * Metodo depth - Indica la profundidad de un nodo
	 * @param tree - Arbol en el que se va a operar
	 * @param v - Nodo del que se desea saber la profundidad
	 * @return la profundidad del nodo
	 * @throws InvalidPositionException - Si la posicion indicada es invalida
	 */
	public static <E> int depth (BinaryTree<E> tree, Position<E> v) throws
InvalidPositionException {

		if (tree.isRoot(v))
			return 0;
		else
			return 1+depth(tree,tree.parent(v));
	}
```

Recorridos de árboles binarios

Recorrido en PreOrden

```
/**
	 * Method lookPreorder - Muestra los elementos de la lista en PREORDEN
	 * @param tree - Arbol a mostrar
	 * @param v - Posicion desde la cual empezar a mostrar elementos
	 * @throws InvalidPositionException - Si la posicion indicada no es valida
	 */
	public static <E> void lookPreorder(BinaryTree<E> tree, Position<E> v)
			throws InvalidPositionException {

		System.out.println(v.element());

		if (tree.hasLeft(v))  lookPreorder(tree, tree.left(v));
		if (tree.hasRight(v)) lookPreorder(tree, tree.right(v));
	}
```

Recorrido en PostOrden

```
/**
	 * Method lookPostorder - Muestra los elementos de la lista en POSTORDEN
	 * @param tree - Arbol a mostrar
	 * @param v - Posicion desde la cual empezar a mostrar elementos
	 * @throws InvalidPositionException - Si la posicion indicada no es valida
	 */
	public static <E> void lookPostorder(BinaryTree<E> tree, Position<E> v)
			throws InvalidPositionException {

		if (tree.hasLeft(v))  lookPostorder(tree, tree.left(v));
		if (tree.hasRight(v)) lookPostorder(tree, tree.right(v));

		System.out.println(v.element());
	}
```

Recorrido en InOrden

Recorrer un árbol binario en Inorden desde un nodo A implica visitar primero el subárbol izquierdo de A, después A, y después el subárbol derecho de A.

```
/**
	 * Method lookInorder - Muestra los elementos de la lista en INORDEN
	 * @param tree - Arbol a mostrar
	 * @param v - Posicion desde la cual empezar a mostrar elementos
	 * @throws InvalidPositionException - Si la posicion indicada no es valida
	 */
	public static <E> void lookInorder(BinaryTree<E> tree, Position<E> v)
			throws InvalidPositionException {

		if (tree.hasLeft(v))  lookInorder(tree, tree.left(v));

		System.out.println(v.element());

		if (tree.hasRight(v)) lookInorder(tree, tree.right(v));
	}
```

Interfaz BTPosition<E>

Para los *Tree<E>* genéricos utilizábamos el interfaz *Position<E>* que tenía como único método *element()*. Este interfaz, que extiende de *Position<E>* nos proveee además de getters y setters para los nodos hijos izquierdo y derecho, padre y elemento.

Los getters devuelven objetos de tipo *BTPosition<E>.*

Similitud de métodos

No se debe confundir el método *left()* del Interfaz *BinaryTree<E>* con el método *getLeft()* del Interfaz *BTPosition<E>* ya que el segundo devuelve el nodo hijo izquierdo del nodo **sobre el que se invoca el método,** mientras que el primero devuelve el nodo hijo izquierdo del nodo sobre el que se invoca el método, **que está en el árbol sobre el que se invoca el método.**

Ambos devuelven lo mismo, pero no son lo mismo. *Left* lanza excepción si el subárbol izquierdo no existe, mientras que *getLeft* no lo hace, se puede asumir que retorna null en dicho caso. Lo mismo se puede aplicar para *right* y *getRight*, *parent* y *getParent*

Es el mismo caso que ocurría entre *list.next(cursor)* y *cursor.getNext().*

Clase BTNode<E>

Esta clase se encarga de implementar el interfaz *BTPosition<E>*.

Tiene **cuatro atributos:**

- **Referencia al nodo padre.**
- **Referencia al elemento.**
- **Referencia al hijo izquierdo.**
- **Referencia al hijo derecho.**

Tiene **dos constructores:**

- Uno crea un nodo **vacío**, dónde todos los atributos **quedan a null.**
- Otro con los **valores que se le especifiquen.**

Tiene también el método *element()*