Ejercicio 1 [3.0 puntos].

1.1: **[1.0 punto]** Represente gráficamente los árboles AVL y Rojo-Negor de altura 5 que almacen el menor número posible de elementos.

1.2: **[1.0 punto]** Se quiere utilizar un mapa para almacenar, como máximo 50 entradas. Determine y justifique qué tamaño debería tener dicha tabla si las colisiones se resuleven por encadenamiento separado y por direccionamiento abierto.

1.3: **[1.0 punto]** Determine y justifique el número máximo de claves que se puede almacenar en un árbol B de orden 5 y altura 3.

Ejercicio 2 [3.5 puntos]

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

a) **[1.0 punto]** Implementar método `toLinkedTree`, que crea un `LinkedTree` con la misma información y la misma relación padre hijo que el `LinkedBinarySearchTree` de partida.

b) **[1.5 puntos]** Implementar el método `removeRange`, que recibe dos argumentos comparables, `[min, max]`, y eliminan los nodos del árbol cuyo valor esté dentro de dicho rango.

c) **[1.0 punto]** Dada la siguiente clase `Student`, hacer los cambios e implementar los métodos que se consiere oportunos sobre ella para que se puedan insertar en `LinkedBinarySearchTree` objetos de dicha clase ordenados por el número de expediente.

```
public class Student{
    String name;
    String surname;
    int exp;

    public Student(){
    }
}
```

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]

Recientemente la URJC ha conseguido el contrato para gestionar el censo de los habitantes de Madrid de cara a las próximas elecciones. Cada votante tiene un número de DNI y, asociado a dicho número, aparece el colegio electoral y la mesa en la que debe votar.

Cada colegio da servicio a un conjunto de calles de cada municipio. A su vez, cada colegio tiene varias meses identificadas con letras del alfabeto. Por último, cada mesa da servicio a un máximo de 800 personas. Por ejemplo, si un colegio da servicio a la calles: Rosal, Margarita y Lirio, con 701, 849 y 377 personas cada una, harán falta 3 mesas (la A para 643 votantes, la B para 642 votantes y la C para 642 votantes).

Además, los votantes se reparten entre las mesas de manera ordenada por DNI. Es decir, la mesa A tiene los votantes con número de DNI más bajo, la mesa B tiene los siguientes en numeración, y así sucesivamente.

La información llega a los colegios en cadenas de texto como la siguiente:

645342334H, c/ Clavel, Manolo Gutiérrez Cano

532452344H, c/ Margarita, Roberto Pérez Alonso

Se desea almacenar toda esta información en varias estructuras de datos para que se puedan realizar dos tipos de consultas. Por un lado, que un votante, introduciendo su DNI pueda buscar la mesa que le corresponde. Por otro lado, se desea crear el listado de los votantes de una mesa que debe estar ordenado por número de DNI creciente.

Se pide:

a) **[1.0 punto]** Implementar dentro de la clase `ElectoralCollege` las estructuras de datos que se consideren convenientes para cumplir con los requisitos indicados.

b) **[1.0 punto]** Implementar los métodos de inicialización `addVoter` (para alimentar la clase con los votantes posibles) y el método `makeStationDistribution` (que reparte los votantes almacenados entre las diferentes mesas electorales).

c) **[1.5 puntos]** Implementar los métodos de consulta `getAllVoters` (para obtener el censo total del colegio), y el método `getStationVoters` (para obtener votantes de cada mesa electoral).

