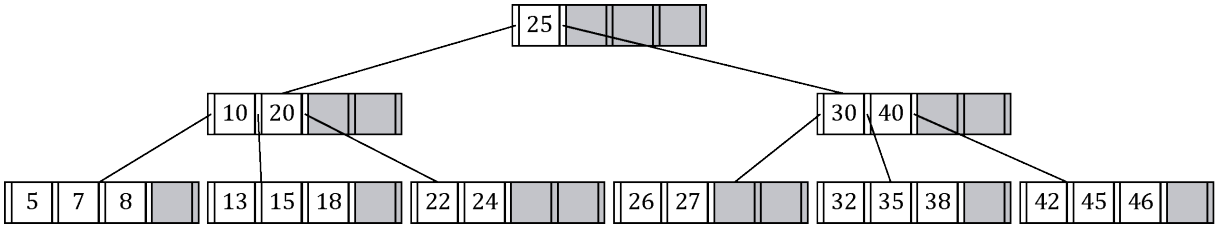
Ejercicio 1 [3.0 puntos].

1.1: **[1.5 puntos]** Dado un mapa de tamaño `20` donde las colisiones se resuelve con prueba lineal y la función de dispersión es `h(k) = k mod TAM_TABLA`, siendo `k` la clave, represente el estado de dicho mapa después insertar las siguientes claves: `4381, 1323, 6183, 4119, 4319, 9619, 1919`. Describa si existe algún problema que pueda reducir la eficiencia de esta tabla y, en caso afirmativo, cómo se podría minimizar.

1.2: **[1.5 puntos]** Dado el árbol B de la figura adjunta, respresentar el árbol resultante después de la siguiente secuencia de borrados: `25 – 45 – 24 – 38 – 32 – 8 – 27 – 46 – 13 – 42 – 5 – 22 – 18 – 26 – 7 – 35 – 15.`



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]

`HashTableMapSC<K,V>` es una estructura de datos de tipo tabla *hash* donde las colisiones se resuelven mediante el uso de listas enlazadas. En particular, cuando se inserta un elemento en un `HashTableMapSC` se utiliza una función de *hash* para localizar la posición de la tabla en la que hay que insertar el elemento. En caso de que dos elementos tengan la misma clave dispersada, se insertan en la posición dada por la función *hash* en una lista enlazada.

El problema de esta estructura es que, en caso de colisiones, la complejidad media en el caso de inserción y de búsqueda es $W(n)$. Por ello, se desea sustituir dichas listas por otra estructura de datos que, en caso de colisión, tenga una complejidad media de inserción y de búsqueda de $W(1)$.

Se pide crear una clase `HTMEfficient` con las siguientes características:

a) **[1 punto]** Crear el constructor y los atributos necesarios en `HTMEfficient` para que se permitan inserciones y borrados con una complejidad media de $W(1)$ en caso de colisión.

b) **[1 punto]** Crear los métodos `put` y `get` de `HTMEfficient` para que la estructura de datos verifique los requisitos de complejidad.

c) **[0.5 puntos]** Crear el método `remove` de `HTMEfficient` para que la estructura de datos verifique los requisitos de complejidad.

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.

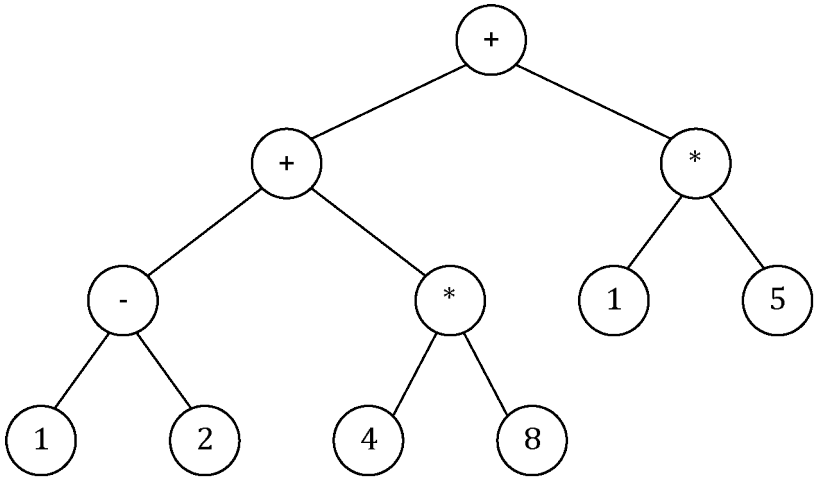
Ejercicio 3 [3.5 puntos]

