TP - Unidad 05-B
Árboles Binarios

Ejercicio 1

Implementar el método buildTree(), invocado desde el constructor del BinaryTree, que permite re construir un árbol binario a partir de un archivo de texto que contiene placeholders y sus datos fueron serializados recorriéndolo por niveles.

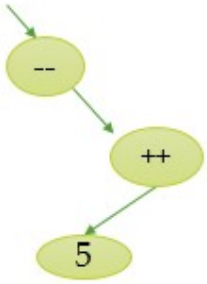
Considerar que el archivo (stream) no se lee a priori completamente en una estructura auxiliar, sino que se lee **de a un token por vez**. Si se crea un nodo y se le coloca el único token actualmente leído no se sabe de antemano si el mismo tendrá o no hijos diferentes a null, porque esa información aparece luego y hay que posponerla (los nodos puede pedir por adelantado requerimientos que serán procesados luego).

Bajar el código desde Campus que considera la idea de posponer acciones por medio de un NodeHelper auxiliar donde el accionar es un enum que resume el requerimiento futuro.

```
static class NodeHelper {

		static enum Action { LEFT, RIGHT, CONSUMIR };


		private Node aNode;
		private Action anAction;

		public NodeHelper(Node aNode, Action anAction ) {
			this.aNode= aNode;
			this.anAction= anAction;
		}


		public Node getNode() {
			return aNode;
		}

		public Action getAction() {
			return anAction;
		}

		}
```

Generar testeos para ver si el árbol se reconstruye correctamente (hay ejemplos de archivos en campus).

Caso de Uso: Dado el árbol



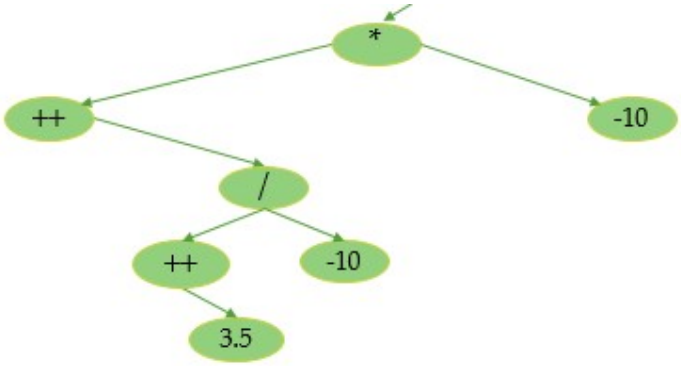
El árbol no es completo, pero si completamos con placeholders y lo recorremos por niveles, tenemos: -- ? ++ ? ? 5

Entonces al reconstruirlo e imprimilo en pre orden debemos obtener:

BinaryTree myTree = **new BinaryTree("data0_1");**
myTree.preOrder();

-- ++ 5

Caso de Uso: Dado el árbol



El árbol no es completo, pero si lo completamos con placeholders y lo recorremos por niveles, tenemos: * ++ -10 ? / ? ? ? ? ++ -10 ? ? ? ? ? ? ? ? ? 3.5

Entonces al reconstruirlo e imprimilo en pre orden debemos obtener:

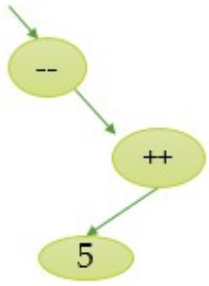
BinaryTree myTree = **new BinaryTree("data0_3");**
myTree.preOrder();

* ++ / ++ 3.5 -10 -10

Ejercicio 2

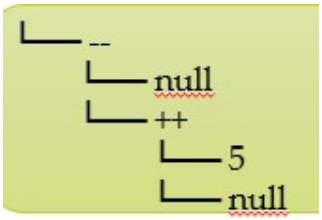
Implementar el método **printHierarchy()** que permita obtener la impresión indentada por niveles.

Caso de Uso: Dado el árbol



BinaryTree rta = **new BinaryTree("data0_1");**
rta.printHierarchy();

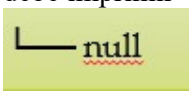
debe imprimir



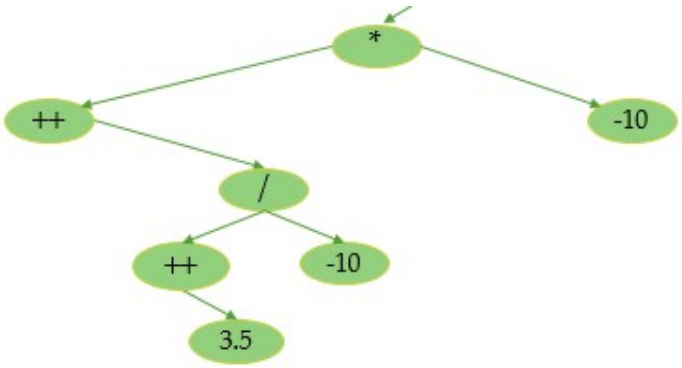
Caso de Uso: Dado el árbol vacío

BinaryTree rta = **new BinaryTree("data0_2");**
rta.printHierarchy();

