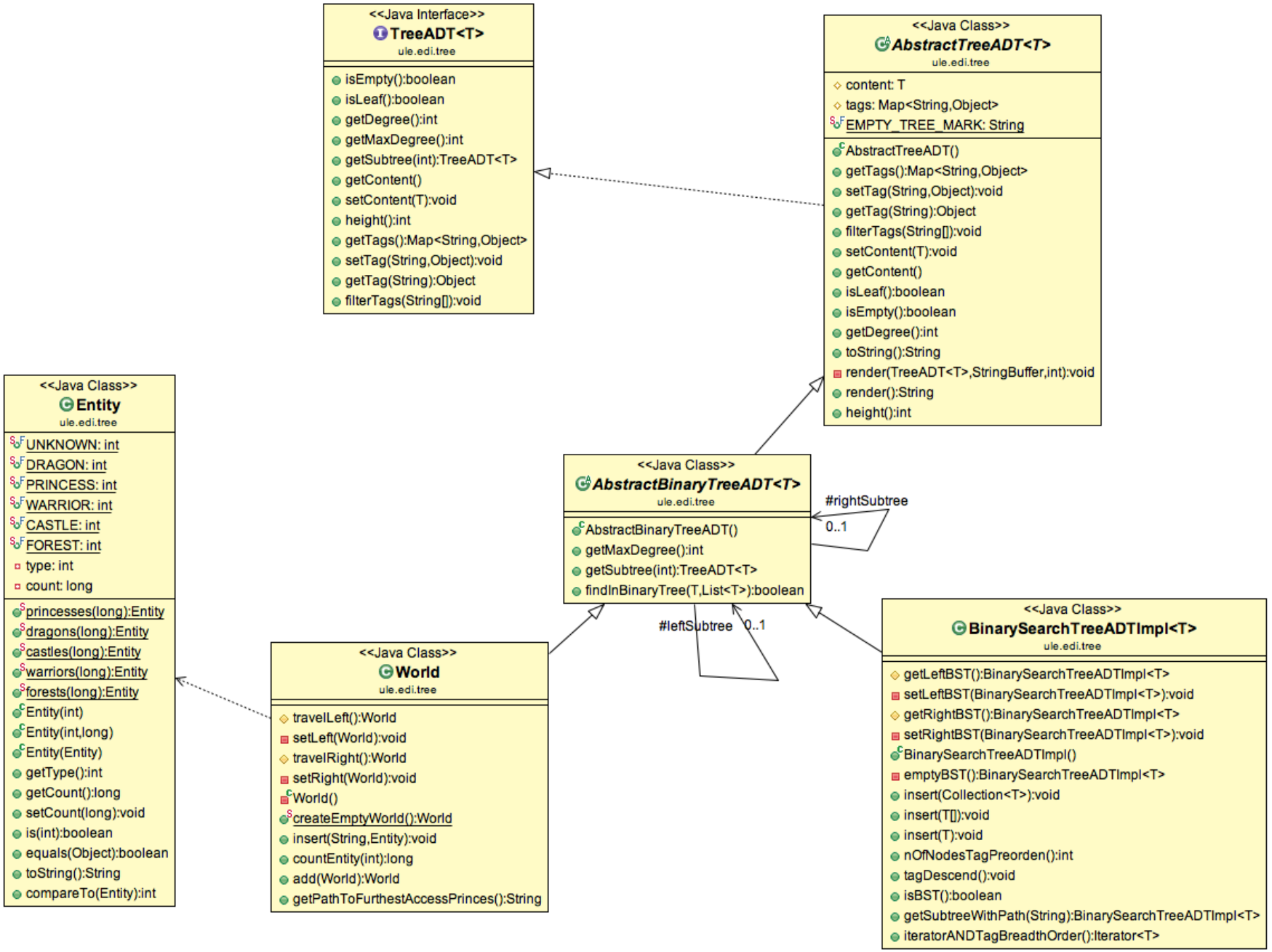
PRÁCTICA 5. - ÁRBOLES

- Árboles binarios de búsqueda
- Arboles que representan mundos

En los dos tipos de árboles se seguirán las siguientes consideraciones:

- El árbol vacío es un nodo vacío (content, left y right a null)
- Si un nodo no tiene hijo izquierdo, su hijo izquierdo apuntará a un nodo vacío
- Si un nodo no tiene hijo derecho, su hijo derecho apuntará a un nodo vacío