Se desea desarrollar un sistema para la evaluación de expresiones aritméticas que se introduzcan en modo texto. Para tal fin, se pide implementar la clase `ArithmeticEvaluator`, que permite crear objetos que evalúan expresiones aritméticas. En concreto, la clase tendrá un método que recibe una cadena con la expresión y devuelve el resultado de evaluarla. La cabecera del método será:

```
public int evaluate(String exp)
```

Solo se contempla el uso de los operadores +,-, y * y la existencia de números naturales en el rango `[0,9]`. Por ejemplo, si un objeto de la clase `ArithmeticEvaluator` recibe "1+12*2", devolvería 25. Obsérvese que la operación de multiplicación tiene prioridad sobre la suma y la resta. La evaluación de la expresión aritmética se puede realizar mediante el uso de árboles binarios por lo que, para construir la clase `ArithmeticEvaluator`, se puede utilizar un objeto de la clase `LinkedBinaryTree<String>`.

En la figura adjunta se muestra cómo se podría representar en memoria un árbol dada la expresión aritmética `1-2+4*8+1*5`.



Este árbol se construye de la siguiente manera. En primer lugar, se identifica el operador de menor prioridad. A igualdad de prioridad, se elige el que aparece más a la izquierda en la expresión aritmética. Dicho operador se insertará en la raíz del árbol. A continuación, se repite el mismo procedimiento para la subcadena izquierda y derecha (insertando el resultado como hijos izquierdo y derecho, respectivamente). Como se puede ver en la figura, los operadores siempre son nodos internos, mientras que los operandos son las hojas.

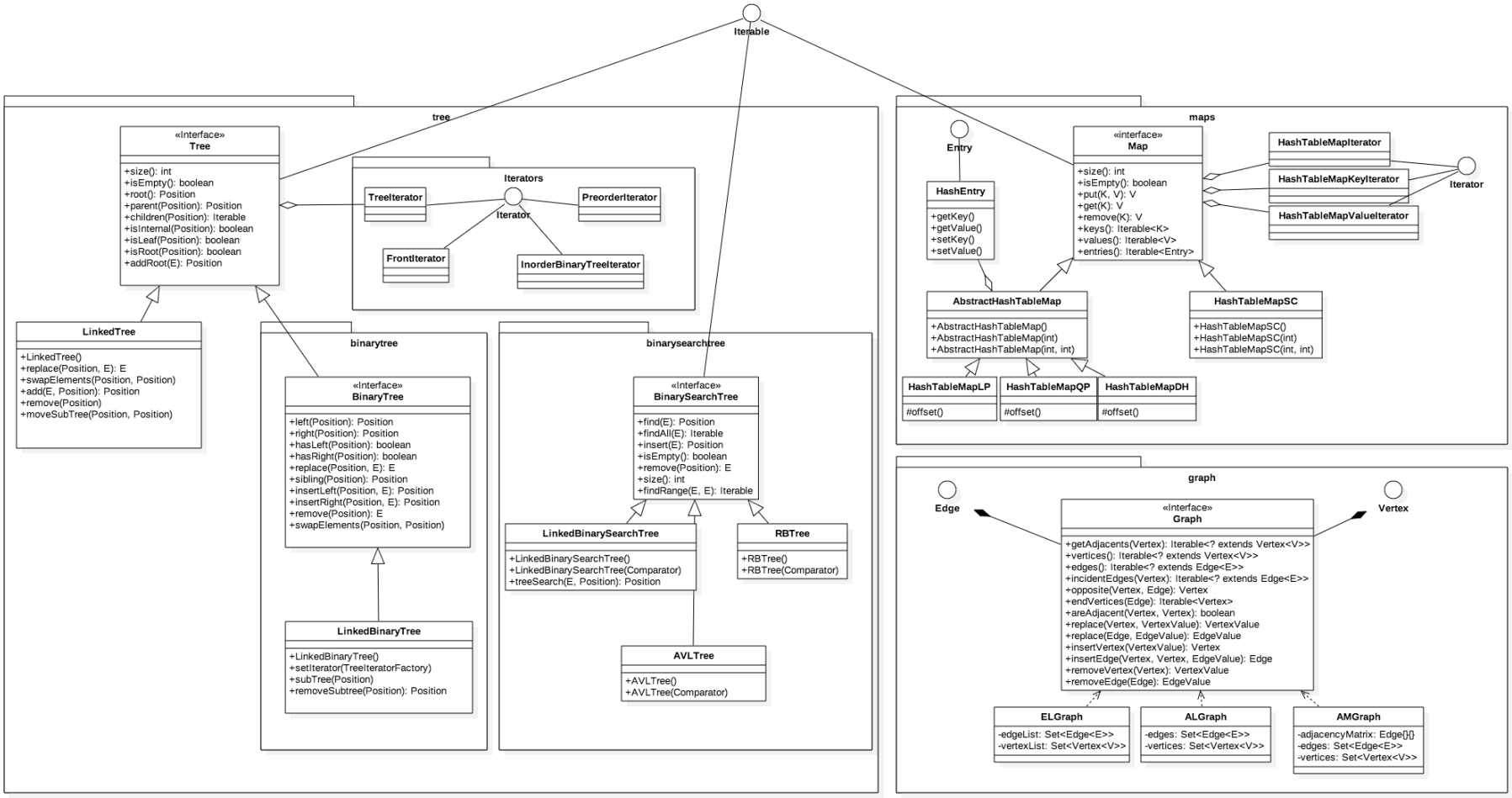
En particular se pide:

