Instituto Superior de Engenharia do Porto

Projeto 1

ESINF – Licenciatura em Engenharia Informática

Relatório

Turma 2DM e 2DN

Beatriz Santos - 1211178 (2DM)

Daniela Soares - 1211229 (2DM)

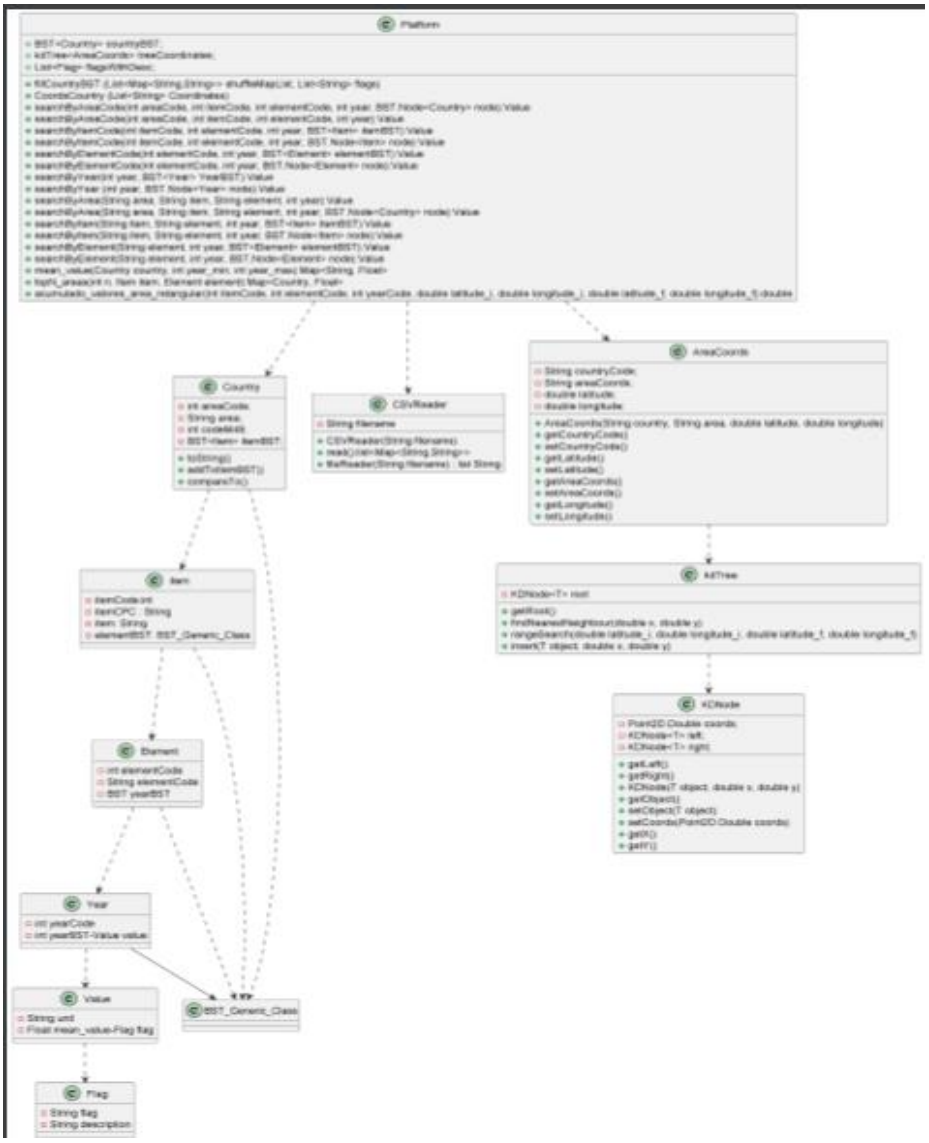
Ruben Silva - 1200546 (2DM)

Daniel Braga – 1200801 (2DN)

Duarte Ramalho - 1221806 (2DM)

2022

Diagrama de Classes



Funcionalidades

Funcionalidade 1

- **Código**

Nesta funcionalidade é pedido para através de arvores de pesquisa carregar a informação relativos aos dados da FAOSTAT que permitam obter os valores {Value,Unit,Flag,Flag Description}, através dos campos {AreaCode,ItemCode,ElementCode,Yeat} e dos campos {Area,Item,Element,Year}.

