

Licenciatura em Engenharia Informática



Relatório Trabalho Prático 2

Francisca Moraes (1181122)

João Marques (1192221)

Paulo Couto (1200587)

Dezembro 2021

Table of Contents

US 201	4
Análise de complexidade	7
US 202	8
Análise de complexidade do método principal da <i>user story</i>	10
Análise de complexidade do método <i>nearstPort</i>	11
Análise de complexidade do método <i>findNearestNeighbour</i>	11

Índice de figuras

Figura 1 – Modelo de domínio relevante us201.....	4
Figura 2 – Short system diagram us201.....	4
Figura 3 – System diagram.....	5
Figura 4 – Class diagram.....	5
Figura 5 - Class CsvReader, method readPorts.....	6
Figura 6 - Class KdTreePort, method insertPorts.....	7
Figura 7 - Class KdTre, method buildTree.....	7
Figura 8 – diagrama de classes relevante us202.....	8
Figura 9 – Short system diagram us202.....	8
Figura 10 – System diagram us202.....	9
Figura 11 – Class diagram us202.....	9
Figura 12 – Método principal da user story.....	10
Figura 13 – Método nearstPort.....	11
Figura 14 – Método findNearestNeighbour.....	11

US 201

- As a Port manager, I wish to import ports from a text file and create a 2D-tree with port locations.

Nesta User Story, foi-nos pedido que como um Gestor de Portos, importássemos um ficheiro csv contendo todos os portos, e em seguida, os colocássemos numa 2d Tree.

