Licenciatura em Engenharia Informática

Relatório Trabalho Prático 2

Diogo Guedes (1170617)
João Teixeira (1180590)
Rui Gonçalves (1191831)

Dezembro
2021

Exercício 1

Para o primeiro exercício era pedido para ler todos os portos de um ficheiro e, de seguida, guardar a infirmação numa 2dTree com as localizações dos portos.

Utilizamos o método "readPortCSV" para guardar os Portos num ArrayList que tem complexidade **O(logn).**

```
public static ArrayList<Port> readPortCSV(String path) {
    ArrayList<Port> portArrayList = 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 newPort = new Port(values[0],
                    values[1],
                    Integer.parseInt(values[2]),
                    values[3],
                    Double.parseDouble(values[4]),
                    Double.parseDouble(values[5]));

            portArrayList.add(newPort);
        }
        return portArrayList;
    } catch (Exception e) {
        System.out.println("No ports were imported - Try again.\n");
        return null;
    }
}
```

De seguida, para os inserir numa 2dTree, utilizamos o método "insertPorts" que, por sua vez, chama o método "buildTree" sendo que, este último método, têm a complexidade **O(n)** devido às chamas recursivas**.**

Assim, o método para inserir os portos, fica com a complexidade:

O(n) + O(n) = 2O(n) = O(n)

```
private static void insertPorts() {
    List<Node<Port>> nodes = new ArrayList<>();
    for (Port port : portsArray) {
        Node<Port> node = new Node<~>(port, port.getLat(), port.getLon());
        nodes.add(node);
    }
    portTree.buildTree(nodes);
}
```

```
public void buildTree(List<Node<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;
            Node<T> node = new Node<>();
            node.coords = nodes.get(mid).coords;
            node.object = nodes.get(mid).object;
            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);
}
```

Finalmente, a análise de complexidade deste exercício é:

O(logn) + O(n) = O(n)

Exercício 2

Para o segundo exercício, era pedido que, enquanto gestor de trafico, fosse possível encontrar um navio introduzindo o CallSign como parâmetro e uma data com o objetivo de obter o nome do porto mais próximo desse navio.

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

Começamos por utilizar o método "readDate" que tem apenas a complexidade **O(1)**

```java
/**
 * Reads a Date from user input with error checking
 *
 * @param sc scanner to read input from the user
 * @return date read from valid user input or null otherwise
 */
public static LocalDateTime readDate(Scanner sc, String msg) {
    DateTimeFormatter format = DateTimeFormatter.ofPattern("dd/MM/yyyy HH:mm");
    System.out.print("Format: dd/MM/yyyy HH:mm\n");
    System.out.print(msg);
    LocalDateTime dateTime;

    String str = sc.nextLine();

    try {
        dateTime = LocalDateTime.parse(str, format);
    } catch (Exception e) {
        System.out.print("Invalid date.\n");
        return null;
    }
    return dateTime;
}
```

De seguida, usamos o método "getDataByDate" que tem a complexidade **O(n)** devido ao seu ciclo

```java
public ShipData getDataByDate(LocalDateTime date) {
    for (ShipData data : this.dynamicShip) {
        LocalDateTime currentDate = data.getDateTime();
        if (currentDate.isEqual(date)) {
            return data;
        }
    }
    return null;
}
```

Por fim, é chamado o método "findNearestNeighbour" para encontrar o porto mais próximo que tem a complexidade **O(logn)**

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

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

        T findNearestNeighbour(Node<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.getElement();
            }
            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, !divX);
            if (delta2 < closestDist) {
                findNearestNeighbour(node2, !divX);
            }
            return closestNode;
        }
    }.findNearestNeighbour(fromNode, divX);
}
```

Concluindo, este exercício tem a complexidade:

O(1) + O(n) + O(logn) = O(n)