Começamos por criar uma classe “CSVReader” onde lemos o ficheiro linha a linha. Em seguida converte-se cada linha num mapa, sendo que cada “key” é correspondente a cada elemento do header, e os “Value” associados são os respetivos valores dessa mesma linha. Dentro desta classe

existe também um método auxiliar “fileReader” que lê ficheiros de texto simples, transformando-os em listas de “String”. No nosso caso, o ficheiro que contém informação relativa às “Flags”.

Com toda a informação importada, podemo-nos preocupar com a passagem de dados para a nossa BST. Esta, é feita através do método “fillCountryBST”, que começa por ler a lista de mapas através de um ciclo “For”. Dentro desse ciclo “For”, iremos ler a lista de “Flags” e, posteriormente, associar a “flag” respetiva ao objeto “Country”. O próximo passo é criar um objeto “Country” com os valores corretos, e procurar na “BST” esse mesmo objeto. Caso o encontremos, adicionamos as respetivas “BSTs” – “ItemBST;ElementBST;YearBST”. Em caso contrário, simplesmente adicionamos o “Country” à “BST” de “Countrys”.

Para a segunda parte da nossa funcionalidade, implementámos um método de pesquisa, que consiste em pesquisar dentro das várias “BST” existentes, pelos valores pedidos. Ou seja, ao pesquisarmos por códigos (AreaCode, ItemCode, ElementCode, Year), vamos pesquisar dentro da

```
public void insert(T object, double x, double y) {
    KDNode<T> node = new KDNode<>(object, x, y);
    if (root == null)
        root = node;
    else
        insert(node, root,  divX: true);
}

private void insert(KDNode<T> node, KDNode<T> currentNode, boolean divX) {
    if (node.coords.equals(currentNode.coords))
        return;
    int cmpResult = (divX ? cmpX : cmpY).compare(node, currentNode);
    if (cmpResult == -1)
        if(currentNode.left == null)
            currentNode.left = node;
        else
            insert(node, currentNode.left, !divX);
    else
    if(currentNode.right == null)
        currentNode.right = node;
    else
        insert(node, currentNode.right, !divX);
}
```

BST “Country” pelo objeto que contém o respetivo “AreaCode”. Sendo este encontrado, inicializa-se uma nova pesquisa dentro da BST “Item” presente no “Element” do “Country” encontrado primeiramente. Este processo irá repetir-se, até encontrarmos o “Year” desejado, e consequentemente, os “Values” desejados.Para tratar do ficheiro que continha as coordenadas da área, utilizamos o método *CoordsCountry* que insere os valores numa 2d-Tree através do método *insert* que se encontra na classe kdTree.