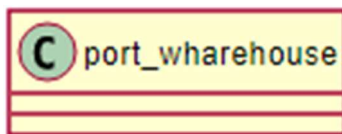


Figura 1 -Modelo de domineo relevante

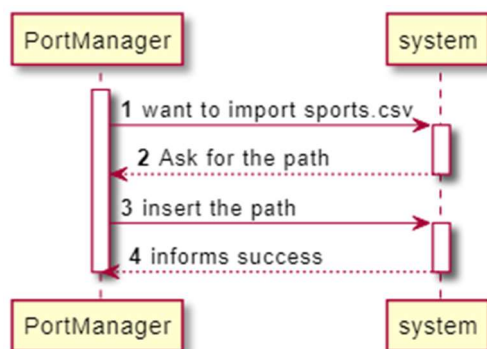


Figura 2 -Short system diagram us201

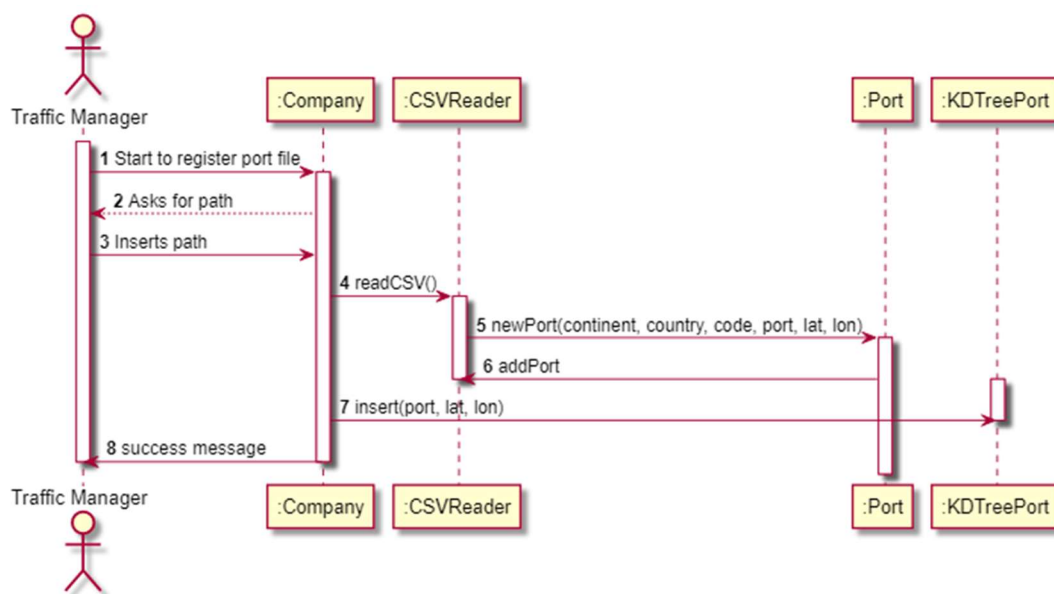


Figura 3 -System diagram us201

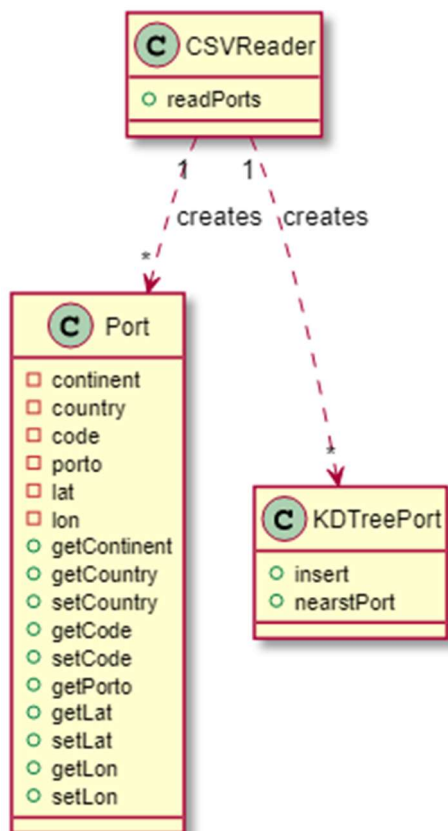


Figura 4 – Class diagram us201

Para tal, criámos o método `readPorts`, que recebe uma string com o ficheiro csv, onde iremos percorrer o ficheiro linha a linha, colocando a informação em cada variável do Porto, criando assim um `ArrayList` de Portos.

Este método tem complexidade de $O(\log n)$, visto ter um ciclo `while`.

```
public static ArrayList<Port> readPorts(String path) throws Exception {
    ArrayList<Port> portArray = new ArrayList<>();
    try (BufferedReader br = new BufferedReader(new FileReader(path))) {
        String line = br.readLine();
        while ((line = br.readLine()) != null) {
            String values[] = line.split("regex: ", ",");
            Port port = new Port(
                values[0],          //continent
                values[1],          //country
                Integer.parseInt(values[2]), //code
                values[3],          //port
                Double.parseDouble(values[4]), //lat
                Double.parseDouble(values[5])); //lon

            portArray.add(port);
        }
        return portArray;
    } catch (FileNotFoundException e){
        System.out.println("File not found!");
        return null;
    }
}
```

Figura 5 - Class `CsvReader`, method `readPorts`

Em seguida, de forma a inserir o `ArrayList` de Ports, retornado pelo método `readPorts`, numa 2d Tree, utilizamos o método “`insertPorts`”. Este método tem uma complexidade de $O(n)$, pois contém um ciclo `for`, no entanto, este método chama um outro método, chamado `buildTree`. Este método também tem complexidade de $O(n)$ devido à utilização de recursividade. Assim, o método “`insertPorts`” teria uma complexidade de $O(n) + O(n)$ que seria = $2 O(n)$, no entanto, em notação Big

Oh, não interessa o valor das constantes, pelo que ficamos então com o valor de $O(n)$.

```
public void insertPorts() {  
    List<NodeKdTree<Port>> nodes = new ArrayList<>();  
    for (Port port : portArray) {  
        NodeKdTree<Port> node = new NodeKdTree<>(port, port.getLat(), port.getLon());  
        nodes.add(node);  
    }  
    portTree.buildTree(nodes);  
}
```

Figura 6 - Class KdTreePort, method insertPorts

```
public void buildTree(List<NodeKdTree<T>> nodes) {  
    root = (Object) buildTree(divX, nodes) → {  
        if (nodes == null || nodes.isEmpty())  
            return null;  
        Collections.sort(nodes, divX ? cmpX : cmpY);  
        int mid = nodes.size() >> 1;  
        NodeKdTree<T> node = new NodeKdTree<>();  
        node.coords = nodes.get(mid).coords;  
        node.info = nodes.get(mid).info;  
        node.left = buildTree(!divX, nodes.subList(0, mid));  
        if (mid + 1 <= nodes.size() - 1)  
            node.right = buildTree(!divX, nodes.subList(mid+1, nodes.size()));  
        return node;  
    }.buildTree(divX: true, nodes);  
}
```

Figura 7 - Class KdTre, method buildTree

Análise de complexidade

A complexidade da US 201 é então:

$$O(\log n) + O(n) = O(n)$$

US 202

As a Traffic manager, I wish to find the closest port of a ship given its CallSign, on a certain DateTime with the acceptance criteria of using the 2dTree to find the port.

Nesta user story era pedido que enquanto gestor de trafico fosse possível encontrar um navio dado como parâmetro o CallSign e uma data com o objetivo de obter o nome do porto mais próximo desse navio.

