Ejercicio 1 [3.0 puntos].

1.1: **[1.5 puntos]** Dada la secuencia: 60, 3, 12, 23, 51, 80, 14, 59, 15, 9, 13, 1; representar gráficamente el árbol AVL y el árbol Rojo-Negro resultante.

1.2: **[1.5 puntos]** Dada la secuencia de claves enteras: 190, 57, 89, 90, 121, 170, 35, 48, 91, 22, 126, 132 y 80; representar gráficamente el árbol B de orden 5 resultante. Posteriormente, elimine la clave 91 y la clave 48 (representando en ambos casos los árboles resultantes).

NOTA: Para que los ejercicios sean evaluados con la máxima calificación, es necesario representar los *pasos intermedios relevantes*. Presentar únicamente el resultado final equivale a una calificación de cero en el ejercicio correspondiente.

Ejercicio 2 [3.5 puntos]

Se desea aumentar la funcionalidad de los árboles binarios incorporando la siguiente funcionalidad:

a) **[1 punto]** Se dice que un árbol binario es perfecto si todos los nodos internos tienen dos hijos. Implementar un método `isPerfect`, que determine si un árbol binario es perfecto o no.

b) **[1 punto]** Se dice que un árbol binario es zurdo si es vacío, es una hoja o más de la mitad de sus descendientes están en el hijo izquierdo. Implementar el método `isOdd`, que determina si un árbol binario es zurdo o no.

c) **[1.5 puntos]** `InternalNodeIterator`: clase que permite recorrer los nodos internos de un árbol binario.

NOTA: Se valorará un buen diseño (de clases y métodos) para obtener la máxima calificación. Se recuerda que, por lo general, es un mal diseño tener métodos que compartan mucho código. Asimismo, es imprescindible que los métodos implementados sean lo más *eficientes y genéricos* posible.

Ejercicio 3 [3.5 puntos]

La empresa *URJC Investment* ha adquirido recientemente la Torre Europa en Madrid. Su intención es alquilar las oficinas a distintas empresas para poder sacar rendimiento a la inversión. Para facilitar el trabajo del personal de recepción, esta empresa quiere desarrollar un módulo en Java que permita buscar con facilidad información de las trabajadores de las empresas que tienen sus oficinas en la Torre Europa.

Esta aplicación permitirá localizar, por el nombre, cada una de las empresas. Además, cada empresa tendrá almacenado su organigrama de empleados, que refleja la posición de las áreas que la integran, sus niveles jerárquicos o líneas de autoridad y de asesoría. Es decir, cada empresa tiene un único CEO, que es el jefe de los directivos de máximo nivel. A su vez, cada trabajador depende de un único jefe. Finalmente, por simplicidad, se considera que el nombre de los trabajadores es único.

Cada trabajador se modelará con una clase `Employee` que debe contener, como mínimo, la siguiente información: Empresa, Nombre del empleado, Cargo que ocupa en la empresa y Descripción personal. Asimismo, cada empresa se modelará con una clase `OrganizationChart` (organigrama) que debe almacenar los datos de una empresa concreta.

Se pide:

a) **[0.5 puntos]** Definir el tipo de datos (clases en Java) `URJCInvest`, `OrganizationChart` y `Employee`, de tal forma que la complejidad algorítmica de las operaciones requeridas sea la menor posible.