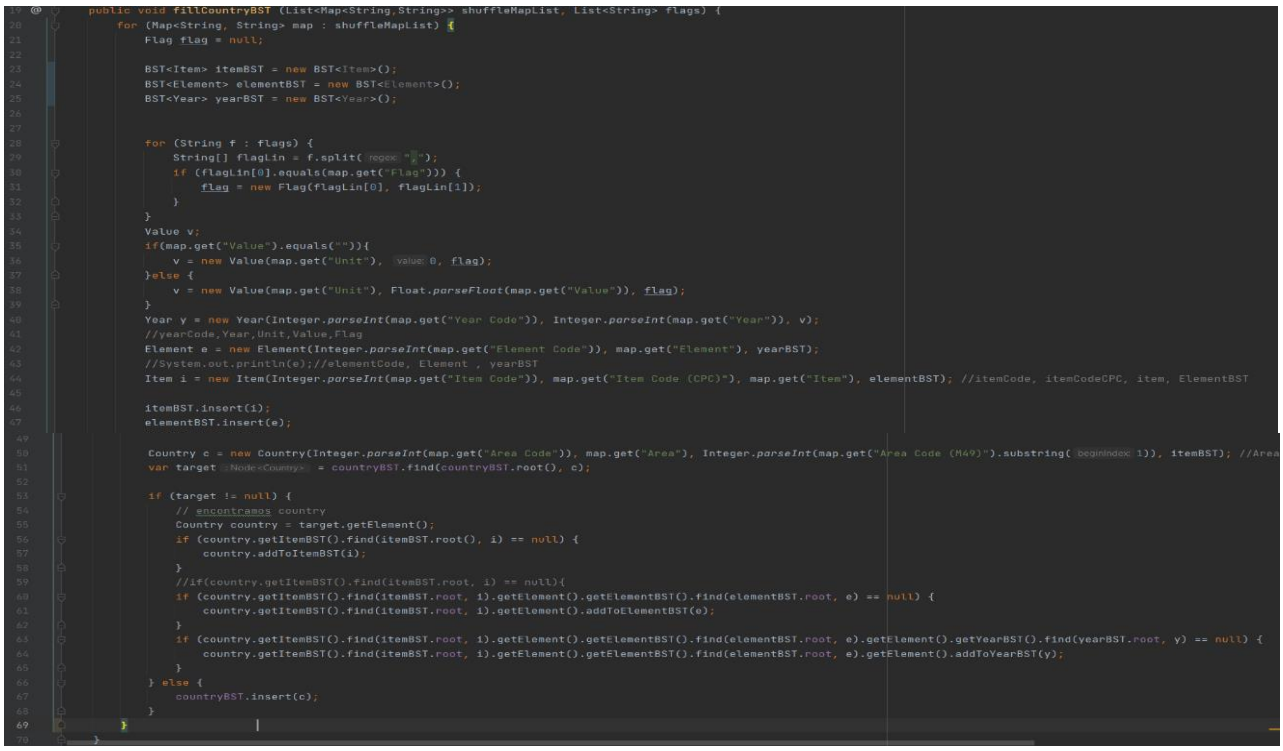
- **Análise da complexidade**

Classe Platform:

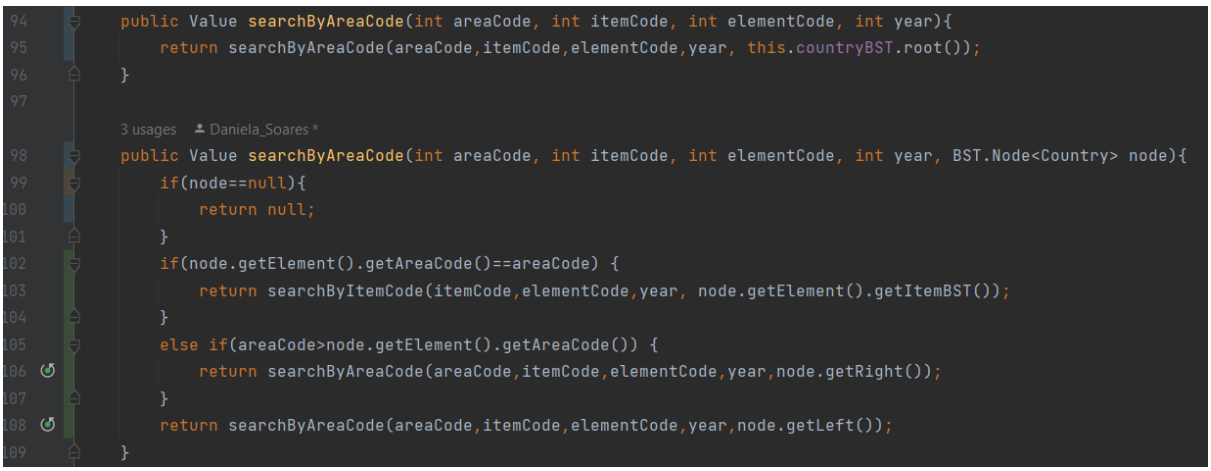
O método fillCountryBST é não determinístico pois existem melhor e pior caso, sendo o melhor caso quando na linha 51 o

Este método no pior caso tem complexidade O(n^3) pois existe um ciclo for das linhas (20 a 69) tem e dentro deste ciclo for encontra-se outro ciclo for das linhas (28 a 33) e dentro desse mesmo for encontram-se operações de ordem O(n).

