Projeto Esinf

Gabriel Gonçalves - 1191296

Tiago Leite - 1191369

João Durães - 1211314

Francisco Bogalho - 1211304

António Bernardo - 1210805

Projeto Esinf 1

Introdução 3

Diagrama de classes 4

Algoritmos 5

Testes unitários 22

Melhoramentos possíveis 36

Conclusão 37

Introdução

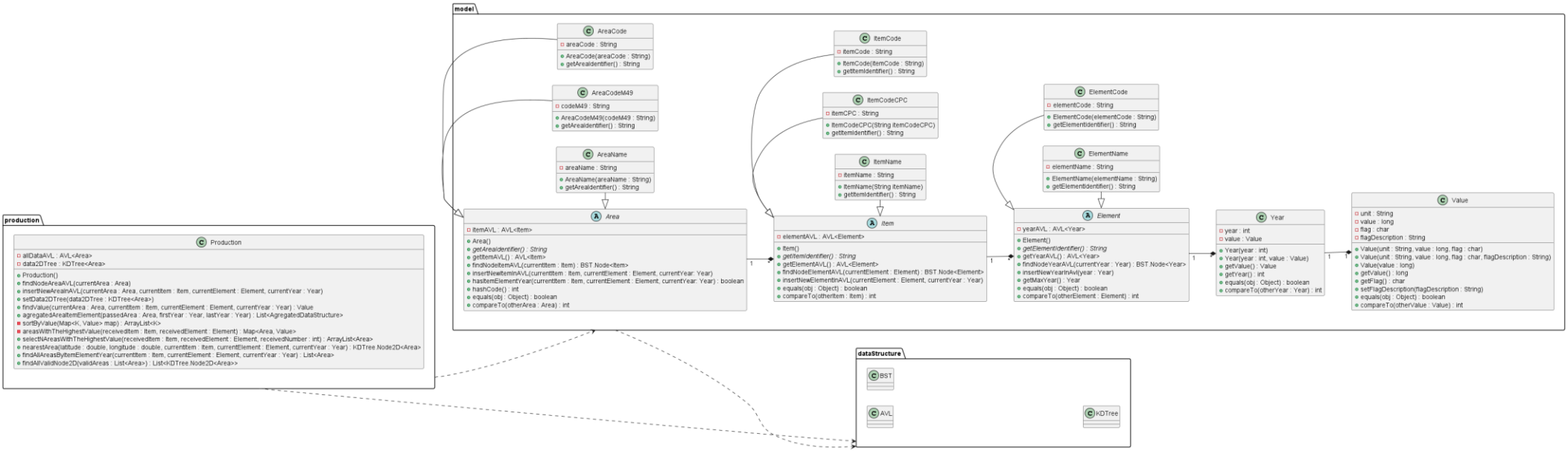
Este projeto tem como objetivo desenvolver um *software* com um conjunto de funcionalidades que permita gerir a informação relativa aos dados de produtos agrícolas e pecuários recolhidos pela FAO.

As funcionalidades requisitadas para a realização deste projeto são:

- Carregar informação a partir de um dado ficheiro com um certo nome e que contenha a extensão *CSV*, com recurso à classe binária de pesquisa (*BST);*
- *Dado uma determinada Area, devolver numa lista* a média dos *Value* agregados por *Item* e *Element* num intervalo de anos passado por parâmetro, ordenado por ordem decrescente de valor;
- Dado um determinado *Item* e *Element*, obter as top-N *Areas* com maior valor no último ano registado no conjunto de dados para aquele *Element*;
- Dado a latitude e a longitude devolver, com recurso à 2d-tree, todos os detalhes da *Area* geograficamente mais próxima das coordenadas;
- Dado um Item Code*, Element Code* e *Year Code* devolver, com recurso à 2d-tree, e o acumulado dos valores de produção para uma área geográfica retangular dada por uma latitude inicial, latitude final, longitude inicial e longitude final;

Estas funcionalidades devem ser implementadas da forma mais eficiente possível através do desenvolvimento das classes necessárias e com a utilização da *Java Collection Framework*.

Diagrama de classes



Para a estruturação dos classes optou-se pela criação de 3 classes abstratas designadas por *Area*, *Item* e *Element* no package models. As classes *AreaName*, *AreaCodeM49* e *AreaCode* são extendidas da classe "pai" *Area*, já as classes *ItemName*, *ItemCodeCPC* e *ItemCode* são extendidas da classe "pai" *Item*, por fim as classes *ElementName* e *ElementCode* são extendidas da classe "pai" *Element*.

A estruturação da informação na árvore binária de pesquisa (BST) encontra-se da seguinte maneira:

- A classe *Production* possui uma *AVL* de *Areas*;
- A classe *Area* contêm uma AVL de *Items*;
- A classe *Items* possui uma AVL de *Elements*;
- A classe *Elements* contêm uma AVL de *Years*;
- A classe Year possui um atributo Value por meio da composição.

Todos os métodos necessários para a resolução dos exercícios foram desenvolvidos na classe *Production*.

Algoritmos

Exercício 1

No exercício 1 é necessário carregar a informação relativa aos dados da FAOSTAT apenas em árvores binárias de pesquisa (BST) para ser realizadas pesquisas para obter os valores de produção através de diferentes combinações.

Para realizar a leitura do ficheiro no exercício 1 optou-se pela criação da classe ***ReadFile*** que contém um método que dado o caminho do ficheiro retorna caso este exista uma lista que contêm um *array* de *strings*. A divisão de cada linha do ficheiro em colunas realizou-se através de "","" caso a linha contenha esta sequência de caracteres, senão através de ",". A complexidade deste algoritmo é O(1) e é determinístico.

```java
public List<String[]> readFile(String filePath) throws IOException {

    if (filePath == null) {
        throw new FileNotFoundException();
    }

    FileReader file = new FileReader(filePath);
    BufferedReader bufferedReader = new BufferedReader(file);

    // Ler o cabeçalho
    String header = bufferedReader.readLine();

    List<String[]> arrayList = new ArrayList<>();
    String sentence;

    while ((sentence = bufferedReader.readLine()) != null) {
        String[] columns;

        if (sentence.contains(",\"")) {
            columns = sentence.split( regex: ",\"");
        } else {
            columns = sentence.split( regex: ",");
        }

        removeTrash(columns);
        arrayList.add(columns);
    }
    bufferedReader.close();

    return arrayList;
}
```

Para uma melhor organização do projeto criou-se 3 interfaces que possuem a posição de cada campo no ficheiro:

- ***IndexColumnsFile*** – Contém a posição de cada campo dos ficheiros de produção;
- ***IndexColumnsFileAreaCoordinates*** – Contém a posição de cada campo do ficheiro de coordenadas;
- ***IndexColumnsFileFlags*** – Contém a posição de cada campo do ficheiro de bandeiras.

Existe ainda uma interface denominada *IndexSearch*, que contêm os *index* de pesquisa. Esta interface é muito importante, pois determina como cada *AVL* de áreas, item e elementos vai ser organizada.

Na classe ***FileToBST*** foi criado o método ***addObjetcsToAVL*** é um método bastante importante da passagem do ficheiro para uma *BST*, pois este recebe objetos do tipo *Area*, *Item*, *Element* e *Year*.

