PRÁCTICA A005.- ÁRBOLES BINARIOS

El árbol utilizado en esta práctica tiene nodos vacíos (content==leftSubtree==rightSubtree==null). Es decir el árbol vacío es un nodo vacío. Si un nodo no tiene hijo izquierdo tendrá como leftSubtree un nodo vacío y lo mismo con el hijo derecho.

Ejemplo de método que recorre un árbol con la estructura indicada:

```
public int size () {
        if (! isEmpty()) {
                        return 1 + getLeftBST().size()+ getRightBST().size();
                } else {
                        return 0;
                }
        }
```

PASOS PARA REALIZAR LA PRÁCTICA:

1. **Descargar el proyecto edi_a005_2019** de la página de la asignatura en agora.unileon.es.
2. **Importar** dicho proyecto en Eclipse : Import… **General**…..**Existing proyects into Workspace**… Select archive file… (indicar el archivo ZIP descargado)
3. En este proyecto hay un nuevo paquete:

   ule.edi.tree: con el interface TreeADT<T>, las clases necesarias para implementar árboles generales(AbstractTreeADT), árboles binarios (AbstractBinaryTreeADT), árboles binarios de búsqueda (BinarySearchTreeADTImpl), Entity (entidades en los mundos binarios), etc.

4. Esta práctica tienes dos partes:

   a. **Arboles binarios de búsqueda**: **BinarySearchTreeADTImpl.java** donde se debe implementar el código de los siguientes métodos:

```
/**
 * Árbol binario de búsqueda (binary search tree, BST).
 *
 * El código fuente está en UTF-8, y la constante EMPTY_TREE_MARK definida en AbstractTreeADT del
 * proyecto API debería ser el símbolo de conjunto vacío: ∅
 *
 * Si aparecen caracteres "raros", es porque el proyecto no está bien configurado en Eclipse para
 * usar esa codificación de caracteres.
 *
 * En el toString() que está ya implementado en AbstractTreeADT se usa el formato:
 *
 *              Un árbol vacío se representa como "∅". Un árbol no vacío
 *              como "{(información raíz), sub-árbol 1, sub-árbol 2, ...}".
 *
 *              Por ejemplo, {A, {B, ∅, ∅}, ∅} es un árbol binario con
 *              raíz "A" y un único sub-árbol, a su izquierda, con raíz "B".
 *
 * El método render() también representa un árbol, pero con otro formato;
 *  por ejemplo, un árbol {M, {E, ∅, ∅}, {S, ∅, ∅}} se muestra como:
 *
 * M
 * |  E
 * |  |  ∅
 * |  |  ∅
 * |  S
 * |  |  ∅
 * |  |  ∅
 *
 * Cualquier nodo puede llevar asociados pares (clave,valor) para adjuntar información extra.
```

```
 *  Si es el caso, toString()  mostrarán los pares asociados a cada nodo.
 *
 * Con {@link #setTag(String, Object)} se inserta un par (clave,valor)
 * y con {@link #getTag(String)} se consulta.
 *
 *
 * Con <T extends Comparable<? super T>> se pide que exista un orden en
 * los elementos. Se necesita para poder comparar elementos al insertar.
 *
 * Si se usara <T extends Comparable<T>> sería muy restrictivo; en
 * su lugar se permiten tipos que sean comparables no sólo con exactamente
 * T sino también con tipos por encima de T en la herencia.
 *
 * @param <T>
 *            tipo de la información en cada nodo, comparable.
 */
public class BinarySearchTreeADTImpl<T extends Comparable<? super T>> extends
                AbstractBinaryTreeADT<T> {

        /**
         * Devuelve el árbol binario de búsqueda izquierdo.
         */
        protected BinarySearchTreeADTImpl<T> getLeftBST() {
                //      El atributo leftSubtree es de tipo AbstractBinaryTreeADT<T> pero
                //      aquí se sabe que es además de búsqueda binario
                //
                return (BinarySearchTreeADTImpl<T>) leftSubtree;
        }