Este método no melhor caso tem complexidade O (n^2 log n) pois existe um ciclo for das linhas (20 a 69) e, dentro deste ciclo for encontra-se outro ciclo for das linhas (28 a 33). Mais abaixo, nas linhas (53 a 65) encontram-se operações de ordem O(logn) no caso do “find”.



Nos métodos “searchBy” vai existir um primeiro método que tem como função fazer uma chamada recursiva do segundo método, estando estes representados nas linhas (94 a 96) e (98 a 109) respetivamente.



Este método é considerado não determinístico, visto que existe mais que uma situação possível. No pior dos casos, o parâmetro “AreaCode” é igual ao “AreaCode” do “Node” recebido,

visto que executa uma outra chamada recursiva. Isto acontece para todos os métodos semelhantes, o que acaba por desencadear um método de complexidade O(4log(n)).

No melhor dos casos, o "Node" a receber é nulo, e então, a complexidade é reduzida a O(1).

Classe kdTree:

O método *insert* é determinístico e tem complexidade O(1).

Funcionalidade 2

- Análise da complexidade

O método é não determinístico. No melhor caso o país passado por parâmetro não está presente na árvore e será retornado *null*, sendo a complexidade O(log n) devido ao método *find()* que irá percorrer a árvore. No pior caso será O(n^3 log (n)^2) ao percorrer os 3 ciclos *for* e ao entrar na condição dentro do último ciclo *for*, executando duas vezes uma pesquisa numa árvore.

- Código

```
3 usages   Daniel Braga *
public Map<String, Float> mean_value(Country country, int year_min, int year_max){
    float mean, sum;
    int n;
    Map<String, Float> res = new HashMap<>();
    if(countryBST.find(countryBST.root,country)!=null){
        country = countryBST.find(countryBST.root,country).getElement();
        for (Item item : country.getItemBST().inOrder()) {
            for (Element element : item.getElementBST().inOrder()) {
                n=0;
                sum=0;
                for (int i=year_min;i<=year_max;i++) {
                    if(element.getYearBST().find(element.getYearBST().root, new Year(i,i, value: null))!=null){
                        sum+=(element.getYearBST().find(element.getYearBST().root, new Year(i,i, value: null))).getElement().getValue().getValue();
                        n++;
                    }
                }
                mean=sum/n;
                String s = year_min + ".." + year_max + ", " + item.getItem() + ", " + element.getElement() + ", " + mean;
                res.put(s, mean);
            }
        }
        return res.entrySet().stream().sorted(Collections.reverseOrder(Map.Entry.comparingByValue())).
                collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue, (e1, e2)->e1, LinkedHashMap::new));
    }else return null;
}
```

Inicialmente será procurado dentro da árvore de países o país que foi passado por parâmentro, caso este não exista, será retornado null mas caso contrário será feito um ciclo *for each* nos items presentes na árvore do país e dentro deste outro *for each* nos elementos de cada país.

Dentro do último ciclo *for each* serão iniciadas duas variáveis *int* referentes ao número de anos presentes e à soma dos valores, valores estes que serão necessários para calcular a média. Será feito um ciclo *for* que vai percorrer os anos relevantes, iniciando no limite inferior e finalizando no limite superior. Dentro do ciclo *for* vai ser executada uma condição a cada iteração do ano para ver se esse ano está presente na árvore do elemento. Caso a condição seja verificada, será adicionado o valor à soma e incrementada a variável referente ao número de anos presentes.

Após a conclusão do ciclo *for*, depois de percorrer todos os anos relevantes, é calculado o valor da média que será adicionado ao *map* criado inicialmente juntamente com a *string* com as informações que serão apresentadas.