Inicialmente este verifica se já existe na *AVL* a área passada caso não exista é chamado o método *insertNewAreaInAVl* que pertence à classe *Production*. Se já existir a área na *AVL* é verificado se o item já existe e assim sucessivamente com todos os objetos passados. A complexidade deste algoritmo é O(1) e é determinístico.

```
private void addObjectsToAVL(Production production, Area currentArea, Item currentItem, Element currentElement, Year currentYear) {
    BST.Node<Area> nodeAreaAVL = production.findNodeAreaAVL(currentArea);

    if (nodeAreaAVL == null) {
        production.insertNewAreaInAVL(currentArea, currentItem, currentElement, currentYear);
    } else {
        BST.Node<Item> nodeItemAVL = nodeAreaAVL.getElement().findNodeItemAVL(currentItem);

        if (nodeItemAVL == null) {
            nodeAreaAVL.getElement().insertNewItemInAVL(currentItem, currentElement, currentYear);
        } else {

            BST.Node<Element> nodeElementAVL = nodeItemAVL.getElement().findNodeElementAVL(currentElement);
            if (nodeElementAVL == null) {
                nodeItemAVL.getElement().insertNewElementInAVL(currentElement, currentYear);
            } else {

                BST.Node<Year> nodeYear = nodeElementAVL.getElement().findNodeYearAVL(currentYear);

                if (nodeYear == null) {
                    nodeElementAVL.getElement().insertNewYearInAvl(currentYear);
                }
            }
        }
    }
}
```

O método ***convertFileInBST*** converte os ficheiros de produção em BST juntando também os ficheiros de coordenadas e o ficheiro de bandeiras.

Para guardar inicialmente a informação do ficheiro de bandeiras numa estrutura adequada optou-se pelo *Map* que contém como *key* o campo *flag* que é um *char* e o *value* o campo *flag description*.

A informação das coordenadas foi guardada numa classe *KDTree*.

Para guardar a informação do ficheiro de produção foi criado um ciclo for para percorrer a lista retornada pelo método de leitura do ficheiro, seguidamente em cada iteração é criado um objeto *Area*, *Item*, *Element*, Value e *Year* com auxílios de alguns métodos.

Por fim, para adicionar os objetos à *AVL* é chamado o método *addObjetcsToAVL* explicado anteriormente.

A complexidade deste algoritmo é O(1) e é determinístico.

```
public Production convertFileInBST(int indexArea, int indexItem, int indexElement, String filePath, String fileAreaCoordinates

    Map<Character, String> mapFlags = convertFileFlags(fileFlagsPath);

    ReadFile rf = new ReadFile();
    List<String[]> coordinatesContentFile = rf.readFile(fileAreaCoordinatesPath);
    KDTree<Area> areaKDTree = convertFileInKDTree(coordinatesContentFile);
    production.setData2DTree(areaKDTree);

    List<String[]> contentFile = rf.readFile(filePath);
    for (String[] columns : contentFile) {

        if (columns.length == TOTAL_COLUMNS_FILE_PRODUCTION) {
            // Area
            Area area = createArea(indexArea, columns[INDEX_AREA_CODE], columns[INDEX_AREA_CODE_M49], columns[INDEX_AREA]);

            // Item
            Item item = createItem(indexItem, columns[INDEX_ITEM_CODE], columns[INDEX_ITEM_CPC], columns[INDEX_ITEM]);

            // Element
            Element element = createElement(indexElement, columns[INDEX_ELEMENT_CODE], columns[INDEX_ELEMENT]);

            // Value
            Value value = joinColumnsValueToFlagDescription(mapFlags, columns[INDEX_VALUE_UNIT], columns[INDEX_VALUE], columns

            // Year
            Year year = createYear(columns[INDEX_YEAR], value);

            addObjectsToAVL(production, area, item, element, year);
        }
    }
```

O método ***FindValue*** permite obter os valores de produção (Value, Unit, Flag, Flag Description) através de várias combinações, como por exemplo.: (Area, Item, Element, Year) ou (AreaCode, ItemCode, ElementCode, Year).

Através do método criado é possível realizar esta pesquisa por várias combinações, pois um objeto do tipo *AreaName* ou *AreaCode* ou *AreaCodeM49* é uma *Area* devido à hierarquia das classes realizada. O mesmo se aplica para o Item e o Elemento.

No método inicialmente é verificado se alguns dos objetos passados por parâmetro são iguais a nulo, caso aconteça é retornado *null*, pois não é possível obter os valores de produção.

Seguidamente é verificado consecutivamente a existência da área, item, elemento e ano passado até chegar ao valor de produção pretendido. Se em alguma situação não existir alguns destes objetos é retornado nulo.

A complexidade deste algoritmo é O(1) e é determinístico.

```
public Value findValue(Area currentArea, Item currentItem, Element currentElement, Year currentYear) {
    if (currentArea == null || currentItem == null || currentElement == null || currentYear == null) {
        return null;
    } else {
        BST.Node< Area > nodeAreaAVL = this.allDataAVL.find(currentArea);

        if (nodeAreaAVL == null) {
            return null;
        } else {
            BST.Node< Item > nodeItemAVL = nodeAreaAVL.getElement().getItemAVL().find(currentItem);

            if (nodeItemAVL == null) {
                return null;
            } else {
                BST.Node< Element > nodeElementAVL = nodeItemAVL.getElement().getElementAVL().find(currentElement);

                if (nodeElementAVL == null) {
                    return null;
                } else {
                    BST.Node< Year > nodeYear = nodeElementAVL.getElement().getYearAVL().find(currentYear);

                    if (nodeYear == null) {
                        return null;
                    } else {
                        return nodeYear.getElement().getValue();
                    }
                }
            }
        }
    }
}
```

Exercício 2

Para a realização do exercício 2, em que nos é pedido devolver, numa estrutura de dados, a média dos valores agregados por *Item* e *Element* num intervalo de anos e numa *Area*, passada por parâmetro.

Optamos por criar o método **agregatedAreaItemElement** na classe *Production* e uma classe **AgregatedDataStructure**.

No método **agregatedAreaItemElement,** onde colocamos em uma lista o intervalo de anos, um item, um element e a média dos valores de produção, tem um pois o pior caso tem uma complexidade diferente do melhor caso a complexidade de $O(N^3)$ e é não determinístico.

Este método começa por verificar se as variáveis passadas por parâmetro são nulas, caso isto se confirme o método devolve nulo, confirmando assim que este método é não determinístico. De seguida verifica se o primeiro ano do intervalo é maior do que o último, caso isto se confirme troca o primeiro ano com o último. Na linha seguinte procura na *AVL* geral a *area* passada por parâmetro e devolve o *node*, caso o encontre. Depois se o *node* não for nulo, o método percorrerá o *item* e por cada *item* percorrerá o *element* e por cada *element* percorrerá o ano. Se o ano em analise estiver dentro do intervalo passado por parâmetro, soma o valor de produção relativamente ao ano, relativamente ao *element*, relativamente ao *item* aumenta o incrementador. Depois, se o incrementador não for 0 calcula a média do valor de produção, cria uma instância com os valores em analise e adiciona essa instância á lista *result.* Por fim devolve a lista *result*.