a) **[0.5 puntos]** En la clase `ArithmeticEvaluator`, definir el tipo de dato que considere más adecuado e implementar el método `getTree` para que devuelva el árbol derivado de la expresión.

b) **[1.5 puntos]** Implementar el método constructor de la clase `ArithmeticEvaluator`. Dicho método constructor recibe una expresión aritmética y rellena el árbol.

c) **[1.5 puntos]** Implementar el método `evaluate` en la clase `ArithmeticEvaluator`. Dicho método devuelve un valor numérico de tipo entero con el resultado de la evaluación.

NOTA: La clase `ArithmeticEvaluator` podrá contener en su interior cualquiera de las estructuras de datos de la clase `Collection` o de las estudiadas durante el curso, valorándose especialmente lo adecuado de la elección que se realice. La clase `String` cuenta con los métodos `substring(int start, int end)`, que obtiene la subcadena que empieza en `start` y termina en end-1, y `lastIndexOf(char c)` que devuelve la posición de la última aparición del carácter `c` en la cadena (`-1` si no está en la cadena). Se recuerda que, por lo general, es un mal diseño tener métodos que compartan mucho código o no introducir métodos privados auxiliares cuando sea necesario. Las soluciones recursivas no serán penalizadas.


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
Iterators
TreeIterator
Iterator
PreorderIterator
FrontIterator
InorderBinaryTreeIterator
LinkedTree
+LinkedTree()
+replace(Position, E): E
+swapElements(Position, Position)
+add(E, Position): Position
+remove(Position)
+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()
graph
Edge
«Interface»
Graph
+getAdjacents(Vertex): Iterable<? extends Vertex<V>>
+vertices(): Iterable<? extends Vertex<V>>
+edges(): Iterable<? extends Edge<E>>
+incidentEdges(Vertex): Iterable<? extends Edge<E>>
+opposite(Vertex, Edge): Vertex
+endVertices(Edge): Iterable<Vertex>
+areAdjacent(Vertex, Vertex): boolean
+replace(Vertex, VertexValue): VertexValue
+replace(Edge, EdgeValue): EdgeValue
+insertVertex(VertexValue): Vertex
+insertEdge(Vertex, Vertex, EdgeValue): Edge
+removeVertex(Vertex): VertexValue
+removeEdge(Edge): EdgeValue
Vertex
ELGraph
-edgeList: Set<Edge<E>>
-vertexList: Set<Vertex<V>>
ALGraph
-edges: Set<Edge<E>>
-vertices: Set<Vertex<V>>
AMGraph
-adjacencyMatrix: Edge[][]
-edges: Set<Edge<E>>
-vertices: Set<Vertex<V>>