Por fim o *map* será ordenado pelo valor e retornado.

Funcionalidade 3

- Análise da complexidade

O método será não determinístico. O melhor caso será se nenhum país tiver o item dentro da sua árvore de items, sendo a complexidade O(n log (n)). No pior caso vai percorrer o ciclo *for* e executar quatro pesquisas em árvores, sendo a complexidade O(n log (n)^4).

- Código

```
3 usages   Daniel Braga *
public Map<Country, Float> topN_areas(int n, Item item, Element element){
    Map<Country, Float> areas= new HashMap<>();
    for(Country c : countryBST.inOrder()){
        if(c.getItemBST().find(c.getItemBST().root,item)!=null){
            Item item1 = c.getItemBST().find(c.getItemBST().root,item).getElement();
            if(item1.getElementBST().find(item1.getElementBST().root,element)!=null){
                Element element1 = item1.getElementBST().find(item1.getElementBST().root,element).getElement();
                areas.put(c,element1.getYearBST().biggestElement().getValue().getValue());
            }
        }
    }
    if(areas.isEmpty()){
        return null;
    }else return areas.entrySet().stream().sorted(Collections.reverseOrder(Map.Entry.comparingByValue())).limit(n).
            collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue, (e1, e2)->e1, LinkedHashMap::new));
}
```

Inicialmente será feito um ciclo *for* que irá percorrer todos os países presentes na árvore de países, dentro deste será feita uma condição para verificar se o item passado por parâmetro está presente na árvore de items do país. Caso esteja, será feita outra condição, desta vez para verificar se o elemento passado por parâmetro está presente na árvore de elementos do item. Caso esteja, será adicionado ao *map* esse país juntamente com o valor do último ano registado (acedido através do método *biggestElement()*).

Por fim, caso não tenha sido adicionado nenhum país ao *map* será retornado n*ull*, caso contrário será retornado o *map* ordenado pelo valor por ordem decrescente como pedido.

Funcionalidade 4

- Análise da complexidade

Classe KD-Tree:

O método *findNearestNeighbour* é não determinístico. O seu melhor caso é quando encontra logo o vizinho mais próximo e tem complexidade O (log n). No seu pior caso é quando o último elemento analisado é o vizinho tendo complexidade O (n).

Classe Platform:

O método *area_mais_proxima* também é não determinístico. O seu melhor caso é quando o primeiro vizinho encontrado satisfaz todas as condições e tem complexidade O(n). O pior caso é quando a área mais próxima que satisfaz as condições é a última a ser encontrada.

- **Código**

Na funcionalidade 4 é pedido para encontrar a área mais próxima das coordenadas enviadas por parâmetro, devendo apenas considerar as áreas que tenham valores para um dado item, elemento e ano.

Para isso começamos por construir uma kd-tree com as coordenadas das áreas. De seguida vamos utilizar o método *findNearestNeighbour* para encontrar o vizinho mais próximo. Corremos a BST de country para encontrar o nó onde está o país correspondente à área geográfica do vizinho encontrado. Através de ifs e do método find da BST vamos perceber se aquele país teve registos de valores para o item, elemento e ano passado por parâmetros. Se essas condições se verificarem, então encontrámos a área mais próxima, se não teremos de continuar à procura até que encontremos.

```
//ex4
public Object[] area_mais_proxima (double latitude, double longitude, Item item, Element element, Year year){
    Object[] areaInfo = new Object[5];
    AreaCoords vizinho = treeCoordinates.findNearestNeighbour(latitude, longitude);
    if (vizinho!=null) {
        boolean checkVizinho = false;
        for (Country c : countryBST.inOrder()) {
            if (c.getArea().equals(vizinho.getAreaCoords())) {
                if (c.getItemBST().find(c.getItemBST().root, item) != null) {
                    Item item1 = c.getItemBST().find(c.getItemBST().root, item).getElement();
                    if (item1.getElementBST().find(item1.getElementBST().root, element) != null) {
                        Element element1 = item1.getElementBST().find(item1.getElementBST().root, element).getElement();
                        if (element1.getYearBST().find(element1.getYearBST().root, year) != null) {
                            checkVizinho = true;
                            break;
                        }
                    }
                }
            }
        }
        if (checkVizinho) {
            areaInfo = new Object[]{vizinho.getLatitude(), vizinho.getLongitude(), item, element, year};
        } else {
            area_mais_proxima(vizinho.getLatitude(), vizinho.getLongitude(), item, element, year);
```

```
        }
    }else{
        areaInfo = null;
    }
    return areaInfo;
}

public void toString_area_mais_proxima (Object[] areaInfo) {
    if (areaInfo == null){
        System.out.println("Não há informação");
    } else {
        System.out.printf("{latitude: %.4f, longitude: %.4f, Item: %s, Element: %s, Year: %d}", (Double) areaInfo[0], (Double) areaInfo[1],
    }

}
```

