ISEP / LEI
ESINF
2021/2022
G76
Autores – Jorge Ferreira (1201564) Rafael Leite (1201566) Rui Pina (1201568)

RELATÓRIO ESINF

 # Abstract #

Uma empresa de logística de transporte requer um Sistema de software capaz de gerir as suas logísticas. Esta empresa opera tanto no mar, como em terra, por vários continentes e possui diversos armazéns espalhados pelo globo.

 # Introdução #

A finalidade deste relatório é de documentar o desenvolvimento das User Stories, no âmbito de ESINF. Todos os passos percorridos para a elaboração de uma solução para os problemas apresentados, assim como a complexidade temporal de cada algoritmo adotado.

De acordo com as boas práticas aprendidas em ESOFT, no semestre passado foi adotado uma metodologia ágil de Scrum e as práticas de Programação Orientada a Objetos.

Com isto as UserStories foram divididas de maneira justa por todos os membros da equipa.

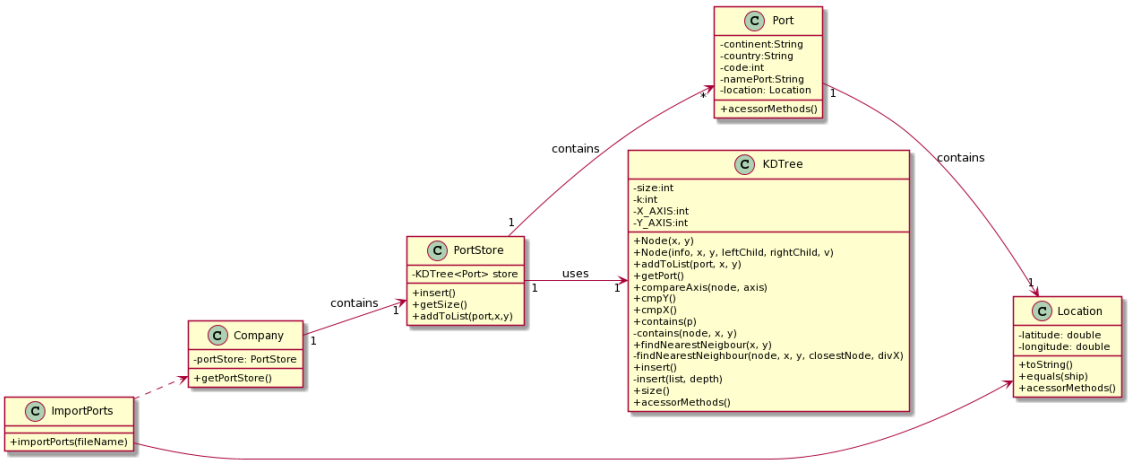
 # Responsáveis das User Stories #

US 201 – Rui Pina 1201568

US 202 – Rafael Leite 1201566

- **US101**

Class Diagram



Em conformidade com as boas práticas adquiridas em ESOFT, nós decidimos usar o padrão Store e, por esse motivo criamos a classe “PortStore” que contem uma instância de uma KDTree, que nos permite encontrar portos perto uns does outros com uma complexidade temporal de O(logn).

- Na Port Store nós iremos precisar guardar instâncias de ports que serão importados por um csv, consequentemente nós criamos a class “Port”, que irá também ter uma classe Location associada que apenas tratará de guardar as informações relativas à sua localização, de acordo com as métricas GRASP.

Algoritmo 1 ImportPorts

Este **Algoritmo** foi dividido em 2 partes lógicas.

```
public static void importPorts(String fileName) {
    PortStore store = App.getInstance().getCompany().getPortStore();
    String continent;
    String country;
    int code;
    String namePort;
    Location location;
    String line;
    String splitBy = ",";
    BufferedReader br = null;
    Port port = null;
    int size = 0;
```

Este pedaço de código apenas inicializa variavéis que serão úteis para o algoritmo em si. Com isto conseguimos ver que cada uma destas linhas têm complexidade temporal de O(1), porque apenas serão executados uma vez.

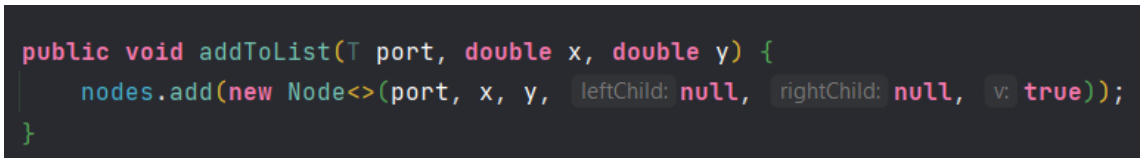
Neste pedaço de Código, conseguimos ver que as primeiras 2 linhas são executadas apenas uma vez, já que não estão dentro de um loop. Após isso todas as outras linhas irão ser executadas n vezes já que estão dentro do while loop, em que no fim do while loop. Como está num loop todas as linhas dentro deste serão executas tantas vezes quantas houver linhas no csv, logo terá complexidade temporal de O(n).