        private void setLeftBST(BinarySearchTreeADTImpl<T> left) {
                this.leftSubtree = left;
        }

        /**
         * Devuelve el árbol binario de búsqueda derecho.
         */
        protected BinarySearchTreeADTImpl<T> getRightBST() {
                return (BinarySearchTreeADTImpl<T>) rightSubtree;
        }

        private void setRightBST(BinarySearchTreeADTImpl<T> right) {
                this.rightSubtree = right;
        }

        /**
         * Árbol BST vacío
         */
        public BinarySearchTreeADTImpl() {

                setContent(null);

                setLeftBST(null);
                setRightBST(null);
        }

        private BinarySearchTreeADTImpl<T> emptyBST() {
                return new BinarySearchTreeADTImpl<T>();
        }

        /**
         * Inserta todos los elementos de una colección en el árbol.
         *
         * No se permiten elementos null.
         *
         * @param elements
         *            valores a insertar.
         */
        public void insert(Collection<T> elements) {
                //      O todos o ninguno; si alguno es 'null', ni siquiera se comienza a insertar
                //TODO Implementar el método

        }

        /**
         * Inserta todos los elementos de un array en el árbol.
```

```
         *
         * No se permiten elementos null.
         *
         * @param elements elementos a insertar.
         */
        public void insert(T ... elements) {
                //      O todos o ninguno; si alguno es 'null', ni siquiera se comienza a insertar
            // TODO Implementar el método

        }

        /**
         * Inserta de forma recursiva (como hoja) un nuevo elemento en el árbol de búsqueda.
         *
         * No se permiten elementos null. Si el elemento ya existe en el árbol NO lo inserta.
         *
         * @param element
         *            valor a insertar.
         */
        public void insert(T element) {
                //      No se admiten null
                if (element == null) {
                        throw new IllegalArgumentException("No se aceptan elementos nulos");
                }
                //      TODO Implementar el método
        }



        /**
         * Elimina los elementos de la colección del árbol.
         */
        public void withdraw(Collection<T> elements) {
        //              O todos o ninguno; si alguno es 'null', no se eliminará ningún elemento
            // TODO Implementar el método
        }

        /**
         * Elimina los valores en un array del árbol.
         */
        public void withdraw(T ... elements) {
            //          O todos o ninguno; si alguno es 'null', no se eliminará ningún elemento
            // TODO Implementar el método
        }

        /**
         * Elimina un elemento del árbol.
         *
         * @throws NoSuchElementException si el elemento a eliminar no está en el árbol
         */
        public void withdraw(T element) {
        //      Si el elemento tiene dos hijos, se tomará el criterio de sustituir el elemento
        //   por el mayor de sus menores y eliminar el mayor de los menores.

         // TODO Implementar el método
        }

        /**
         * Devuelve el sub-árbol indicado. (para tests)
         * path será el camino para obtener el sub-arbol. Está formado por 0 y 1.
         * Si se codifica "bajar por la izquierda" como "0" y
         * "bajar por la derecha" como "1", el camino desde un
         * nodo N hasta un nodo M (en uno de sus sub-árboles) será la
         * cadena de 0s y 1s que indica cómo llegar desde N hasta M.