debe imprimir

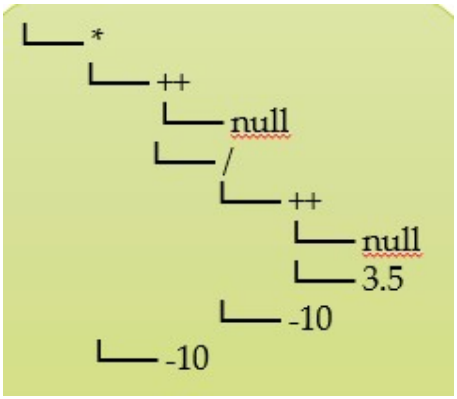


Caso de Uso: Dado el árbol



BinaryTree rta = **new BinaryTree("data0_3");**
rta.printHierarchy();

debe imprimir



¿Qué tipo de recorrido se precisa para obtener este formato?

Ejercicio 3

Re implementar el método builTree() del ejercicio 1, pero ahora **en vez de usar un enum** para encapsular el comportamiento diferido que se quiere producir, hacerlo con una **función lazy que se evaluará a futuro**.

El resultado debe ser el mismo que en el ejercicio 1.

Ejercicio 4

Implementar el método **toFile**(fileName) que dado un BinaryTree almacena sus datos como lo explicado anteriormente, es decir, **generando implicitamente** el completo correspondiente con placeholders y recorriendolo por niveles.

Ejercicio 5

Implementar el método **boolean equals(BinaryTree)** que detecta la equivalencia de árboles binarios.

Utilizarlo para saber si el método **toFile(fileName)** está correctamente implementado.

Ej (suponiendo que los archivos tienen los mismos espacios)

```
BinaryTree original = new BinaryTree("data0_3");
original.toFile("mydata0_3");
BinaryTree copia = new BinaryTree("mydata0_3");
System.out.println(original.equals(copia) );    // true
System.out.println(copia.equals(original) );    // true
System.out.println(original.equals(original) );    // true
System.out.println(copia.equals(copia) );    // true

BinaryTree otro = new BinaryTree("data0_1");
System.out.println(original.equals(otro) );    // false
System.out.println(otro.equals(original) );    // false
```

Ejercicio 6

Escribir la función **getHeight()** que devuelva la altura del árbol binario. Si el árbol está vacío devolver -1. Si tiene solo una raíz devolver 0.

- Implementarlo recursivamente
- Implementarlo iterativamente

Ejercicio 7

Escribir la clase ParametrizedBinaryTree<T> que es la versión que permite parametrizar el tipo de datos data de la clase Node.

Esta versión permite reconstruir un árbol desde un archivo de texto con el formato antes explicado. Lo interesante es que cualquier clase (tipo opaco) que tenga un constructor que acepte generarse a partir de un string (que es lo que se almacenaría en disco), podría ser T.

Caso de Uso: Supongamos que tenemos la clase Jefe que sabe construirse a partir de un String (con cierta convención):

```
public class Jefe  {

	private String name;
	private int edad;

	public Jefe(String value) {
		try {
			String[] arrOfStr = value.split(":");
			if (arrOfStr.length != 2)
				throw new IllegalArgumentException();

			if (! arrOfStr[0].startsWith("(") || ! arrOfStr[1].endsWith(")"))
				throw new IllegalArgumentException();


				name= arrOfStr[0].substring(1);
				edad= Integer.parseInt(arrOfStr[1].substring(0,
arrOfStr[1].length()-1));

		}
		catch(Exception e) {

			name= value;
			edad= Integer.MIN_VALUE;
		}


	}


	@Override
	public String toString() {
		return String.format("(%s:%d)", name, edad);
	}

}
```

Entonces deberíamos poder hacer

```
ParametrizedBinaryTree<Jefe> rta = new ParametrizedBinaryTree<>("jefe", Jefe.class);
rta.printHierarchy();
```

Probarlo con el archivo en Campus jefe

```
└─ (Maria:62)
    └─ (Jose:58)
        └─ (Juana:60)
            └─ null
            └─ (Ariel:35)
        └─ (Patricia:49)
    └─ null
```

Caso de Uso: Si levantamos información que no contiene datos de un jefe, todo depende de cómo se haya implementado el constructor de Jefe: lanza excepción?, lo construye igual?.
Con la implementación de Jefe anterior, al invocar

```
ParametrizedBinaryTree<Jefe> rta2 = new ParametrizedBinaryTree<>("data1_1",
Jefe.class);
rta2.printHierarchy();
```

obtendríamos:

```
└─ (*:-2147483648)
    └─ null
    └─ (vble1:-2147483648)
        └─ (vble2:-2147483648)
        └─ null
```

Caso de Uso: como un String tiene un constructor que acepta String, debe seguir funcionando el árbol de expresiones.

```
ParametrizedBinaryTree<String> rta1 = new ParametrizedBinaryTree<>("data0_3",
String.class);
rta1.printHierarchy();
```

Se obtendría:

```
└─ *
    └─ ++
        └─ null
        └─ /
            └─ ++
                └─ null
                └─ 3.5
            └─ -10
    └─ -10
```