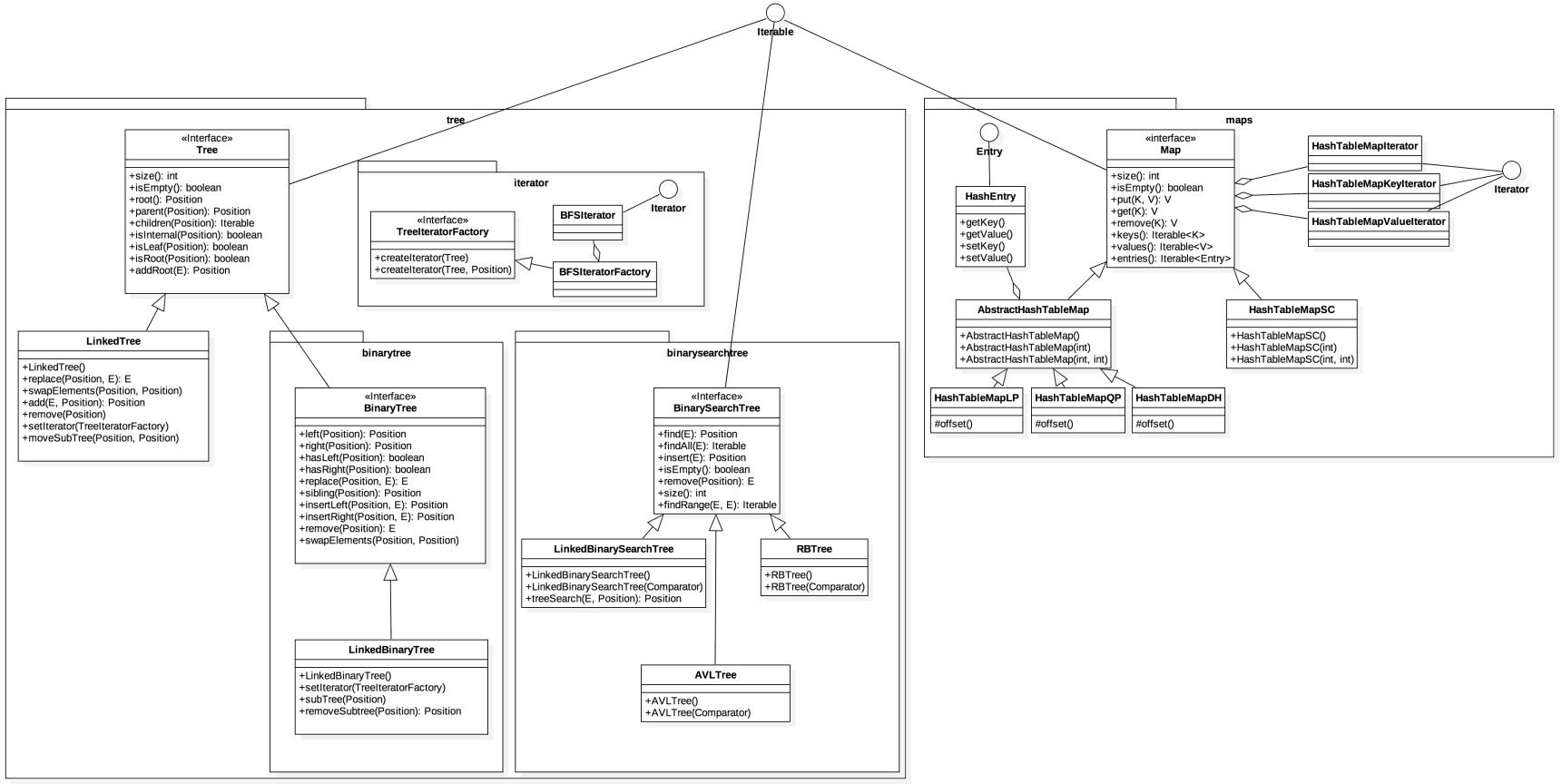
A continuación se proporciona un fragmento de código conen el que ilustra el uso de esta estructura de datos.

```
public void test ElectoralCollege () {
    ElectoralCollege instance = new ElectoralCollege();
    instance.addVoter("111111111A", "c/ Clavel", "Juan Perez Perez");
    instance.addVoter("222222222A", "c/ Petunia", "Andres Martin Palomo");
    instance.addVoter("333333333B", "c/ Lila", "Pedro Serrano Vega");
    instance.addVoter("444444444C", "c/ Leonidas", "Daniel Vela Amor");

    instance.makeStationDistribution();

     ..... resultA = instance.getAllVoters();
     ..... resultB = instance.getStationVoters('A');
....
}
```

NOTA: En la clases suministradas se podrán agregar los métodos que se consideren oportunos y cualquiera de las estructuras de datos estudiadas durante el curso, valorándose especialmente lo adecuado de la elección que se realice.


