PRÁCTICA A005.- ÁRBOLES BINARIOS

El árbol binario utilizado en esta práctica tiene nodos vacíos (content==null; leftSubtree==null y rightSubtree==null). Es decir el árbol vacío es un nodo vacío. Si un nodo no tiene hijo izquierdo tendrá como l**eftSubtree 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. Revisar documento **EstructurasDatos_Practica5_2020.pdf.**
2. **Descargar el proyecto edi_a005_2020** de la página de la asignatura en agora.unileon.es.
3. **Importar** dicho proyecto en Eclipse : Import… **General**…..**Existing proyects into Workspace**… Select archive file… (indicar el archivo ZIP descargado)
4. En este proyecto hay un paquete:
    a. **ule.edi.tree: c**on el interface **TreeADT<T>, l**as clases necesarias para implementar árboles generales**(AbstractTreeADT),** árboles binarios **(AbstractBinaryTreeADT),** árboles binarios de búsqueda **(BinarySearchTreeImpl),** Entity **(entidades en los mundos binarios), etc.**
        i. **TreeADT:** interface con operaciones de gestión de un árbol general.
        ii. **AbstractTreeADT:** clase abstracta que implementa un árbol general.
        iii. **AbstractBinaryTree:** clase abstracta que implementa un **árbol binario**, con subárbol izquierdo (leftSubtree) y subárbol derecho (rightSubtree).
5. Esta práctica tiene dos partes (una obligatoria y otra optativa):
    a. **(OBLIGATORIA) Arboles binarios de búsqueda**:
    BinarySearchTreeImpl.java donde se debe implementar el código de los siguientes métodos:

    NOTA: En las operaciones de insertar y eliminar varios elementos, si alguno de los elementos es nulo NO SE INSERTARÁ O ELIMINARÁ NINGUNO.

        i. **boolean insert(T element)**: inserta un elemento en un árbol binario de búsqueda teniendo en cuenta que para cada nodo, todos los elementos del subarbol izquierdo deben ser menores que el elemento contenido en el nodo y todos los elementos del subarbol izquierdo deben ser mayores que el elemento contenido en el nodo. **Al insertar el nodo en el árbol debe asignarse valor a su atributo father (referencia a su nodo padre).**

Devuelve true si se pudo insertar (no existia ese elemento en el arbol, false en caso contrario)

ii. **int insert(T … elements)**: inserta los elementos de un array en un árbol binario de búsqueda. **Devuelve el numero de elementos insertados en el arbol (los que ya están no los inserta). Si algún elemento es null ; no inserta ninguno.**

iii. **int insert(Collection<T> elements)**: inserta los elementos de una colección en un árbol binario de búsqueda. **Devuelve el número de elementos insertados en el arbol (los que ya están no los inserta). Si algún elemento es null ; no inserta ninguno.**

iv. **boolean contains(T element):** indica si el elemento está en el árbol o no.

v. **remove(T element)**: elimina un elemento en un árbol binario de búsqueda.

Lanza la excepción **NoSuchElementException** si el elemento **a eliminar** no está en el árbol.

Para **eliminar un elemento del árbol**, se debe buscar y si se encuentra se comprueba en qué caso estaría:

- **Es una hoja**: se vacía el nodo.

- **Tiene un hijo**: su hijo ocupa su lugar. Para ello se copian los atributos de su hijo en el nodo que contiene el elemento a borrar

- **Tiene dos hijos**: se busca el **MENOR DE SUS DESCENDIENTES MAYORES** (los contenidos en su subarbol derecho), es decir, el siguiente en el árbol de búsqueda; se sustituye el contenido del nodo a eliminar por el MENOR DE SUS MAYORES, y se llama a eliminar el menor de sus mayores **en su rama derecha**.

vi. void **remove(T … elements)**: elimina los elementos de un array en un árbol binario de búsqueda. Comprueba que se puedan eliminar todos y sino no elimina ninguno disparando la excepción NoSuchElementException.

vii. **Iterator<T> iteratorwidth():** Devuelve un iterador que recorre los elementos del árbol por niveles según el recorrido en anchura.