A classe publica **AgregatedDataStructure** tem cinco variáveis e implementa a interface comparable para poder utilizar o método compareTo.

```
public class AgregatedDataStructure implements Comparable<AgregatedDataStructure>{

    3 usages
    private Year firstYear, lastYear;
    3 usages
    private Item item;
    3 usages
    private Element element;
    5 usages
    private double averageValue;
```

Possibilita a instanciação de um objeto, com todas as cinco variáveis, da classe AgregatedDataStructure.

```
public AgregatedDataStructure(Year firstYear, Year lastYear, Item item, Element element, double averageValue) {
    this.firstYear = firstYear;
    this.lastYear = lastYear;
    this.item = item;
    this.element = element;
    this.averageValue = averageValue;

}
```

O método compreTo serve para poder comparar dois objetos.

```
@Override
public int compareTo(AgregatedDataStructure o) { return (int)(o.averageValue-averageValue); }
```

O método equals compara duas instâncias da classe AgregatedDataStructure devolvendo true se as instâncias forem iguais e falso no caso contrário.

```
public boolean equals(AgregatedDataStructure obj) {
    if (this == obj) {
        return true;
    }

    if (obj == null || this.getClass() != obj.getClass()) {
        return false;
    }

    AgregatedDataStructure otherAgregatedDataStructure = (AgregatedDataStructure) obj;


    return firstYear.getYear() == otherAgregatedDataStructure.firstYear.getYear()
            && lastYear.getYear() == otherAgregatedDataStructure.lastYear.getYear()
            && item.equals(otherAgregatedDataStructure.item)
            && element.equals(otherAgregatedDataStructure.element)
            && averageValue == otherAgregatedDataStructure.averageValue;
}
```

Exercício 3

Para a realização do **exercício 3, em que nos é pedido para um determinado Item e Element, obter as top-N Areas com maior valor no último ano registado no conjunto de dados para aquele Element.**

Optamos por criar o método ***areasWithTheHighestValue*** na class *Production* onde colocamos em uma treeMap as Areas com o maior valor no último registo para um determinado Item e Elemento. A complexidade deste método é O(n).

```
private Map<Area, Value> areasWithTheHighestValue(Item receivedItem, Element receivedElement) {
    Iterable<Area> areasAVLIterator = this.allDataAVL.inOrder();
    Map<Area, Value> treeMapAreaValue = new TreeMap<>();

    for (Area area : areasAVLIterator) {
        BST.Node<Item> nodeItemAVL = area.getItemAVL().find(receivedItem);

        if (nodeItemAVL != null) {
            Item item = nodeItemAVL.getElement();
            BST.Node<Element> nodeElementAVL = item.findNodeElementAVL(receivedElement);

            if (nodeElementAVL != null) {
                Year maxYear = nodeElementAVL.getElement().getMaxYear();

                treeMapAreaValue.put(area, maxYear.getValue());
            }
        }
    }
    return treeMapAreaValue;
}
```

O último registo é entendido como o último ano em que foi registado o maior valor no ficheiro de dados em análise, por isso foi criado o método ***getMaxYear*** na class *Element*. A complexidade deste método é O(n).

```
public Year getMaxYear() {
    Iterable<Year> yearAVLIterator = yearAVL.inOrder();
    Year maxYear = null;

    for (Year year : yearAVLIterator) {
        if (maxYear == null) {
            maxYear = year;
        } else {
            if (year.compareTo(maxYear) > 0) {
                maxYear = year;
            }
        }
    }
    return maxYear;

}
```

Ainda na class *Production* criamos o método *sortByValue* e *selectNAreasWithTheHighestValue. No método sortByValue* recebe por parâmetro um mapa e retorna um *ArrayList* ordenado por ordem decrescente em função do *Value.* A complexidade deste método é O(N²).

```
private <K> ArrayList<K> sortByValue(Map<K, Value> map) {
    ArrayList<K> sortedList = new ArrayList<>();
    ArrayList<Value> list = new ArrayList<>();

    for (Map.Entry<K, Value> entry : map.entrySet()) {
        list.add(entry.getValue());
    }

    list.sort(new ValueComparator());

    for (Value value : list) {
        for (Map.Entry<K, Value> entry : map.entrySet())
            if (entry.getValue().equals(value)) {
                sortedList.add(entry.getKey());
            }
        }
    }

    return sortedList;
}
```

Para finalizar o exercício, no método *selectNAreasWithTheHighestValue* recebemos como parâmetro o item, o elemento e as n áreas pretendidas. Chama o método já referido anteriormente, *areasWithTheHighestValue*. Posteriormente organiza o *ArrayList* que o método anterior teria nos retornado. Para finalizar criamos outro *ArrayList* onde só será adicionado as top N áreas. A complexidade deste método é O(N).

```
3 usages
public ArrayList<Area> selectNAreasWithTheHighestValue(Item receivedItem, Element receivedElement, int receivedNumber) {
Map<Area, Value> treeMapAreaValue = areasWithTheHighestValue(receivedItem, receivedElement);
ArrayList<Area> finalArray = sortByValue(treeMapAreaValue);
ArrayList<Area> nAreasWithTheHighestValue = new ArrayList<>();

if (!finalArray.isEmpty()) {
    for (int i = 0; i < receivedNumber; i++) {
        nAreasWithTheHighestValue.add(finalArray.get(i));

    }
}
return nAreasWithTheHighestValue;
}
```

Exercício 4

No exercício 4 é pedido para devolver todos os detalhes da área geograficamente mais próxima das coordenadas: latitude longitude passadas por parâmetro. Estas áreas devem ser apenas aquelas que contenham um determinado item, um determinado elemento num dado ano, sendo estes passados por parâmetro.

A solução deste exercício utiliza uma estrutura de dados denominada de 2D-Tree. Os métodos utilizados da 2D-Tree para a resolução deste exercício são os seguintes: buildTree, findNearestNeighbour e o inOrder.

O método buildTree é responsável por realizar a construção da 2D-Tree. Ele recebe uma lista com todos os nós que a instância do objeto que chama este método vai conter. Este método garante que a árvore é construída e mantida em equilíbrio ao ordenar os elementos da lista cada vez que um novo elemento vai ser inserido. Esta ordenação pode ser realizada pelo valor da coordenada x ou pelo valor da coordenada y, dependendo do nível da árvore que estamos a tratar, pois primeiro ordenamos por x e depois y, trocando novamente para x e assim sucessivamente.

```
/**
 * Método para construir a KDTree completa com a passagem de uma lista que contem todos os nós que devem ser
 * armazenados
 *
 * @param info Lista que contem objetos do tipo Node2D
 */
public void buildTree(List<Node2D<E>> info) {
    root = buildTree(info, divX: true);
}
```

```java
/**
 * Método para construir a KDTree com uma lista de objetos do tipo Node2D de forma a manter a estrutura da árvore
 * equilibrada
 *
 * @param info Lista que contem objetos do tipo Node2D
 * @param divX true para comparar os nós pelas coordenadas do x, ou false para comparar os nós pelas coordenadas do
 *             y
 * @return
 */
private Node2D<E> buildTree(List<Node2D<E>> info, boolean divX) {

    if (info == null || info.isEmpty())
        return null;

    QuickSort2DNode.quickSort(info, divX ? cmpX : cmpY);

    int mid = info.size() / 2;

    Node2D<E> node2D = new Node2D<E>();
    node2D.element = info.get(mid).getElement();
    node2D.coords = info.get(mid).getCoords();
    node2D.left = buildTree(info.subList(0, mid), !divX);

    if (mid + 1 <= info.size() - 1)
        node2D.right = buildTree(info.subList(mid + 1, info.size()), !divX);

    return node2D;

}
```

Este método é não determinístico devido ao tempo de complexidade do QuickSort já ser não determinístico por si só. O melhor caso do algoritmo QuickSort é de O(n*logn) e o pior caso é de O(n^2). Como este método necessita de colocar todos os elementos na árvore, tem uma complexidade no melhor caso de O(n*logn) e no pior caso O(n^2).