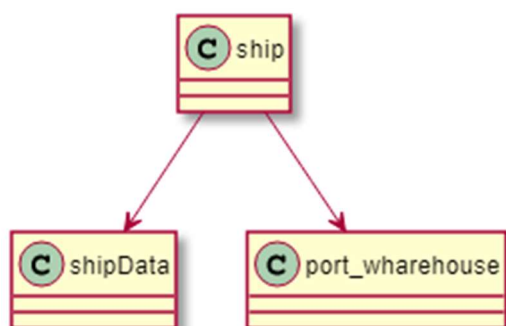


Figura 8 – diagrama de classes relevante us202

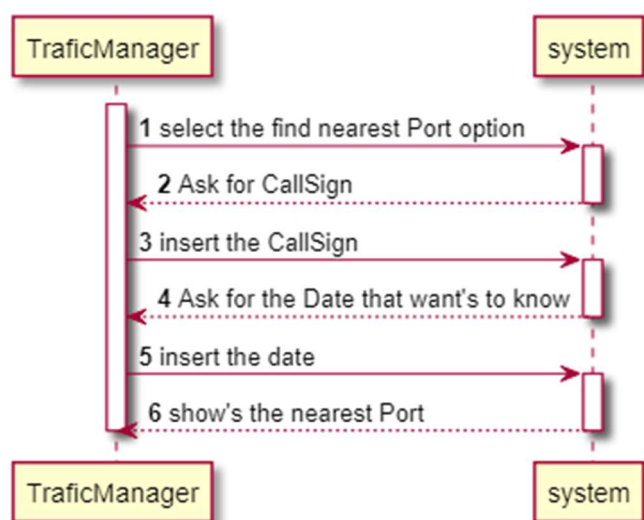


Figura 9 – Short system diagram us202

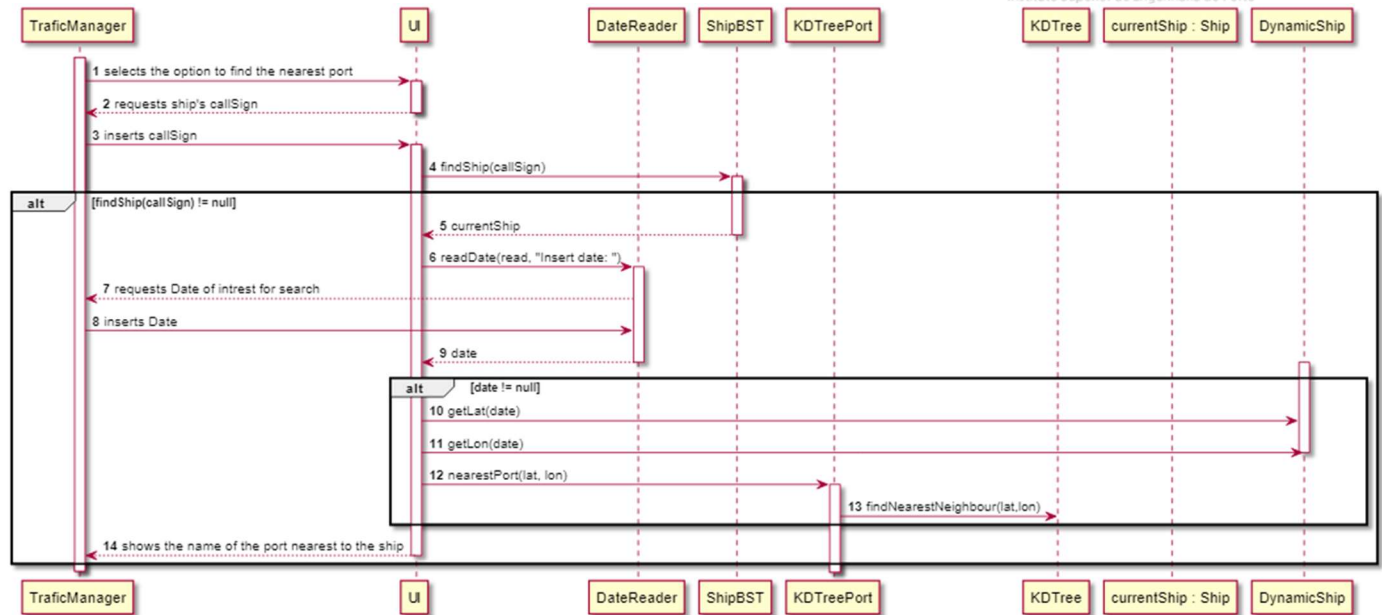


Figura 10 – System diagram us202

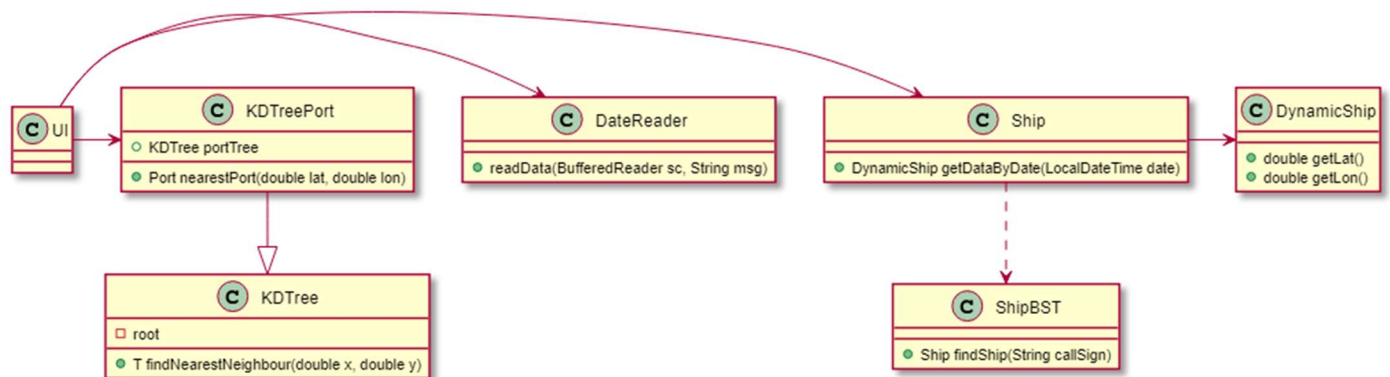


Figura 11 – Class diagram us202

Neste método o utilizador coloca o CallSign e é verificada a existência desse navio de seguida é pedido ao utilizador que coloque a data que deseja fazer a procura e nessa altura é realizada uma procura na 2dTree e é apresentado o nome do porto que se encontra mais próximo do navio naquele instante.

```

LocalDateTime date;
Ship currentShip;
ShipBST bst = new ShipBST();
bst.insert();
System.out.print("Insert the ship's CallSign:");
String callSign = read.readLine();
if (bst.findShip(callSign) != null) {
    currentShip = bst.findShip(callSign);
    date = DateReader.readDate(read, msg: "Insert date: ");
    DynamicShip data = currentShip.getDataByDate(date);
    if (data != null) {
        Port nearestPort = portTree.nearestPort(data.getLat(), data.getLon());
        System.out.println("The nearest port is "+nearestPort.getPorto());
    }
} else {
    System.out.println("Ship not found");
}

```

Figura 12 – Método principal da user story

Análise de complexidade do método principal da user story

A complexidade deste método é de $O(n)$ pois temos um *if* com seguido de um *else*, sendo que o *if* que se encontra no seu interior tem uma complexidade de $O(1)$ a complexidade é como antes referido $O(n)$.

Foi necessário utilizar o método *nearestPort* que irá fazer o retorno do Porto que esta no nó que foi retornado pela função *findNearestNeighbour*.

```
public Port nearestPort(double lat, double lon){
    return (Port) portTree.findNearestNeighbour(lat,lon);