Por fim adicionamos todos os ports que foram criados à KDTree. O que terá uma complexidade temporal de O(nlogn) já que percorremos os Ports todos O(n) e O(logn) para inseri-los na KDTree.

Como isso temos **O(1)** do **primeiro** de pedaço de código, **O(n)** para o **segundo**. Logo O(1+ n + n) ⇔O(n).

Complexidade temporal: O(n)

ALGORITMO 2: DAR INSERT DOS PORTS NA 2DTree



Este método adiciona a uma lista todos os portos criados, para após estes serem ordenados quer pelo x, quer pelo y, consoante o caso, e achar a mediana desses portos. Isto acontece já que a 2DTree não suporta a técnica da rotação da árvore, logo temos que balanceá-la de uma vez só.

Nós no algoritmo em cima utilizamos o método addToList, que irá ser executado no while loop. Como apenas estámos a criar os nodes e a inseri-los numa lista, podemos observar que terá complexidade temporal O(n), já que este método será executo tantas vezes quantas houver linhas no csv.

```java
public void insert() {
    root = insert(nodes,  depth: 0);
}
```

Foi criado um método público sem parâmetros, que apenas irá sem chamada uma vez pelo programa, no imporPorts, que apenas irá dar insert ao root. As folhas da raíz serão adicionadas através da chamada recursiva do método private, que iremos explicar à frente.

```java
private Node<T> insert(List<Node<T>> list, int depth) {
    int sizeOfLists = list.size();
    Node<T> node = null;

    if (depth < 0) {
        return null;
    }
    if (sizeOfLists == 0) {
        return node;
    }

    int axis = depth % k;


    if (axis == X_AXIS) {
        Collections.sort(list, cmpX);
    } else if (axis == Y_AXIS) {
        Collections.sort(list, cmpY);
    }
    Node<T> median;
    int mid = sizeOfLists / 2;
    median = list.get(mid);
```

Este método privado vai ser dividido em 2 partes lógicas, esta primeira que apenas irá fazer "preparar" o node a ser inserido, e a próxima parte, que irá inserir o leaf do node inserido anteriormente no respetivo sítio.

Primeiramente estamos a verificar, se não foi inserido uma depth negativa ou uma lista vazia de nodes contendo os ports. Ambas as situações não são passíveis de serem adicionadas à 2DTree, uma vez que a depth só pode ter números positivos (contendo o 0) e, uma 2DTree de 0 pontos não existe, logo não faz sentido adicioná-los em primeiro lugar.

De seguida vamos calcular o eixo a ser comparado (x ou y no caso da 2DTree). Isto acontece, já que de forma à KDTree estar balanceada e oferecer uma pesquisa do ponto mais próximo de complexidade logarítmica, teremos que buscar a mediana dos pontos a serem adicionados. Essa

mediana será obtida através da lista ordenada por x (caso seja o nível seja par) ou y (caso seja o nível seja ímpar). Esta ordenação será repetida até não haver mais nodes a serem inseridos.

Depois de termos a lista ordenada, quer pelo eixo dos x ou do y, iremos buscar a mediana desses pontos. Para isso apenas iremos fazer a divisão inteira do tamanho da lista de nodes a inserir por 2. Com isso vamos à lista na posição da mediana obtida através da divisão buscar o node.

```
    node = new Node<>(median);
    if (depth == 0) {
        root = node;
    }


    if (sizeOfLists > 2) {

        node.setLeft(this.insert(list.subList( i: 0, mid),  depth: depth+1));
        node.setRight(this.insert(list.subList( i: mid+1, sizeOfLists),  depth: depth+1));
    } else if (sizeOfLists == 2) { //mid must be 1
        if (list.get(0).compareAxis(list.get(1), axis) >= 0) {
            node.setRight(this.insert(list.subList( i: 0,  i1: 1),  depth: depth + 1));
        }else {
            node.setLeft(this.insert(list.subList( i: 0,  i1: 1),  depth: depth + 1));
        }
    }
    size++;

    return node;
}
```

Na segunda parte lógica do código, iremos adicionar o node à 2DTree. Para isso iremos ver, primeiramente, se o depth é igual a 0, e, portanto, a root da árvore. Caso seja atribuímos a mediana da lista ordenada por x à root.

De seguida vamos atribuir as folhas ao último node inserido, que este seja a root da árvore ou apenas de uma sub-árvore. Para isso primeiro certificamo-nos de que pelo menos temos 2 nodes a serem adicionados, se não apenas retornamos o último node.

Caso haja mais de 2 nodes a serem adicionados fazemos uma chamada recursiva à função privada insert, com uma sublista, que caso seja para a subárvore esquerda será do início da lista até ao ponto da mediana. Caso seja da direita será da posição da mediana até ao fim da lista.