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()

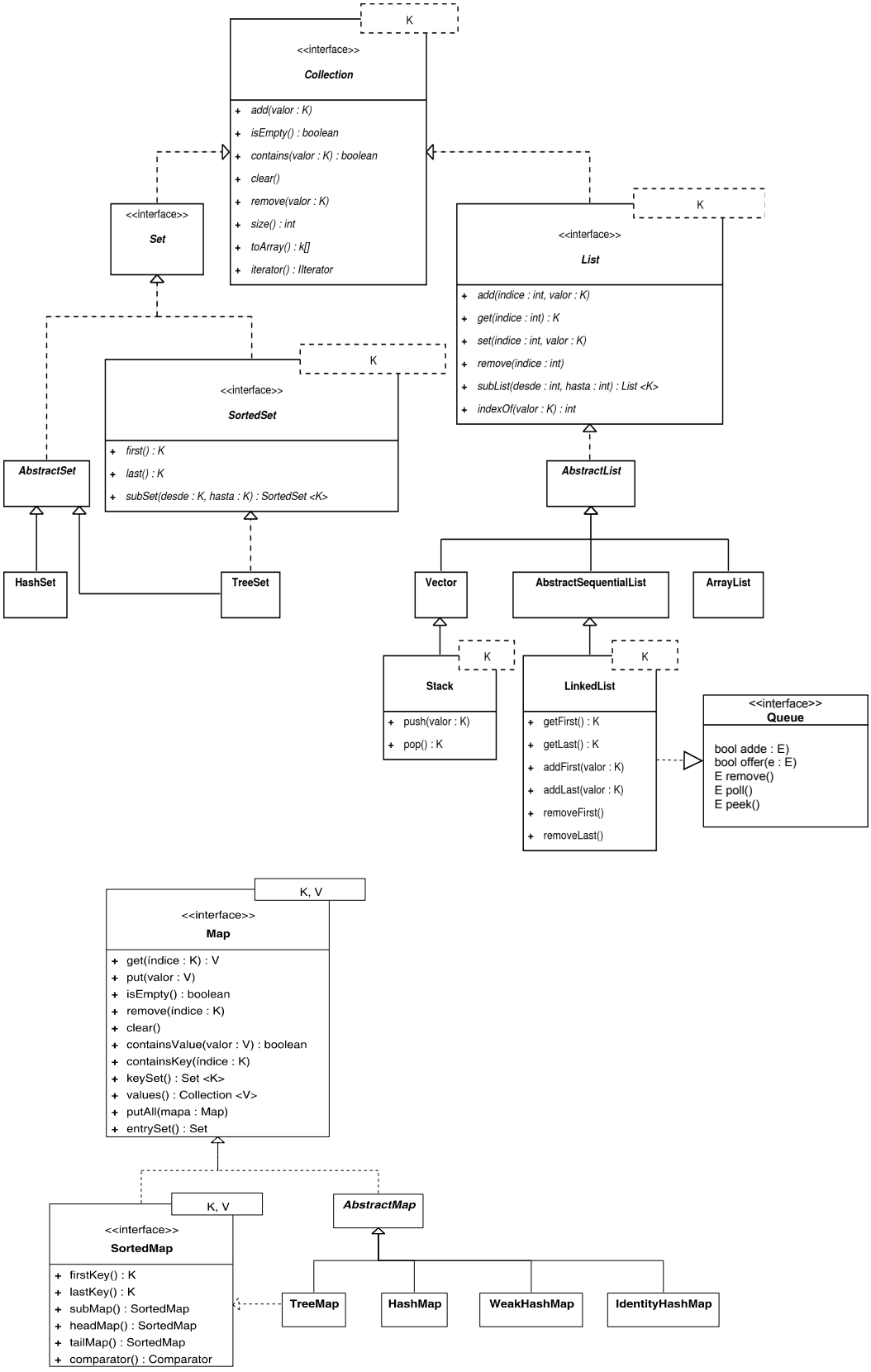


Diagrama de clases correspondiente a las estructuras propias de Java