<<Java Interface>>
TreeADT<T>
ule.edi.tree
isEmpty():boolean
isLeaf():boolean
getDegree():int
getMaxDegree():int
getSubtree(int):TreeADT<T>
getContent()
setContent(T):void
height():int
getTags():Map<String,Object>
setTag(String,Object):void
getTag(String):Object
filterTags(String[]):void
<<Java Class>>
AbstractTreeADT<T>
ule.edi.tree
content: T
tags: Map<String,Object>
EMPTY_TREE_MARK: String
AbstractTreeADT()
getTags():Map<String,Object>
setTag(String,Object):void
getTag(String):Object
filterTags(String[]):void
setContent(T):void
getContent()
isLeaf():boolean
isEmpty():boolean
getDegree():int
toString():String
render(TreeADT<T>,StringBuffer,int):void
render():String
height():int
<<Java Class>>
Entity
ule.edi.tree
UNKNOWN: int
DRAGON: int
PRINCESS: int
WARRIOR: int
CASTLE: int
FOREST: int
type: int
count: long
princesses(long):Entity
dragons(long):Entity
castles(long):Entity
warriors(long):Entity
forests(long):Entity
Entity(int)
Entity(int,long)
Entity(Entity)
getType():int
getCount():long
setCount(long):void
is(int):boolean
equals(Object):boolean
toString():String
compareTo(Entity):int
<<Java Class>>
AbstractBinaryTreeADT<T>
ule.edi.tree
AbstractBinaryTreeADT()
getMaxDegree():int
getSubtree(int):TreeADT<T>
findInBinaryTree(T,List<T>):boolean
#rightSubtree
0..1
#leftSubtree
0..1
<<Java Class>>
World
ule.edi.tree
travelLeft():World
setLeft(World):void
travelRight():World
setRight(World):void
World()
createEmptyWorld():World
insert(String,Entity):void
countEntity(int):long
add(World):World
getPathToFurthestAccessPrinces():String
<<Java Class>>
BinarySearchTreeADTImpl<T>
ule.edi.tree
getLeftBST():BinarySearchTreeADTImpl<T>
setLeftBST(BinarySearchTreeADTImpl<T>):void
getRightBST():BinarySearchTreeADTImpl<T>
setRightBST(BinarySearchTreeADTImpl<T>):void
BinarySearchTreeADTImpl()
emptyBST():BinarySearchTreeADTImpl<T>
insert(Collection<T>):void
insert(T[]):void
insert(T):void
nOfNodesTagPreorden():int
tagDescend():void
isBST():boolean
getSubtreeWithPath(String):BinarySearchTreeADTImpl<T>
iteratorANDTagBreadthOrder():Iterator<T>

AbstractTreeADT<T>

- **Atributos:**
  - content: T
  - **tags:Map<String,Object>**
    - (creado como un HashMap<>())
    - Map es un interface y HashMap es una clase que implementa el interfaz Map con tablas hash.
- **Métodos relevantes**:
  - **setTag(String,Object)**: etiqueta el nodo actual con la etiqueta pasada como primer parámetro. (Añade al HashMap tags el par (String,Object)
    - setTag("level", nivel)
  - **getTag(String)**: setTag(String,Object): devuelve el valor almacenado junto a la clave pasada como parámetro.
    - getTag("level")
  - **filterTags(String… labels):** deja solamente las etiquetas indicadas. Muy util para los tests (en cada método, filtrar solamente las etiquetas que se piden en ese método)
    - filterTags("level")

Interface Map

```
1  public interface Map<K,V> {
2
3  // Basic operations
4
5  V put(K key, V value);
6
7  V get(Object key);
8
9  V remove(Object key);
10
11 boolean containsKey(Object key);
12
13 boolean containsValue(Object value);
14
15 int size();
16
17 boolean isEmpty();
18
19 // Bulk operations
20
21 void putAll(Map<? extends K, ? extends V> m);
22
23 void clear();
24
25 // Collection Views
26
27 public Set<K> keySet();
28
29 public Collection<V> values();
30
```

HashMap

- **Método put(K key, V value)**: inserta el par (key,value) en la colección. Si la key ya existe se sobreescribe, ya que no puede haber elementos con la misma key.
- **Método V get(K key)**: devuelve la clave almacenada junto al key pasado como parámetro.

AbstractBinaryTreeADT<T>

- Atributos:
  - AbstractBinaryTreeADT<T> leftSubtree
  - AbstractBinaryTreeADT<T> rightSubtree

BinarySearchTreeADTImpl<T extends Comparable<? super T>> extends AbstractBinaryTreeADT<T>

- No define nuevos atributos
- El tipo de los elementos del árbol tiene que ser comparable (implementar el interface comparable o tener en su jerarquía de herencia algún antepasado que lo implemente).

Entity

- Implementa el interface Comparable<Entity>
- Atributos:
  - int type: tipo definido por constantes:
  - long count: nº de intancias de ese tipo

```
UNKNOWN = 0;

DRAGON = 1;

PRINCESS = 3;

WARRIOR = 5;

CASTLE = 7;

FOREST = 9;
```

World **extends** **AbstractBinaryTreeADT<LinkedList<Entity>>**

- Es un árbol binario (no de búsqueda).
- Los elementos del árbol son listas de entidades.
- Por ejemplo el mundo:

{[F(1)], {[F(1)], {[D(2), P(1)], ∅, ∅}, {[C(1)], ∅, ∅}}, ∅}