O método findNearestNeighbour é utilizado para encontra o nó mais próximo da umas dadas coordenadas. Ele recebe como parâmetro dois valores, o valor da coordenada x e o valor da coordenada y e depois realiza a procura a partir do nó raiz.

```java
/**
 * Método para encontrar o nó mais próximo de umas dadas coordenadas desta KDTree
 *
 * @param x O valor da coordenada x (longitude)
 * @param y O valor da coordenada y (latitude)
 * @return Devolve o Node2D que se encontram mais próximo das coordenadas fornecidas
 */
public Node2D<E> findNearestNeighbour(double x, double y) {
    return findNearestNeighbour(root, x, y, root, divX: true);
}


/**
 * Método para realizar a procura do nó desta KDTree mais próximo das coordenadas fornecidas
 *
 * @param node2D      O nó atual em que nos encontramos a fazer a procura por um nó mais próximo das coordenadas
 *                    desejadas
 * @param x           O valor da coordenada x (longitude)
 * @param y           O valor da coordenada y (latitude)
 * @param closestNode O Node2D atual mais próximo das coordenadas desejadas
 * @param divX        Para sabermos se neste nível da árvore devemos comparar as coordenadas do x ou as coordenadas
 *                    do y
 * @return Devolve o Node2D mais próximo das coordenadas desejadas
 */
private Node2D<E> findNearestNeighbour(Node2D<E> node2D, double x, double y, Node2D<E> closestNode, boolean divX) {

    if (node2D == null)
        return null;

    double d = Point2D.distanceSq(node2D.coords.x, node2D.coords.y, x, y);
    double closestDist = Point2D.distanceSq(closestNode.coords.x, closestNode.coords.y, x, y);

    if (closestDist > d)
        closestNode.setObject(node2D);

    double delta = divX ? x - node2D.coords.x : y - node2D.coords.y;
    double delta2 = delta * delta;

    Node2D<E> node1 = delta < 0 ? node2D.left : node2D.right;
    Node2D<E> node2 = delta < 0 ? node2D.right : node2D.left;


    findNearestNeighbour(node1, x, y, closestNode, !divX);

    if (delta2 < closestDist)
        findNearestNeighbour(node2, x, y, closestNode, !divX);

    return closestNode;
}
```

Este método é não determinístico, tendo como melhor tempo de complexidade O(logn) e como pior tempo de complexidade O(n).

O método inOrder é responsável por armazenar o elemento e as coordenadas de cada nó da árvore num HashMap, permitindo assim uma procura otimizada quando da utilização do mapa devolvido por este método. Como a área é única foi utilizada como chave para o mapa, garantindo assim uma procura de valores no mapa de complexidade O(1).

```java
/**
 * Método que devolve um HashMap dos elementos da KDTree, construído pela travessia in-order
 *
 * @return HashMap dos elementos da KDTree pela travessia in-order
 */
public Map<E, Node2D<E>> inOrder() {
    Map<E, Node2D<E>> snapshot = new HashMap<>();
    if (root != null)
        inOrderSubtree(root, snapshot);   // Preenche o mapa de forma recursiva
    return snapshot;
}

/**
 * Adiciona os elementos das subtree presentes no nó atual para um dado HashMap usando a travessia in-order
 *
 * @param node2D   Nó que serve como root da subtree
 * @param snapshot O HashMap onde os valores são colocados
 */
private void inOrderSubtree(Node2D<E> node2D, Map<E, Node2D<E>> snapshot) {
    if (node2D == null)
        return;
    inOrderSubtree(node2D.getLeft(), snapshot);
    Node2D<E> newNode2D = new Node2D<>(node2D.getElement(), leftChild: null, rightChild: null, node2D.getX(),
            node2D.getY());
    snapshot.put(node2D.element, newNode2D);
    inOrderSubtree(node2D.getRight(), snapshot);
}
```

O método inOrderSubtree é um algoritmo determinístico com complexidade de O(n), pois ao realizar este método, é sempre necessário passar pelos n nós existentes na árvore.

Como métodos para a resolução do exercício 4 na classe Production, temos os seguintes: nearestArea, findAllAreasByItemElementYear e findAllValidNode2D.

O método nearestArea é responsável por descobrir qual é a área mais próxima de umas determinadas coordenadas onde existe um determinado item, com um determinado elemento num dado ano.

```java
/**
 * Método para encontrar a área (País) mais próximo de uma dada latitude e longitude, que contenha um determinado
 * Item, e que esse Item contenha um determinado Element de um determinado Year.
 *
 * @param latitude       A latitude desejada
 * @param longitude      A longitude desejada
 * @param currentItem    O 'item' que queremos que o país tenha
 * @param currentElement O elemento que queremos que o Item tenha
 * @param currentYear    O ano em que queremos que o 'item' e o elemento existam
 * @return Devolve o KDTree.Node2D mais próximo das coordenadas desejadas que satisfaçam as condições do 'item', do
 * elemento e do ano
 */
public KDTree.Node2D<Area> nearestArea(double latitude, double longitude, Item currentItem, Element currentElement,
                                       Year currentYear) {
    List<Area> validAreas = findAllAreasByItemElementYear(currentItem, currentElement, currentYear);

    if (validAreas.isEmpty())
        return null;

    List<KDTree.Node2D<Area>> validAreaCoordinates = findAllValidNode2D(validAreas);

    if (validAreaCoordinates.isEmpty())
        return null;

    KDTree<Area> kdTree = new KDTree<>();
    kdTree.buildTree(validAreaCoordinates);

    return kdTree.findNearestNeighbour(longitude, latitude);

}
```

O método findAllAreasByItemElementYear é responsável por procurar todas as áreas dentro de uma AVL que contenham um dado item com um dado elemento e num dado ano. Este método devolve uma lista com todas as áreas que satisfazem a condição de valores que deve conter.

```java
/**
 * Encontra todos os países que contenham um determinado item, com um determinado elemento num determinado ano, e
 * devolve uma lista com todos esses países.
 *
 * @param currentItem    O 'item' que queremos que o país tenha
 * @param currentElement O elemento que queremos que o Item tenha
 * @param currentYear    O ano em que queremos que o 'item' e o elemento existam
 * @return Uma lista com todos os países que satisfaçam a condição
 */
public List<Area> findAllAreasByItemElementYear(Item currentItem, Element currentElement, Year currentYear) {
    List<Area> validAreas = new ArrayList<>();

    if (currentItem == null || currentElement == null)
        return validAreas;

    Iterable<Area> areasAVLIterator = this.allDataAVL.inOrder();

    for (Area area : areasAVLIterator) {
        if (area.hasItemElementYear(currentItem, currentElement, currentYear)) {
            validAreas.add(area);
        }
    }

    return validAreas;
}
```

Este algoritmo é não determinístico devido ao método com o nome hasItemElementYear utilizado de forma auxiliar ser não determinístico.