}
```

Figura 13 – Método *nearestPort*

Análise de complexidade do método *nearestPort*

A complexidade deste método é de $O(1)$ pois apenas se limita a fazer um *cast* e um *return*.

De seguida, temos o método *findNearestNeighbour* que vai fazer uma pesquisa pela rvore para isso é utilizado cálculos de distâncias entre dois pontos e a comparação entre esses dois pontos para a decisão de qual se encontra mais próximo.

```
public T findNearestNeighbour(double x, double y) {
    return findNearestNeighbour(root, x, y, divX: true);
}

private T findNearestNeighbour(NodeKdTree<T> fromNode, final double x, final double y, boolean divX) {
    return new Object() {
        double closestDist = Double.POSITIVE_INFINITY;

        T closestNode = null;

        T findNearestNeighbour(NodeKdTree<T> node, boolean divX) {
            if (node == null)
                return null;
            double d = Point2D.distanceSq(node.coords.x, node.coords.y, x, y);
            if (closestDist > d) {
                closestDist = d;
                closestNode = node.info;
            }

            double delta = divX ? x - node.coords.x : y - node.coords.y;
            double delta2 = delta * delta;
            NodeKdTree<T> node1 = delta < 0 ? node.left : node.right;
            NodeKdTree<T> node2 = delta < 0 ? node.right : node.left;
            findNearestNeighbour(node1, !divX);
            if (delta2 < closestDist) {
                findNearestNeighbour(node2, !divX);
            }
            return closestNode;
        }
    }.findNearestNeighbour(fromNode, divX);
}
```

Figura 14 – Método *findNearestNeighbour*

Análise de complexidade do método *findNearestNeighbour*

A complexidade deste método é de $O(\log n)$ pois trata-se de uma árvore balanceada.