    *
    * Se define también el camino vacío desde un nodo N hasta
    * él mismo, como cadena vacía.
         *
         * Si el subarbol no existe lanzará la excepción NoSuchElementException.
         *
         * @param path
         * @return
         * @throws NoSuchElementException si el subarbol no existe
         */
        public BinarySearchTreeADTImpl<T> getSubtreeWithPath(String path) {
```

```
        //TODO implementar el método

        return null;
    }

    /**
     * Acumula en orden descendente, una lista con los pares 'padre-hijo' en este árbol.
     *
     * Por ejemplo, sea un árbol "A":
     *
     * {10, {5, {2, ø, ø}, ø}, {20, ø, {30, ø, ø}}}
     *
     * el resultado sería una lista de cadenas:
     *
     *      [(20,30), (10,20), (10,5), (5,2)]
     *
     * y además quedaría etiquetado como:
     *
     *  {10 [(descend, 3)],
     *       {5 [(descend, 4)], {2 [(descend, 5)], ø, ø}, ø},
     *       {20 [(descend, 2)], ø, {30 [(descend, 1)], ø, ø}}}
     *
     * @param buffer lista con el resultado.
     */
    public void parentChildPairsTagDescend(List<String> buffer) {

        // TODO Implementar el método
    }



    /**
     * Importante: Solamente se debe recorrer el árbol una vez
     *
     * Comprueba si los elementos de la lista coinciden con algún camino desde la raiz.
     * Además, si existe algún camino que coincida con los elementos de la lista,
     * los etiqueta en el árbol, numerándolos empezando por la raíz como 1.
     *
     * Por ejemplo, el árbol
     *
     * {50, {30, {10, ø, ø}, {40, ø, ø}}, {80, {60, ø, ø}, ø}}
     *
     * si path = [50, 30, 10]
     *
     * devolvería true y el árbol quedaría así etiquetado:
     *
    * {50 [(path, 1)], {30 [(path, 2)], {10 [(path, 3)], ø, ø}, {40, ø, ø}}, {80, {60, ø, ø}, ø}}
     *
     * Para el mismo árbol, si path es [50, 40]  devolvería true
     *     y el árbol quedaría así etiquetado:
     * {50 [(path, 1)], {30, {10, ø, ø}, {40 [(path, 2)], ø, ø}}, {80, {60, ø, ø}, ø}}
     *
     * Para el mismo árbol, si path es [50, 80]  devolvería false y el árbol no se etiqueta:
     * {50, {30, {10, ø, ø}, {40, ø, ø}}, {80, {60, ø, ø}, ø}}
     *
     *
     * @return  true si los elementos de la lista coinciden con algún camino desde la raiz,
     *          falso si no es así
     */
    public boolean isPathIn(List<T> path) {
        //      TODO Implementar método
        return false;
    }


    /**
     *
     * Etiqueta cada nodo con su posición en el recorrido en anchura, con la etiqueta "width"
     *
     *  Por ejemplo, el árbol
     *
     * {50, {30, {10, ø, ø}, {40, ø, ø}}, {80, {60, ø, ø}, ø}}
     *
     *  queda etiquetado como
     *
```

```
         *   {50 [(width, 1)],
                {30 [(width, 2)], {10 [(width, 4)], ø, ø},{40 [(width, 5)], ø, ø}},
                {80 [(width, 3)], {60 [(width, 6)], ø, ø}, ø}}
         *
         */
        public void tagWidth(){
                //      TODO Implementar método
        }


        /**
         * Devuelve un iterador que recorre los elementos del arbol en inorden (de menor a mayor)
         *
         * Por ejemplo, con el árbol
         *
         *              {50, {30, {10, ø, ø}, {40, ø, ø}}, {80, {60, ø, ø}, ø}}
         *
         * y devolvería el iterador que recorrería los ndos en el orden: 10, 30, 40, 50, 60, 80
         *
         * @return iterador para el recorrido inorden o ascendente
         */
        public Iterator<T> iteratorInorden() {
                //      TODO Implementar método
                return null;
        }
}
```