O método findAllValidNode2D é responsável por colocar numa lista as informações relativas aos nós que contêm dados sobre determinadas áreas. Essas áreas são passadas por parâmetro através de uma lista que contém objetos do tipo Area. Este método devolve uma lista com nós em que contem a informação da área e afins e ainda as coordenadas em que este nó se deve encontrar.

```java
/**
 * Método para encontrar todos os Node2D da KDTree que correspondam aos países que são passados por parâmetro, e
 * devolver todos esses KDTree.Node2D numa lista
 *
 * @param validAreas Lista com as áreas que queremos saber as coordenadas e respetivo KDTree.Node2D
 * @return Devolve uma lista que contem todos os KDTree.Node2D que correspondam aos países presentes na lista
 * passada por parâmetro
 */
public List<KDTree.Node2D<Area>> findAllValidNode2D(List<Area> validAreas) {
    List<KDTree.Node2D<Area>> validAreasCoordinates = new ArrayList<>();

    if (validAreas == null)
        return validAreasCoordinates;

    Map<Area, KDTree.Node2D<Area>> allAreasCoordinates = data2DTree.inOrder();

    for (Area area : validAreas) {
        if (allAreasCoordinates.get(area) != null) {
            validAreasCoordinates.add(allAreasCoordinates.get(area));
        }
    }

    return validAreasCoordinates;
}
```

Neste método, temos um algoritmo determinístico devido a que o método inOrder é determinístico com complexidade de O(n), sendo que posteriormente percorremos cada elemento da lista passada por parâmetro e o acesso realizado ao HashMap é O(1). Temos então uma complexidade temporal de O(n) para este método.

Exercício 5

Neste exercício, é pedido para recorrer à 2D-Tree, para devolver para um Item Code, Element Code e Year Code o acumulado dos valores de produção de uma área geográfica retangular, sendo que esta é dada por uma latitude inicial, uma latitude final, uma longitude inicial e uma longitude final.

Posto isto, decidimos criar o método **rangeSearch** na classe **KDTree** para podermos pesquisar num determinado intervalo de coordenadas, sendo que neste caso, vamos, através das latitudes (final e inicial) e longitudes (final e inicial), encontrar as áreas que estão dentro desses mesmos limites.

```
private List<E> rangeSearch(List<E> result, Node2D<E> node2D, boolean divX, double initialX, double initialY,
    if (node2D == null)
        return result;

    boolean insideX = node2D.getX() > initialX && node2D.getX() < finalX;
    boolean insideY = node2D.getY() > initialY && node2D.getY() < finalY;

    if (insideX && insideY)
        result.add(node2D.getElement());

    if (divX) {
        if (insideX) {
            result = rangeSearch(result, node2D.getRight(), divX: false, initialX, finalX, initialY, finalY);
            return rangeSearch(result, node2D.getLeft(), divX: false, initialX, finalX, initialY, finalY);
        } else if (node2D.getX() < initialX) {
            return rangeSearch(result, node2D.getRight(), divX: false, initialX, finalX, initialY, finalY);
        } else if (node2D.getX() > finalX) {
            return rangeSearch(result, node2D.getLeft(), divX: false, initialX, finalX, initialY, finalY);
        }
    } else {
        if (insideY) {
            result = rangeSearch(result, node2D.getLeft(), divX: true, initialX, finalX, initialY, finalY);
            return rangeSearch(result, node2D.getRight(), divX: true, initialX, finalX, initialY, finalY);
        } else if (node2D.getY() < initialY) {
            return rangeSearch(result, node2D.getRight(), divX: true, initialX, finalX, initialY, finalY);
        } else if (node2D.getY() > finalY) {
            return rangeSearch(result, node2D.getLeft(), divX: true, initialX, finalX, initialY, finalY);


        }
    }
```

De seguida, criamos o método **valueProductionGeographicArea** na class **Production**, sendo que inicialmente é chamado o método anterior referido para podermos guardar na lista de **Areas** todas as áreas que se encontram dentro desses limites, ou seja, dentro dos limites de latitude inicial e final e também longitude inicial e final (x e y, respetivamente).

Depois é realizado um ciclo para percorrer todas as **Areas** dessa lista e em cada área é chamado o método **finvalue** para retornar os valores de produção para aquela **area** e o **item**, **element** e o **year**, sendo que são passados por parâmetro. Se o **value** for diferente de **null** então é somado o valor de **production** à variável sum.

Por fim, retorna a soma do acumulado dos valores de **production** para uma área geográfica dada por uma latitude inicial, latitude final, longitude inicial e longitude final.

```
public long valueProductionGeographicalArea(Item item, Element element, Year year, Double latI, Double longI, Double latF, Double long
    List<Area> listArea = allDataAVL.inOrderList();

    List<Area> list = data2DTree.rangeSearch(listArea, latI, longI, latF, longF);

    long sum = 0;
    for (Area area : list) {
        Value value = findValue(area, item, element, year);
        if (value != null) {
            sum += value.getValue();
        }
    }

    return sum;
}
```

Testes unitários

Os testes unitários servem para verificar o grau de qualidade do código para um determinado método a partir da criação de classes de testes.

Classe de testes *AreaCodeM49Test*, *AreaCodeTest* e *AreaNameTest*

Nestas classes testamos os métodos ***getAreaIdentifier***, ***equals*** e ***compareTo*.**

Teste do método ***getAreaIdentifier***

No método **getAreaIdentifier** verificamos se o método retorna o valor esperado, testando valores iguais, diferentes e *null*.

```java
/**
 * Teste ao método getAreaIdentifier para verificar se retorna o valor esperado.
 */
@Test
public void getAreaIdentifierTest1() {
    AreaCodeM49 areaCodeM49 = new AreaCodeM49("3214");

    String result = areaCodeM49.getAreaIdentifier();
    String expectedResult = "3214";

    Assert.assertEquals(expectedResult, result);
}
```

```java
/**
 * Teste ao método getAreaIdentifier para verificar se retorna null.
 */
@Test
public void getAreaIdentifierTest3() {
    AreaCodeM49 areaCodeM49 = new AreaCodeM49(null);

    String result = areaCodeM49.getAreaIdentifier();

    Assert.assertNull(result);
}
```

```java
/**
 * Teste ao método getAreaIdentifier para verificar se retorna um valor diferente do esperado.
 */
@Test
public void getAreaIdentifierTest2() {
    AreaCodeM49 areaCodeM49 = new AreaCodeM49("934");

    String result = areaCodeM49.getAreaIdentifier();
    String expectedResult = "31";

    Assert.assertNotEquals(expectedResult, result);
}
```