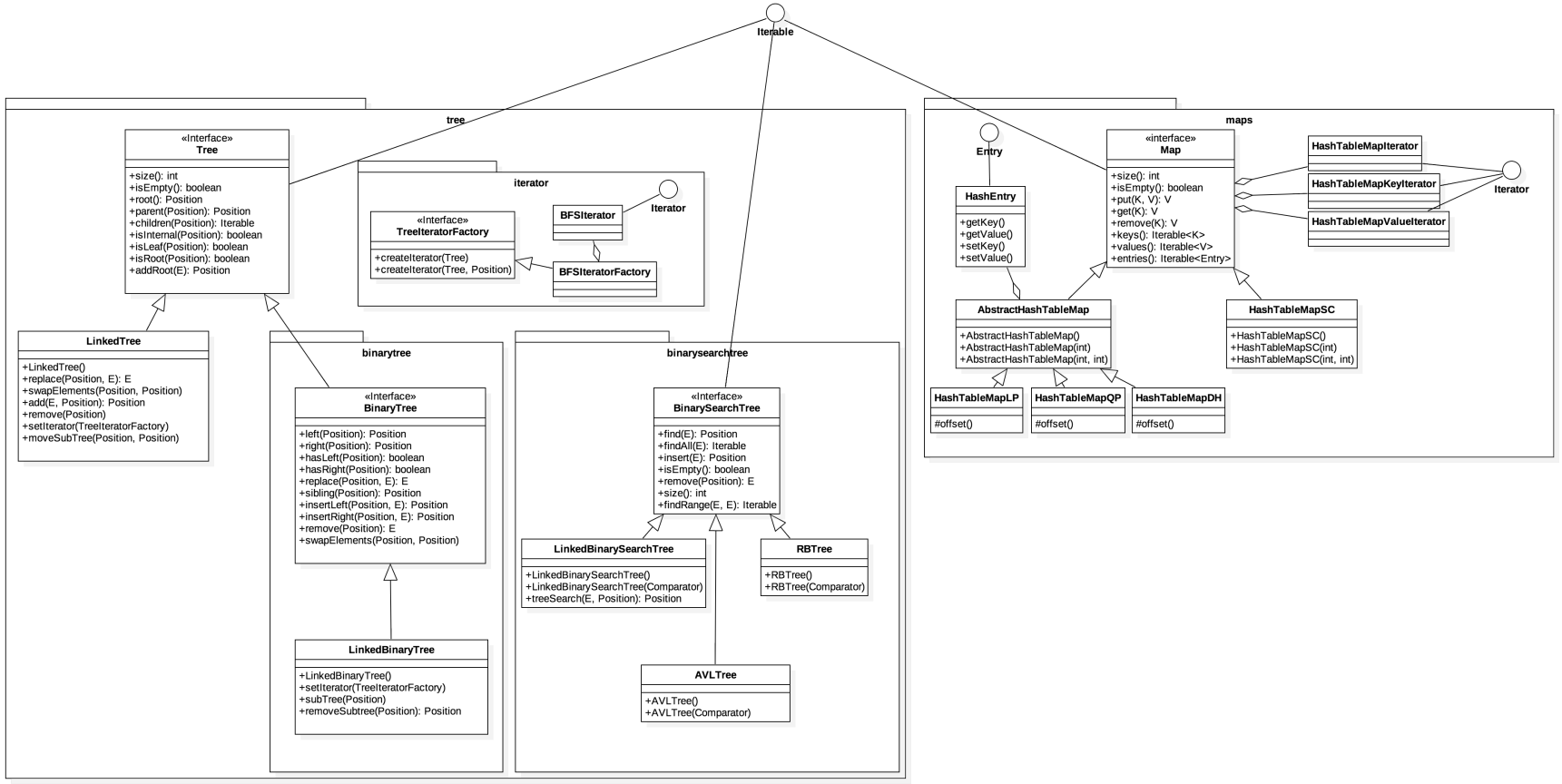
b) **[0.5 puntos]** Implementar el método `searchCompany` que reciba el nombre de una empresa y devuelva, en el caso de que esté entre la cartera de empresas, el organigrama correspondiente.

c) **[0.75 puntos]** Implementar el método `getGrantHolders` que reciba el nombre de una compañía y devuelva (en la estructura de datos que considere adecuada) la colección de becarios (personal de mínima responsabilidad) de dicha compañía.

d) **[0.75 puntos]** Implementar el método `getChiefs` que reciba el nombre de una compañía y un empleado, y devuelva (en la estructura de datos que considere adecuada) la colección de superiores de dicho empleado.

e) **[1.0 punto]** Implementar el método `getEmployees` que reciba un cargo y devuelva (en la estructura de datos que considere adecuada) la colección de empleados que ocupan dicho cargo en la Torre Europa.

NOTA: Para crear las clases `URJCInvest`, `OrganizationChart` y `Employee` se podrá agregar en su interior cualquiera de las estructuras de datos estudiadas durante el curso, valorándose especialmente lo adecuado de la elección que se realice. No es necesario proporcionar el código de aquellas estructuras de datos desarrolladas durante el curso.


Iterable
tree
«Interface»
Tree
+size(): int
+isEmpty(): boolean
+root(): Position
+parent(Position): Position
+children(Position): Iterable
+isInternal(Position): boolean
+isLeaf(Position): boolean
+isRoot(Position): boolean
+addRoot(E): Position
iterator
«Interface»
TreeIteratorFactory
+createIterator(Tree)
+createIterator(Tree, Position)
BFSIterator
Iterator
BFSIteratorFactory
LinkedTree
+LinkedTree()
+replace(Position, E): E
+swapElements(Position, Position)
+add(E, Position): Position
+remove(Position)
+setIterator(TreeIteratorFactory)
+moveSubTree(Position, Position)
binarytree
«Interface»
BinaryTree
+left(Position): Position
+right(Position): Position
+hasLeft(Position): boolean
+hasRight(Position): boolean
+replace(Position, E): E
+sibling(Position): Position
+insertLeft(Position, E): Position
+insertRight(Position, E): Position
+remove(Position): E
+swapElements(Position, Position)
LinkedBinaryTree
+LinkedBinaryTree()
+setIterator(TreeIteratorFactory)
+subTree(Position)
+removeSubtree(Position): Position
binarysearchtree
«Interface»
BinarySearchTree
+find(E): Position
+findAll(E): Iterable
+insert(E): Position
+isEmpty(): boolean
+remove(Position): E
+size(): int
+findRange(E, E): Iterable
LinkedBinarySearchTree
+LinkedBinarySearchTree()
+LinkedBinarySearchTree(Comparator)
+treeSearch(E, Position): Position
RBTree
+RBTree()
+RBTree(Comparator)
AVLTree
+AVLTree()
+AVLTree(Comparator)
maps
Entry
HashEntry
+getKey()
+getValue()
+setKey()
+setValue()
«interface»
Map
+size(): int
+isEmpty(): boolean
+put(K, V): V
+get(K): V
+remove(K): V
+keys(): Iterable<K>
+values(): Iterable<V>
+entries(): Iterable<Entry>
HashTableMapIterator
HashTableMapKeyIterator
HashTableMapValueIterator
Iterator
AbstractHashTableMap
+AbstractHashTableMap()
+AbstractHashTableMap(int)
+AbstractHashTableMap(int, int)
HashTableMapSC
+HashTableMapSC()
+HashTableMapSC(int)
+HashTableMapSC(int, int)
HashTableMapLP
#offset()
HashTableMapQP
#offset()
HashTableMapDH
#offset()