Por ejemplo, con el árbol {50, {30, {10, ∅, ∅}, {40, ∅, ∅}}, {80, {60, ∅, ∅}, ∅}

devolvería el iterador que recorrería los nodos en el orden: 50, 30, 80, 10, 40, 60

Puede implementarse creando una lista con el recorrido en anchura de los elementos del árbol y devolver el iterador de dicha lista.

viii. **void tagDecendents():** Etiqueta cada nodo con el valor correspondiente al número de descendientes que tiene en este árbol.

Importante: Solamente se debe recorrer el árbol una vez

Por ejemplo, sea un árbol "A":

{10, {5, {2, ∅, ∅}, ∅}, {20, {15, ∅, ∅}, {30, ∅, ∅}}}

10
| 5
| | 2
| | | ∅
| | | ∅
| | ∅
| 20
| | 15
| | | ∅
| | | ∅
| | 30
| | | ∅
| | | ∅

El árbol quedaría etiquetado como:

{10 [(decendents, 5)],
{5 [(decendents, 1)], {2 [(decendents, 0)], ∅, ∅}, ∅},
{20 [(decendents, 2)], {15 [(decendents, 0)], ∅, ∅}, {30 [(decendents, 0)], ∅, ∅}}}

ix. **int tagOnlySonInorder():** Calcula el número de nodos que son hijos únicos **y** etiqueta cada nodo que sea hijo único (no tenga hermano hijo del mismo padre) con el número correspondiente a su posición según el recorrido inorden en este árbol. Si el árbol tiene 3 hijos únicos los va etiquetando según su recorrido en inorden. La raíz no se considera hijo único.

Importante: Solamente se debe recorrer el árbol una vez

Por ejemplo, sea un árbol aEjemplo:

{10, {5, {2, ∅, ∅}, ∅}, {20, {15, {12, ∅, ∅}, ∅}, ∅}}

Hay 3 nodos hijos únicos, recorriendo el árbol en inorden aparecen en este orden: 1º->2, 2º->12, 3º->15

Después de la llamada a aEjemplo.tagOnlySonInorder(), quedará:

{10, {5, {2 [(onlySon, 1)], ∅, ∅}, ∅}, {20, {15 [(onlySon, 3)], {12 [(onlySon, 2)], ∅, ∅}, ∅}, ∅}}

x. **void tagHeight() :** etiqueta cada nodo con la etiqueta "height" y su altura.

Importante: Solamente se debe recorrer el árbol una vez

Por ejemplo, sea un árbol aEjemplo:

{10, {5, {2, ∅, ∅}, ∅}, {20, {15, {12, ∅, ∅}, ∅}, ∅}}

Después de la llamada a aEjemplo.tagHeight() ,quedará:

{10 [(height, 1)], {5 [(height, 2)], {2 [(height, 3)], ∅, ∅}, ∅}, {20 [(height, 2)], {15 [(height, 3)], {12 [(height, 4)], ∅, ∅}, ∅}, ∅}}

b. [OPTATIVO- PARA SUBIR NOTA] World.java: donde se implementarán arboles binarios cuyos nodos representan mundos:

Un mundo es un árbol binario con las siguientes características:

- En cada nodo de un mundo se almacena una **lista de entidades** (o personajes), cada una **con su tipo y cardinalidad**. El tipo de las entidades está definido en la clase Entity, a través de 6 constantes (UNKNOWN=0; DRAGON=1; PRINCESS=3; WARRIOR=5; CASTLE=7; FOREST=9) y la **cardinalidad indica número de entidades** de ese tipo que hay en ese nodo en concreto.
- Si se codifica "bajar por la izquierda" como "0" y "bajar por la derecha" como "1", la dirección de un nodo, representa el camino desde la raíz hasta ese nodo y será la cadena de 0s y 1s que indica cómo llegar desde la raíz hasta ese nodo.
- Se define también el camino vacío desde un nodo N hasta él mismo, como cadena vacía. Por ejemplo, el mundo contiene un bosque (FOREST) en "" (la raíz), otro en "0" (hijo izquierdo de la raiz) , dos dragones y una princesa en "00" (hijo izquierdo del hijo izquierdo de la raíz) y un castillo en "01" (hijo derecho del hijo izquierdo de la raíz).