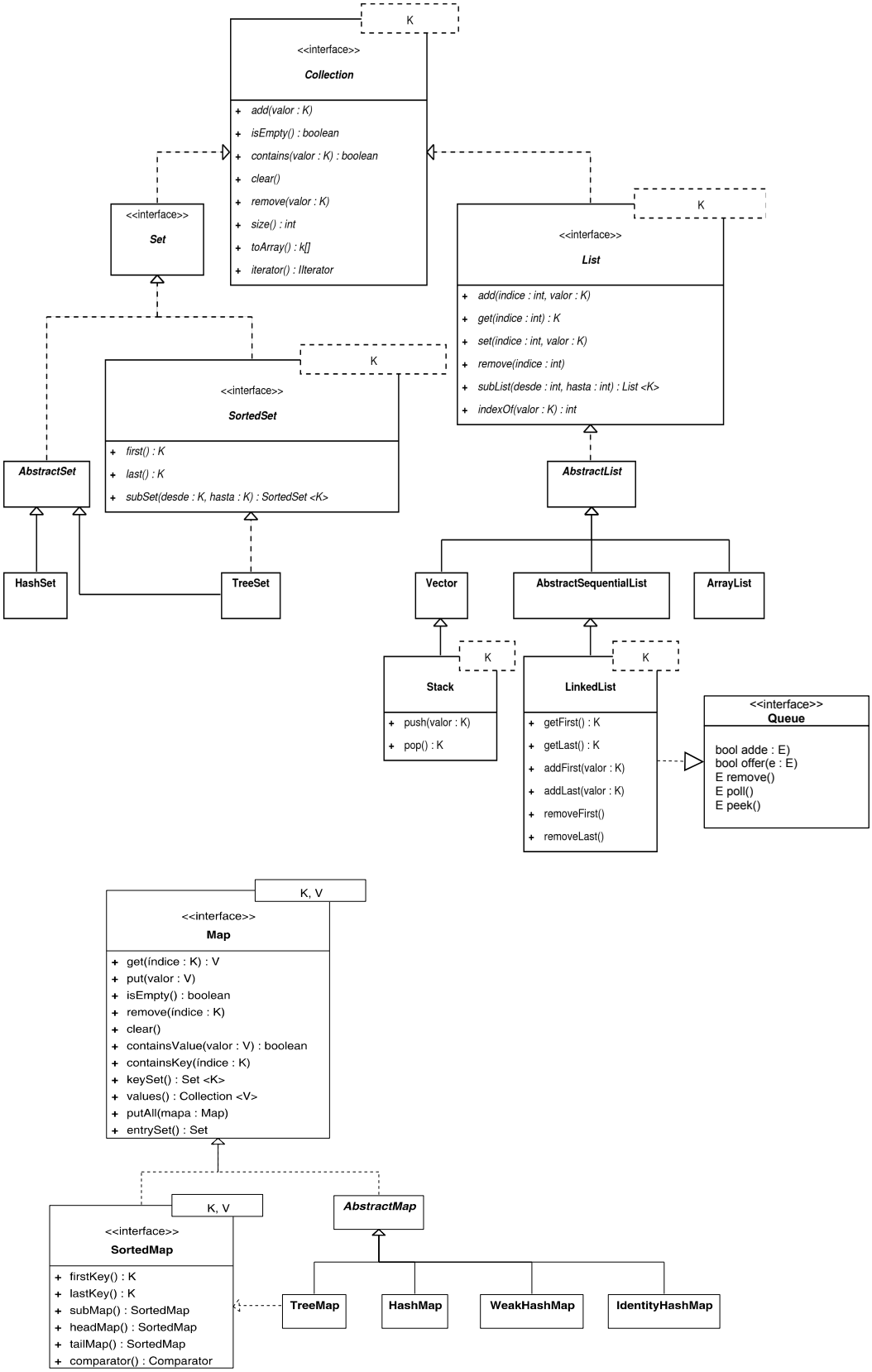


Diagrama de clases correspondiente a las estructuras propias de Java