O método *area_mais_proxima* irá retornar um object que contém todos os dados que queremos imprimir e de seguida chamamos o método toString para imprimir esses dados.

O algoritmo de busca do vizinho mais próximo (NN) tem como objetivo encontrar o ponto na árvore que está mais próximo de um determinado ponto de entrada. A pesquisa começa na raiz e é executada recursivamente na esquerda e só depois nas subárvores direitas do nó atual.

```java
private T findNearestNeighbour(KDNode<T> node, double x, double y, KDNode<T> closestNode, boolean divX) {
    if (node == null)
        return null;
    double d = Point2D.distanceSq(node.coords.x, node.coords.y, x, y);
    double closestDist = Point2D.distanceSq(closestNode.coords.x, closestNode.coords.y, x, y);
    if (closestDist > d) {
        closestNode.setObject(node.getObject());
    }
    double delta = divX ? x - node.coords.x : y - node.coords.y;
    double delta2 = delta * delta;
    KDNode<T> node1 = delta < 0 ? node.getLeft() : node.getRight();
    KDNode<T> node2 = delta < 0 ? node.getRight() : node.getLeft();
    findNearestNeighbour(node1, x, y, closestNode, !divX);

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

    return closestNode.getObject();
}
```

Funcionalidade 5

- Análise de Complexidade

Classe kdTree:

O método *RangeSearch* é não determinístico.

Classe Plataform:

O método *acumulado_valores_area_retangular* é determínistico. A sua complexidade é $O(n^5)$.

- Código:

Neste funcionalidade é pedido o seguinte: Com recurso à 2d-tree devolva para um Item Code, Element Code e Year Code o acumulado dos valores de produção para uma área geográfica retangular dada por uma latitude inicial, latitude final, longitude inicial e longitude final.

Em primeiro lugar foi implementada a *K-d tree*, na classe *kdTree*, que contém os seguintes métodos: *insert*, *rangeSearch* e o *findNearestNeigbour*. Esta funcionalidade utiliza o método *rangeSearch* que tem a função de devolver numa lista os pontos(coordenadas) que estão contidos na área retangular definida por dois pontos (duas coordenadas), que são definidos por uma latitude e longitude, recebidos por parâmetro. Para encontrar todos os pontos contidos em um determinado retângulo este método começa na raiz e procura recursivamente por pontos em ambas as subárvores usando a seguinte regra: se o retângulo não cruzar o retângulo correspondente a um nó (node), não há necessidade de explorar esse nó ou as suas subárvores. Apenas pesquisa uma subárvore se ela conter um ponto que esteja contido no retângulo.

```java
public List<T> rangeSearch(double latitude_i, double longitude_i, double latitude_f, double longitude_f) {
    return new Object() {
        final List<T> result = new LinkedList<>();
        final double area_retangular = Math.abs(latitude_f-latitude_i) * Math.abs(longitude_f-longitude_i);

        List<T> rangeSearch(KDNode<T> node, boolean divX) {
            if (node == null)
                return result;

            double d = Point2D.distanceSq(latitude_i, longitude_i, latitude_f, longitude_f);

            if (area_retangular >= d)
                result.add(node.getObject());

            double delta = divX ? Math.abs(latitude_f-latitude_i) : Math.abs(longitude_f-longitude_i);
            double delta2 = delta * delta;
            KDNode<T> node1 = delta < 0 ? node.left : node.right;
            KDNode<T> node2 = delta < 0 ? node.right : node.left;
            rangeSearch(node1, !divX);
            if (delta2 < area_retangular) {
                rangeSearch(node2, !divX);
            }
            return result;
        }
    }.rangeSearch(root,  divX: true);
}
```