Caso seja apenas 2, iremos comparar o eixo para ver de que lado iremos adicionar os nodes. Caso O eixo seja o x iremos adicionar à folha à direita, caso seja o y iremos adicionar à folha à esquerda.

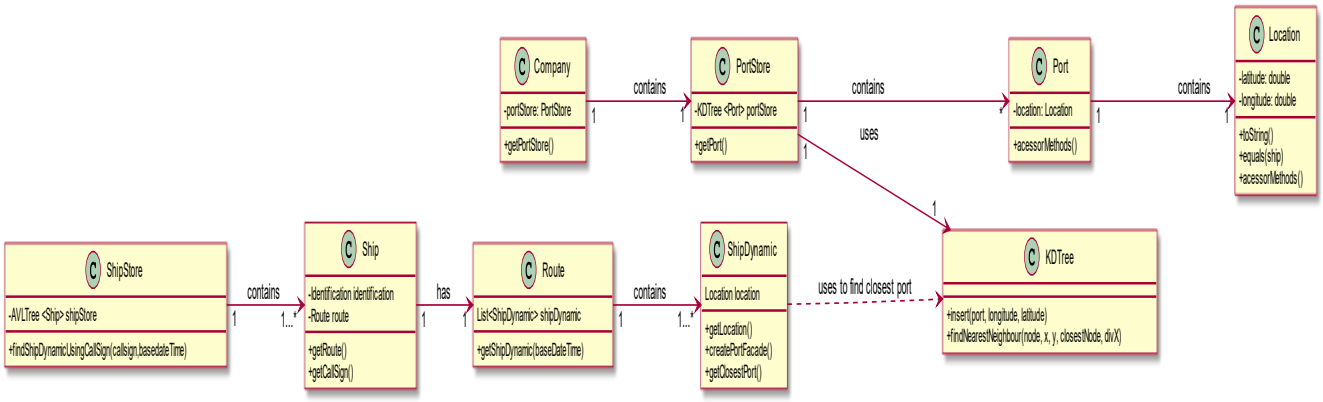
Com isto tudo podemos chegar à conclusão que para obtermos uma 2DTree Balanceada temos que passar por muitos processos até chegarmos ao resultado pretendido, o que pode ser um pouco demoroso, mas que no fim irá compensar já que teremos pesquisas logarítmicas na 2DTree.

A complexidade temporal deste procedimento será O(n) do loop pelas linhas do csv e adicioná-las na lista para depois podermos balancear a 2DTree, visto no primeiro algoritmo e O(nlogn) par ao método insert tem uma complexidade de O(nlogn), uma vez que como temos de balancear a tree, temos de fazer comparações. Essas comparações terão complexidade O(nlogn), já que utilizamos o método Collections.sort . Com isto tudo chegamos à conclusão que o procedimento para chegar à solução da US201 tem complexidade temporal O(n + nlogn) ⇔ O(nlogn).

Complexidade temporal: O(nlogn).

US202

Class Diagram



O class diagram desta US basicamente junta a KDTree, os Ports e os Ships, em que os ports e os ships vão utilizar as suas coordenadas de latitude e longitude para serem feitas pesquisas na kd tree.

findShipDynamicUsingCallSign

```
private ShipDynamic findShipDynamicUsingCallSign(String callSign ,String baseDateTime){
    for (Ship ship:store.inOrder()){
        if (ship.getShipId().getCallsign().equals(callSign)){
            for (ShipDynamic shipDynamic: ship.getRoute().getRoute()){
                if (shipDynamic.getBaseDateTime().equals(baseDateTime)){
                    return shipDynamic;
                }
            }
        }
    }
    return null;
}
```

O método utilizado percorre todos os ships ate encontrar o certo e depois percorre todas as suas mensagens ate encontrar a correta por isso tem complexidade temporal de O(n)(numero de ships) *o(n)(numero de mensagens) no pior caso.

findNearestNeighbour

```
private T findNearestNeighbour(Node<T> node, double x, double y,
                               Node<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.info=node.getInfo();
        closestNode.coords.x=node.coords.getX();
        closestNode.coords.y=node.coords.getY();
    }
    double delta = divX ? x - node.coords.x : y - node.coords.y;

    double delta2 = delta * delta;
    Node<T> node1 = delta < 0 ? node.left : node.right;
    Node<T> node2 = delta < 0 ? node.right : node.left;
    findNearestNeighbour(node1, x, y, closestNode, !divX);
    if (delta2 < closestDist) {
        findNearestNeighbour(node2, x, y, closestNode, !divX);

    }

    return closestNode.info;
}
```

O método findNearestNeighbour,vai percorrer uma KD-Tree de forma recursiva comparando a distancia de cada ponto ao ponto desejado, e verificando se o resto dos pontos podem ser maiores ou menores, parando quando não podem ser menores tendo no pior caso complexidade temporal de O(n),mas apenas no pior caso, mas tendo no melhor caso O(log n).