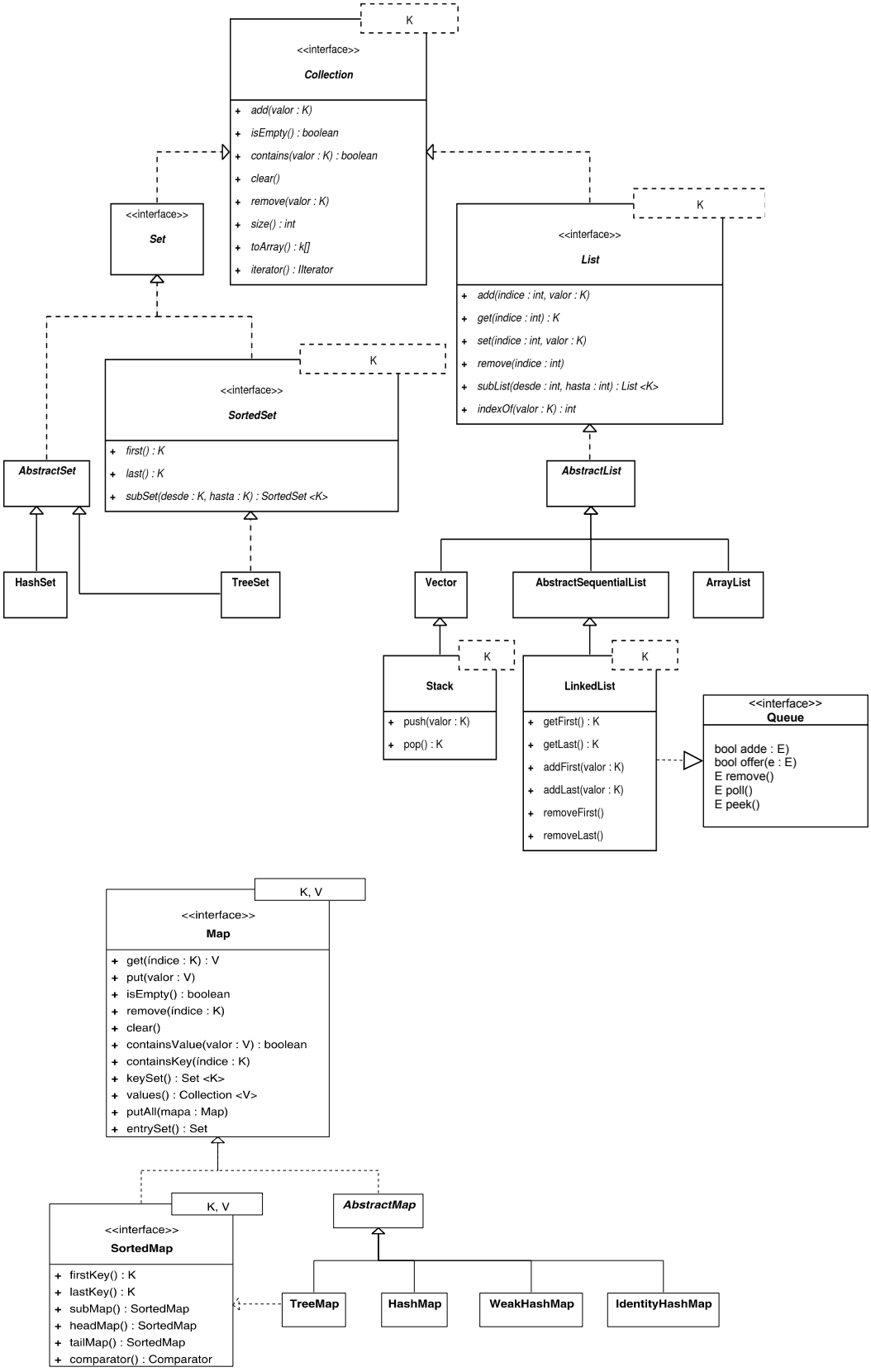


Diagrama de clases correspondiente a las estructuras propias de Java