  {[F(1)], {[F(1)], {[D(2), P(1)], ∅, ∅}, {[C(1)], ∅, ∅}}, ∅} o lo que es igual:

  ```
  [F(1)]
  | [F(1)]
  | | [D(2), P(1)]
  | | | ∅
  | | | ∅
  | | [C(1)]
  | | | ∅
  | | | ∅
  | ∅
  ```

- Los métodos a implementar en esta parte son:

  1. **public void insert(String address, Entity e):** inserta una entidad en un árbol, en la posición que indique address. El parámetro **address** es una secuencia de 0's y 1's que marca el camino para llegar al nodo donde insertar la entidad. Si address es la cadena vacía indica que hay que insertarlo en la raíz. El 0 indica bajar por la izquierda y el 1 indica bajar por la derecha.

     La inserción se produce en el nodo indicado por address; todos los nodos recorridos para alcanzar aquel, que no estén creados se crearán e inicializarán con una entidad '**UNKNOWN**'. La dirección se **supondrá correcta**, compuesta de cero o más 0's y 1's.
     Dentro de la lista del nodo indicado, la inserción de nuevas entidades se realizará al final, como último elemento de la lista de entidades.
     - Por ejemplo, en un árbol vacío se pide insertar un '**dragón**' en la dirección "00". El resultado final será: {[U(1)], {[U(1)] , {[D(1)], ∅, ∅}, ∅}, ∅}
     - La dirección "" indica la raíz, de forma que insertar un '**guerrero**' en"" en el árbol anterior genera: {[U(1), W(1)], {[U(1)] , {[D(1)], ∅, ∅}, ∅}, ∅}

- La inserción tiene en cuenta la cardinalidad, de forma que al volver a insertar un guerrero en "" se tiene: {[U(1), W(2)], {[U(1)] , {[D(1)], ∅, ∅}, ∅}, ∅}

2. **public long countEntity(int type)**: Indica cuántas entidades del tipo hay en este mundo (en el árbol completo). El parámetro tope es el tipo de entidad.

3. **public long countAccesiblePrincess(List<String> lista):** indica cuántas princesas accesibles hay en el árbol. Además introduce en una lista de Strings los direcciones a los nodos en las que se encuentran dichas princesas.

Se considera princesa accesible :

- a toda princesa que en su camino desde la raíz hasta el nodo que la contiene no hay ningún dragón,
- o si hay algún dragón en el camino desde la raíz hasta la princesa, se cumpla que desde el último dragón hasta la princesa haya al menos un castillo que la proteja (podrían estar en el mismo nodo, tanto la princesa, como el castillo y el dragón).

Ejemplo: Dado el árbol arbol1, siendo arbol1.toString() ={[U(1)], {[U(1)], ∅, {[D(3)], {[P(4)], ∅, ∅}, ∅}}, {[U(1)], {[P(7)], ∅, ∅}, {[C(1), D(1)], ∅, {[P(1)], ∅, ∅}}}}

(se ha insertado: un dragón y un castillo en "11", 7 princesas en "10", un dragón en "01", 4 princesas en "010" y 1 princesa en "111")

Al llamar a arbol1.countAccesiblePrincess(lista) siendo lista una lista vacía, devolverá 8 y la lista contendrá ("10", "111").

6. A la vez que se van desarrollando las clases anteriores **se deben crear las correspondientes métodos de pruebas JUnit 4 (**cuyo nombre debe acabar en Test) para ir comprobando su correcto funcionamiento.
7. Se deberá entregar en agora.unileon.es la versión final de la práctica. **Para ello primero habrá que exportar el proyecto edi-a005-2020 como zip (Export… General… Archive File)**

FECHA LIMITE de entrega de la práctica A005-2020 : 5 de junio a las 23:59