Teste do método ***equals***

No método *equals* verificamos quando temos 2 objetos *Area* iguais ou diferentes.

```java
/**
 * Testa o método equals com dois objetos AreaCodeM49 iguais.
 */
@Test
public void equalsTest1() {
    AreaCodeM49 area1CodeM49 = new AreaCodeM49("123");
    AreaCodeM49 area2CodeM49 = new AreaCodeM49("123");

    boolean result = area1CodeM49.equals(area2CodeM49);

    Assert.assertTrue(result);
}
```

```java
/**
 * Testa o método equals com dois objetos AreaCodeM49 diferentes.
 */
@Test
public void equalsTest2() {
    AreaCodeM49 area1CodeM49 = new AreaCodeM49("123");
    AreaCodeM49 area2CodeM49 = new AreaCodeM49("12345");

    boolean result = area1CodeM49.equals(area2CodeM49);

    Assert.assertFalse(result);
}
```

Teste do método ***compareTo***

Ainda nas classes *Area* testamos o método ***compareTo*** verificamos quando:

- Dois objetos *Area* com o atributo igual.
- Um objeto *Area* com um atributo maior que o outro.
- Um objeto *Area* com um atributo menor que o outro

```java
@Test
public void compareToTest1() {
    //Códigos M49 iguais
    AreaCodeM49 area1CodeM49 = new AreaCodeM49("123");
    AreaCodeM49 area2CodeM49 = new AreaCodeM49("123");

    int result = area1CodeM49.compareTo(area2CodeM49);
    int expResult = 0;

    Assert.assertEquals(expResult, result);
}

@Test
public void compareToTest2() {
    //Código M49 1 maior que Código M49 2
    AreaCodeM49 area1CodeM49 = new AreaCodeM49("124");
    AreaCodeM49 area2CodeM49 = new AreaCodeM49("123");

    int result = area1CodeM49.compareTo(area2CodeM49);
    int expResult = 1;

    Assert.assertEquals(expResult, result);
}
```

```java
*/
@Test
public void compareToTest3() {
    //Código M49 2 maior que Código M49 1
    AreaCodeM49 area1CodeM49 = new AreaCodeM49("123");
    AreaCodeM49 area2CodeM49 = new AreaCodeM49("124");

    int result = area1CodeM49.compareTo(area2CodeM49);
    int expResult = -1;

    Assert.assertEquals(expResult, result);
}
```

Classe de testes ***ElementCodeTest*** e ***ElementNameTest***

Nestas classes testamos os métodos ***getElementIdentifier***, ***equals*** e ***compareTo*.**

Teste do método ***getElementIdentifier***

No método **getElementIdentifier** verificamos se o método retorna o valor esperado, testando valores iguais, diferentes e *null*.

```java
@Test
public void getElementIdentifierTest1() {
    ElementName elementName = new ElementName("5152");

    String result = elementName.getElementIdentifier();
    String expectedResult = "5152";

    Assert.assertEquals(expectedResult, result);
}
```

```java
@Test
public void getElementIdentifierTest2() {
    ElementName elementName = new ElementName("3121312");

    String result = elementName.getElementIdentifier();
    String expectedResult = "31";

    Assert.assertNotEquals(expectedResult, result);
}
```

```java
@Test
public void getElementIdentifierTest3() {
    ElementName elementName = new ElementName(null);
    String result = elementName.getElementIdentifier();

    Assert.assertNull(result);
}
```

Teste do método ***equals***

No método *equals* verificamos quando temos 2 objetos *Element* são iguais ou diferentes.

```java
@Test
public void equalsTest1() {
    ElementName elementName1 = new ElementName("5152");
    ElementName elementName2 = new ElementName("5152");

    boolean result = elementName1.equals(elementName2);

    Assert.assertTrue(result);
}
```

```java
@Test
public void equalsTest2() {
    ElementName elementName1 = new ElementName("5152");
    ElementName elementName2 = new ElementName("98765");

    boolean result = elementName1.equals(elementName2);

    Assert.assertFalse(result);
}
```

Teste do método ***compareTo***

Ainda nas classes *Element* testamos o método ***compareTo*** verificamos quando:

- Dois objetos *Element* com o atributo igual.
- Um objeto *Element* com um atributo maior que o outro.
- Um objeto *Element* com um atributo menor que o outro.

```java
@Test
public void compareToTest1() {
    ElementName elementName1 = new ElementName("5152");
    ElementName elementName2 = new ElementName("5152");

    int result = elementName1.compareTo(elementName2);
    int expResult = 0;

    Assert.assertEquals(expResult, result);
}

@Test
public void compareToTest2() {
    ElementName elementName1 = new ElementName("5153");
    ElementName elementName2 = new ElementName("5152");

    int result = elementName1.compareTo(elementName2);
    int expResult = 1;

    Assert.assertEquals(expResult, result);
}
```

```java
@Test
public void compareToTest3() {
    ElementName elementName1 = new ElementName("5152");
    ElementName elementName2 = new ElementName("5153");

    int result = elementName1.compareTo(elementName2);
    int expResult = -1;

    Assert.assertEquals(expResult, result);
}
```

Classe de testes ***ItemCodeTest, ItemCPCTest*** e ***itemNameTest***

Nestas classes testamos os métodos ***getItemIdentifier***, ***equals*** e ***compareTo***.

Teste do método ***getItemIdentifier***

No método **getItemIdentifier** verificamos se o método retorna o valor esperado, testando valores iguais, diferentes e *null*.

```
@Test
public void getItemIdentifierTest1() {
    ItemCode itemCode = new ItemCode("51233");

    String result = itemCode.getItemIdentifier();
    String expectedResult = "51233";

    Assert.assertEquals(expectedResult, result);
}
```

```
@Test
public void getItemIdentifierTest2() {
    ItemCode itemCode = new ItemCode("331");

    String result = itemCode.getItemIdentifier();
    String expectedResult = "1";

    Assert.assertNotEquals(expectedResult, result);
}
```

```
@Test
public void getItemIdentifierTest3() {
    ItemCode itemCode = new ItemCode(null);

    String result = itemCode.getItemIdentifier();

    Assert.assertNull(result);
}
```

Teste do método ***equals***

No método *equals* verificamos quando temos 2 objetos *Element* são iguais ou diferentes.

```
@Test
public void equalsTest1() {
    ItemCode itemCode1 = new ItemCode("331");
    ItemCode itemCode2= new ItemCode("331");

    boolean result = itemCode1.equals(itemCode2);

    Assert.assertTrue(result);
}
```

```
@Test
public void equalsTest2() {
    ItemCode itemCode1 = new ItemCode("331");
    ItemCode itemCode2= new ItemCode("9876");

    boolean result = itemCode1.equals(itemCode2);

    Assert.assertFalse(result);
}
```

Teste do método ***compareTo***

Ainda nas classes Item testamos o método ***compareTo*** verificamos quando:

- Dois objetos Item com o atributo igual.
- Um objeto Item com um atributo maior que o outro.
- Um objeto Item com um atributo menor que o outro.

```java
@Test
public void compareToTest1() {
    ItemCode itemCode1 = new ItemCode("331");
    ItemCode itemCode2= new ItemCode("331");

    int result = itemCode1.compareTo(itemCode2);
    int expResult = 0;

    Assert.assertEquals(expResult, result);
}
```

```java
@Test
public void compareToTest2() {
    ItemCode itemCode1 = new ItemCode("332");
    ItemCode itemCode2= new ItemCode("331");

    int result = itemCode1.compareTo(itemCode2);
    int expResult = 1;

    Assert.assertEquals(expResult, result);
}
```

```java
@Test
public void compareToTest3() {
    ItemCode itemCode1= new ItemCode("331");
    ItemCode itemCode2 = new ItemCode("332");

    int result = itemCode1.compareTo(itemCode2);
    int expResult = -1;

    Assert.assertEquals(expResult, result);
}
```