b. World.java: donde se implementarán los siguientes métodos:

```
/**
 * Un mundo es un árbol binario.
 * En cada nodo de un mundo se almacena una lista de entidades, cada una con su tipo y
 * cardinalidad. Ver {@link Entity}.
 *
 * Si se codifica "bajar por la izquierda" como "0" y  "bajar por la derecha" como "1", el camino desde un
 * nodo N hasta un nodo M (en uno de sus sub-árboles) será la cadena de 0s y 1s que indica cómo llegar
 * desde N hasta M.
 *
 * Se define también el camino vacío desde un nodo N hasta  él mismo, como cadena vacía.
 *
 * Por ejemplo, el mundo:
 *
 * {[F(1)], {[F(1)], {[D(2), P(1)], ø, ø}, {[C(1)], ø, ø}}, ø}
 *
 * o lo que es igual:
 *
 * [F(1)]
 * |  [F(1)]
 * |  |  [D(2), P(1)]
 * |  |  |  ø
 * |  |  |  ø
 * |  |  [C(1)]
 * |  |  |  ø
 * |  |  |  ø
 * |  ø
 *
 * contiene un bosque (forest) en "", otro en "0", dos dragones y una princesa en "00" y un castillo en "01".
 * @param <T>
 *
 */
public class World extends AbstractBinaryTreeADT<LinkedList<Entity>> {

        /**
         * Devuelve el mundo al que se llega al avanzar a la izquierda.
         *
         * @return
         */
        protected World travelLeft() {
                return (World) leftSubtree;
        }

        private void setLeft(World left) {
```

```
                this.leftSubtree = left;
        }

        /**
         * Devuelve el mundo al que se llega al avanzar a la derecha.
         *
         * @return
         */
        protected World travelRight() {
                return (World) rightSubtree;
        }

        private void setRight(World right) {
                this.rightSubtree = right;
        }

        private World() {
                //      Crea un mundo vacío
                // TODO
        }

        public static World createEmptyWorld() {
                return new World();
        }

        /**
         * Inserta la entidad indicada en este árbol.
         *
         * La inserción se produce en el nodo indicado por la dirección; todos los nodos recorridos para
         *   alcanzar aquel, que no estén creados se inicializarán con una entidad 'forest'.
         *
         * La dirección se supondrá correcta, compuesta de cero o más 0s y 1s.
         *
         * Dentro de la lista del nodo indicado, la inserción de nuevas entidades
         * se realizará al final, como último elemento.
         *
         * Por ejemplo, en un árbol vacío se pide insertar un 'dragón' en la dirección "00".
         *  El resultado final será:
         *
         * [F(1)]
         * |  [F(1)]
         * |  |  [D(1)]
         * |  |  |  ø
         * |  |  |  ø
         * |  |  ø
         * |  ø
         *
         * La dirección "" indica la raíz, de forma que insertar un 'guerrero' en
          * "" en el árbol anterior genera:
    *
    * [F(1), W(1)]
    * |  [F(1)]
    * |  |  [D(1)]
    * |  |  |  ø
    * |  |  |  ø
    * |  |  ø
    * |  ø
    *
    * La inserción tiene en cuenta la cardinalidad, de forma que al volver a
    * insertar un guerrero en "" se tiene:
    *
    * [F(1), W(2)]
    * |  [F(1)]
    * |  |  [D(1)]
    * |  |  |  ø
    * |  |  |  ø
    * |  |  ø
    * |  ø
    *
         * @param address dirección donde insertar la entidad.
         * @param e entidad a insertar.
         */
        public void insert(String address, Entity e) {

                //TODO implementar el metodo
```

```
	}

	/**
	 * Indica cuántas entidades del tipo hay en este nodo.
	 *
	 * @param type tipo de entidad.
	 * @return cuántas entidades de ese tipo hay en este nodo.
	 */
	public long countEntityNode(int type) {
		// TODO Implementar el método
		return 0;
	}


	/**
	 * Indica cuántas entidades del tipo hay en este mundo (en el árbol completo).
	 *
	 * @param type tipo de entidad.
	 * @return cuántas entidades de ese tipo hay en este árbol.
	 */
	public long countEntity(int type) {
		// TODO Implementar el método
		return 0;
	}


	/**
	 * Calcula el número de princesas accesibles que hay en este mundo situadas en la altura h,
	 * e introduce en una lista las referencias a los nodos en las que se encuentran.
	 *
	 * Una princesa es accesible si en el camino desde la raiz hasta ella no aparece ningún Dragón
	 *
	 * @param List<World> donde dejará las referencias a los nodos situados en altura h
	 *            que contienen princesas accesibles.
	 * @return el número de princesas accesibles situadas a altura h
	 */
	public long countAccesiblePrincesHeight(int h, List<World> lista){
		// TODO implementar el método
	return 0;
	}
```

5. A la vez que se van desarrollando los métodos propuestos en las clases **BinarySearchTreeADTImpl.java** y **world.java**, se deben crear los correspondientes métodos de prueba JUnit 4 para ir comprobando su correcto funcionamiento.
6. Además **se deberá entregar en agora.unileon.es la versión final de la práctica** (proyecto exportado como zip) .

<u>NOTA IMPORTANTE</u>: NO SE PUEDE MODIFICAR LA ESTRUCTURA DE DATOS DE LOS ATRIBUTOS DEFINIDOS EN LAS CLASES DEL PROYECTO (HAY QUE UTILIZAR LAS ESTRUCTURAS DE DATOS INDICADAS EN LAS CLASES DEL PROYECTO edi-a005-2019)

FECHA LIMITE de entrega de la práctica A005-2019: 1 de JUNIO de 2019