Depois, na classe *Plataform*, está implementado o método *acumulado_valores_area_retangular* que invoca o *rangeSearch,* a função deste método é através da lista que contém todos os pontos que pertencem à área dada pela latitude inicial, latitude final, longitude inicial e longitude final recebida por parâmetro que é retornada pelo *rangeSearch* calcular o acumulado dos valores de produção para um dado item code, element code e year code. Para fazer este cálculo, percorre-se os países existentes na *binary search tree (BST)* e através de uma condição verifica-se se é igual ao país correspondeste à coordenada existente na lista, se isso acontecer, então é percorrida a árvore dos itens e novamente é utilizada uma condição, mas desta vez para verificar se é igual ao *item code* recebido por parâmetro, se for igual, então percorro a árvore dos *elements* e novamente usada uma condição, mas desta vez que verifica para o *element code* enviado por parâmetro e o mesmo acontece para o *year code* recebido por parâmetro. O cálculo do acumulado é realizado através da variável *sum* que cada vez que os ciclos são executados e as condições verificadas vai somar todos os valores das produções para o item code, element code e year code recebidos para cada área associada às coordenadas contidas na área calculada.

```java
//ex5
public double acumulado_valores_area_retangular(int itemCode, int elementCode, int yearCode, double latitude_i, double longitude_i, double latitude_f, double longitude_f){
    List<AreaCoords> listaCoordenadas = treeCoordinates.rangeSearch(latitude_i, longitude_i, latitude_f, longitude_f);
    double sum=0;
    for(Country c : countryBST.inOrder()){
        for (int i=0; i< listaCoordenadas.size()-1; i++){
            if (c.getArea().equals(listaCoordenadas.get(i).getAreaCoords())){
                for (Item item : c.getItemBST().inOrder()) {
                    if (item.getItemCode() == itemCode) {
                        for (Element element : item.getElementBST().inOrder()) {
                            if (element.getElementCode() == elementCode) {
                                for (Year year : element.getYearBST().inOrder()) {
                                    if (year.getYearCode() == yearCode) {
                                        sum += year.getValue().getValue();
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
    return sum;
}
```

Melhoramentos possíveis

O método fillCountryBST é não determinístico pois existem melhor e pior caso, sendo o melhor caso quando na linha 51 o

O primeiro ciclo *for* das linhas (20 a 69) tem complexidade O(n^2). Isto porque dentro deste ciclo *for* encontra-se outro ciclo *for* das linhas (28 a 33).

O método searchByAreaCode é um método não determinístico pois existem melhor e pior caso, sendo o melhor caso a AreaCode que estamos a comparar ser igual nesse caso a complexidade é de O(1)

O método *findNearestNeighbou*r da kd-Tree não está funcional. Na funcionalidade 4, o método *area_mais_proxima* poderia ter sido feito de outra forma para que fosse mais eficiente. Poderia ter começado por eliminar da 2d-Tree criada os países que não tivessem valores para o item, elemento e ano e só depois procurar o vizinho mais próximo.

O método *rangeSearch* da *kd-Tree* não está funcional. Na funcionalidade 5, no método *acumulado_valores_area_retangular* que invoca o primeiro método apenas estão a ser percorridos os dois primeiros ciclos e a primeira condição para verificar as áreas porque como o *rangeSearch* está a retornar a lista vazia, então não existem informações para a partir daquela área ir determinar o acumulado dos valores de produção por isso no main o valor que aparece é sempre “0.0”. Portanto o método *rangeSearch* tem que ser reformulado e melhorado.