Classe de testes **ProductionTest**

Teste do método ***findValue***

Para o método ***findValue*** criamos 7 testes:

- Verificar se a função devolve para um determinado código do país, código do item, código do elemento e ano o valor que corresponde à realidade;
- Verificar se a função devolve para um determinado nome do país, nome do item, nome do elemento e ano o valor que corresponde à realidade;
- Verificar se a função devolve para um determinado nome do país, nome do item, nome do elemento e ano o valor nulo devido a não existirem dados para o ano indicado;
- Verificar se a função devolve para um determinado nome do país, nome do item, nome do elemento e ano o valor nulo devido a não existirem dados para o elemento indicado;
- Verificar se a função devolve para um determinado nome do país, nome do item, nome do elemento e ano o valor nulo devido a não existirem dados para o item indicado;
- Verificar se a função devolve para um determinado nome do país, nome do item, nome do elemento e ano o valor nulo devido a não existirem dados para o país indicado;
- Verificar se a função devolve para um determinado código M49 do país, nome do item, código do elemento e ano o valor nulo devido a não existirem dados.

```
@Test
public void findValueTest2() {
    Area areaName = new AreaName("Croatia");
    Item itemName = new ItemName("Rye");
    Element elementName = new ElementName("Production");
    Year year = new Year(1993);

    Value result = production.findValue(areaName, itemName, elementName, year);
    Value expectedResult = new Value( unit: "tonnes",  value: 6273L,  flag: 'A',  flagDescription: "Official figure");

    Assert.assertEquals(expectedResult, result);
}

/**
 * Testa o método findValue para verificar se a função devolve para o país com o nome "Croatia", 'item' com o nome "Rye",
 * elemento com o nome "Production" e ano "2023" o null, pois não existe dados para o ano indicado.
 */
@Test
public void findValueTest3() {
    Area areaName = new AreaName("Croatia");
    Item itemName = new ItemName("Rye");
    Element elementName = new ElementName("Production");
    Year year = new Year(2023);

    Value result = production.findValue(areaName, itemName, elementName, year);

    Assert.assertNull(result);
}
```

Teste do método ***selectNAreasWithTheHighestValue***

Para o método ***selectNAreasWithTheHighestValue*** criamos 3 testes:

- Verificar se a função devolve para o um determinado item e elemento o valor que corresponde à realidade;
- Verifica se a função devolve para o um determinado item que não exite e elemento válido, um *ArrayLis*t vazio;
- Verifica se a função devolve para o um determinado item válido e elemento que não exite, um *ArrayLis*t vazio;

```
@Test
 public void selectNAreasWithTheHighestValueTest1() {
    Item itemName = new ItemName("Beer of barley, malted");
    Element elementName = new ElementName("Production");

    ArrayList<Area> result = production.selectNAreasWithTheHighestValue(itemName, elementName, receivedNumber: 2);
    ArrayList<Area> expectedResult = new ArrayList<>();
    expectedResult.add(new AreaName("United Kingdom of Great Britain and Northern Ireland"));
    expectedResult.add(new AreaName("Spain"));

    Assert.assertEquals(expectedResult, result);
}

 @Test
 public void selectNAreasWithTheHighestValueTest2() {
    Item itemName = new ItemName("Peras");
    Element elementName = new ElementName("Production");

    ArrayList<Area> result = production.selectNAreasWithTheHighestValue(itemName, elementName, receivedNumber: 2);

    Assert.assertTrue(result.isEmpty());
}
 @Test
 public void selectNAreasWithTheHighestValueTest3() {
    Item itemName = new ItemName("Beer of barley, malted");
    Element elementName = new ElementName("maças");

    ArrayList<Area> result = production.selectNAreasWithTheHighestValue(itemName, elementName, receivedNumber: 2);

    Assert.assertTrue(result.isEmpty());
}
```

Testes do método **agregatedAreaItemElement**

Para o método **agregatedAreaItemElement** criamos 4 testes, em que:

- Verificamos se a função permanece vazia quando o intervalo é impossivel, para os dados em questão;
- Verificamos se a função permanece vazia quando a area não existe para os dados em questão;
- Verificamos se a função devolve os valores corretos para um intervalo em que os extremos não têm valor
- Verifica-los se a função devolve os valores corretos para um intervalo de anos e area sem erros, para os dados em questão.

```
@Test
public void agregatedAreaItemElementTest1() {
    Area areaName = new AreaName("Croatia");
    Year firstYear = new Year(2050);
    Year lastYear = new Year(2100);

    List<AgregatedDataStructure> result = production.agregatedAreaItemElement(areaName, firstYear, lastYear);

    Assert.assertTrue(result.isEmpty());
}
```

```java
@Test
public void agregatedAreaItemElementTest4() throws Exception {
    readExercise2TestFile();
    Area areaName = new AreaName("Croatia");
    Year firstYear = new Year(2002);
    Year lastYear = new Year(2012);

    List<AgregatedDataStructure> result = production.agregatedAreaItemElement(areaName, firstYear, lastYear);

    Item itemName1 = new ItemName("Swine / pigs");
    Element elementName1 = new ElementName("Stocks");
    Item itemName2 = new ItemName("Sugar Crops Primary");
    Element elementName2 = new ElementName("Yield");
    Item itemName3 = new ItemName("Carrots and turnips");
    Element elementName3 = new ElementName("Yield");
    Item itemName4 = new ItemName("Bees");
    Element elementName4 = new ElementName("Stocks");
    Item itemName5 = new ItemName("Treenuts, Total");
    Element elementName5 = new ElementName("Yield");

    List<AgregatedDataStructure> test = new ArrayList<>();
    test.add(new AgregatedDataStructure(firstYear, lastYear, itemName1, elementName1,  averageValue: 1204960.0));
    test.add(new AgregatedDataStructure(firstYear, lastYear, itemName2, elementName2,  averageValue: 475585.0));
    test.add(new AgregatedDataStructure(firstYear, lastYear, itemName3, elementName3,  averageValue: 252391.0));
    test.add(new AgregatedDataStructure(firstYear, lastYear, itemName4, elementName4,  averageValue: 111000.0));
    test.add(new AgregatedDataStructure(firstYear, lastYear, itemName5, elementName5,  averageValue: 8582.0));

    boolean expResult = true;
    for (int i = 0; i < result.size(); i++) {
        if (!result.get(i).equals(test.get(i))) {
            expResult = false;
        }
    }

    Assert.assertTrue(expResult);
    beforeClass();
}
```

Para o método *findAllAreasByItemElementYear* que pertence ao exercício 4 foram criados três testes diferentes:

- Um em que é em que é verificado se as áreas para um dado 'item', elemento e ano são todas encontradas corretamente;
- Outro em que é verificado se o valor devolvido ao utilizar objetos com informação que não existe está correto (se a lista vem vazia);
- E outro onde é verificado se ao passar valores nulos, o método devolve uma lista vazia.

```java
@Test
public void findAllAreasByItemElementYearTest1() {
    Item item = new ItemName("Carrots and turnips");
    Element element = new ElementName("Production");
    Year year = new Year(2004);

    List<Area> resultList = production.findAllAreasByItemElementYear(item, element, year);
    List<Area> createdList = new ArrayList<>();
    createdList.add(new AreaName("Lithuania"));
    createdList.add(new AreaName("Malta"));
    createdList.add(new AreaName("Poland"));

    Assert.assertTrue(AuxiliariesTestMethods.assertEqualsLists(resultList, createdList));

    resultList.remove( index: 2);

    Assert.assertFalse(AuxiliariesTestMethods.assertEqualsLists(resultList, createdList));

}

@Test
public void findAllAreasByItemElementYearTest2() {
    Item item = new ItemName("Não existe");
    Element element = new ElementName("Production");
    Year year = new Year(2004);

    List<Area> resultList = production.findAllAreasByItemElementYear(item, element, year);
    List<Area> createdList = new ArrayList<>();

    Assert.assertTrue(AuxiliariesTestMethods.assertEqualsLists(resultList, createdList));

}
```

```
@Test
public void findAllAreasByItemElementYearTest3() {
    Year year = new Year(2004);

    List<Area> resultList = production.findAllAreasByItemElementYear( currentItem: null,  currentElement: null, year);
    List<Area> createdList = new ArrayList<>();

    Assert.assertTrue(AuxiliariesTestMethods.assertEqualsLists(resultList, createdList));

}
```

Para o método findAllValidNode2D foram criados três testes diferentes:

- Um onde é verificado se os valores que são devolvidos pelo método com a passagem de determinados valores para o Item, para o *Element* e para o *Year* estão corretos;
- Outro onde é verificado se ao utilizar um valor que não existe num objeto, o método devolve o valor que é suposto (lista vazia);
- E por fim outro em que verifica o valor obtido quando é passado por parâmetro o valor *null*.

```
@Test
public void findAllValidNode2DTest1() {
    Item item = new ItemName("Carrots and turnips");
    Element element = new ElementName("Production");
    Year year = new Year(2004);
    List<Area> validAreas = production.findAllAreasByItemElementYear(item, element, year);

    List<KDTree.Node2D<Area>> validNodes2DResult = production.findAllValidNode2D(validAreas);

    List<KDTree.Node2D<Area>> validNodes2DExpected = new ArrayList<>();
    KDTree.Node2D<Area> newNode2DOne = new KDTree.Node2D<>(new AreaName("Lithuania"),  leftChild: null,  rightChild: null,
            x: 23.881275,  y: 55.169438);
    KDTree.Node2D<Area> newNode2DTwo = new KDTree.Node2D<>(new AreaName("Malta"),  leftChild: null,  rightChild: null,
            x: 14.375416,  y: 35.937496);
    KDTree.Node2D<Area> newNode2DThree = new KDTree.Node2D<>(new AreaName("Poland"),  leftChild: null,  rightChild: null,
            x: 19.145136,  y: 51.919438);
    validNodes2DExpected.add(newNode2DOne);
    validNodes2DExpected.add(newNode2DTwo);
    validNodes2DExpected.add(newNode2DThree);

    Assert.assertTrue(AuxiliariesTestMethods.assertEqualsListNodes2D(validNodes2DResult, validNodes2DExpected));

    validNodes2DResult.remove( index: 2);
    Assert.assertFalse(AuxiliariesTestMethods.assertEqualsListNodes2D(validNodes2DResult, validNodes2DExpected));

}
```

```
@Test
public void findAllValidNode2DTest2() {
    Item item = new ItemName("Carrots and turnips");
    Element element = new ElementName("Não existe");
    Year year = new Year(2004);
    List<Area> validAreas = production.findAllAreasByItemElementYear(item, element, year);
    List<KDTree.Node2D<Area>> validNodes2DResult = production.findAllValidNode2D(validAreas);

    List<KDTree.Node2D<Area>> validNodes2DExpected = new ArrayList<>();

    Assert.assertTrue(AuxiliariesTestMethods.assertEqualsListNodes2D(validNodes2DExpected, validNodes2DResult));

}

@Test
public void findAllValidNode2DTest3() {
    List<KDTree.Node2D<Area>> validNodes2DResult = production.findAllValidNode2D( validAreas: null);
    List<KDTree.Node2D<Area>> validNodes2DExpected = new ArrayList<>();

    Assert.assertTrue(AuxiliariesTestMethods.assertEqualsListNodes2D(validNodes2DExpected, validNodes2DResult));
}
```

Para o exercício 4 foram ainda desenvolvidos três teste para o método *nearestArea*:

- Uma para verificar se ao passar um valor que não existe num dos objetos, o método devolve o valor de *null*;
- Outro para verificar se ao passar uma latitude e longitude muito alta, o método devolve na mesma o valor correto;
- E por fim outro para verificar se o valor devolvido para umas dadas condições está correto.

```
@Test
public void nearestAreaTest1() {
    Item item = new ItemName("Carrots and turnips");
    Element element = new ElementName("This element doesn't exists");
    Year year = new Year(1981);
    KDTree.Node2D<Area> node2D = production.nearestArea( latitude: 61.92411,  longitude: 25.748151, item, element, year);

    Assert.assertNull(node2D);
}
```

```java
@Test
public void nearestAreaTest2() {
    Item item = new ItemName("Carrots and turnips");
    Element element = new ElementName("Yield");
    Year year = new Year(1981);
    KDTree.Node2D<Area> result = production.nearestArea( latitude: 9999999,  longitude: 9999999, item, element, year);

    Assert.assertEquals(new AreaName("Finland"), result.getElement());
    Assert.assertEquals( expected: 61.92411, result.getY(),  delta: 0.00001);
    Assert.assertEquals( expected: 25.748151, result.getX(),  delta: 0.00001);
}
```

```java
@Test
public void nearestAreaTest3() {
    Item item = new ItemName("Carrots and turnips");
    Element element = new ElementName("Yield");
    Year year = new Year(1981);
    KDTree.Node2D<Area> result = production.nearestArea( latitude: 61.92411,  longitude: 25.748151, item, element, year);

    Assert.assertEquals(new AreaName("Finland"), result.getElement());
    Assert.assertEquals( expected: 61.92411, result.getY(),  delta: 0.00001);
    Assert.assertEquals( expected: 25.748151, result.getX(),  delta: 0.00001);
}
```

Melhoramentos possíveis

Como melhoramentos a este projeto, podemos indicar os seguintes:

- Realização de uma maior cobertura de testes;
- Aplicação de uma maior programação genérica;
- Melhoramento da documentação do projeto.

Conclusão

Em suma, neste trabalho apreendemos as utilidades da classe árvore binária de pesquisa (BST) apresentada nas aulas tal como a 2d-tree.

Podemos concluir ainda, que quanto mais estudado for o caso e melhor planeado for o trabalho mais facilmente se absorvem as alterações que acontecem ao longo do processo. Posto isto, consideramos que, apesar de ser possível efetuar algumas melhorias neste trabalho, o projeto encontra-se de acordo com os requisitos pedidos.