Relatório do Trabalho Prático 3

Instituto Superior de Engenharia do Porto

Estruturas de Informação

2020/2021

1181743 - Guilherme Daniel

1171589 - Lucas Sousa

Classes Definidas:

Para a elaboração do trabalho prático foram definidas 7 classes diferentes:

- **ReadFile**: classe para ler de um ficheiro e transformar essa informação em elementos da tabela periódica;
- **Element**: classe com os atributos e composição do elemento;
- **EletronicConfigurationAssembler**: classe com a configuração eletrónica e o número de ocorrências;
- **TreesPesquisaPorCampos**: classe que constrói árvores e com diferentes algoritmos necessários à execução das alíneas “1.a” e “1.b” do trabalho prático.
- **TreeElectronConfiguration**: classe que constrói árvores e que possui diferentes algoritmos necessários à execução das alíneas “2.a”, “2.b”, “2.c” e “2.d” do trabalho prático.
- **SearchFields**: classe que contém todas as inner classes necessárias ao primeiro exercício, para ser possível comparar elementos por diferentes campos.

A criação destas classes facilitou a leitura dos ficheiros de texto assim como a resolução com eficácia dos exercícios.

Para além das 7 classes definidas, são também implementadas as classes base do package “Tree”. Estas imprescindíveis, para a resolução do trabalho prático.

Análise de Complexidade:

Exercício 1.A:

Para evitar a repetição da análise algorítmica nos métodos utilizados no exercício 1 alínea a, apenas serão demonstrados algumas das funcionalidades desta alínea:

```
/**
 * Faz uma pesquisa pelo número atómico
 *
 * @param e
 * @return
 */
public Node<Element> searchAtomicNumber(int atomicNumber) {
    insertByAtomicNumber();
    return findNodeByAtomicNumber(root, atomicNumber);
}
```

Figura 1- Método intermediário de pesquisa de um elemento

No método de pesquisa de um elemento, primeiro cria-se uma árvore binária comparando os elementos pelo respetivo campo e só depois se realiza a pesquisa. Foram criadas várias classes que estendem a classe Element, de modo a ser possível comparar elementos por diferentes campos.

```
/**
 * Insere valores a uma árvore pelo seu número atómico
 */
protected void insertByAtomicNumber() {
    cleanTree();
    List<ElementAtomicNumber> l = new ArrayList<>();
    for (Element e : elementsList) {
        ElementAtomicNumber at = new ElementAtomicNumber(e);
        l.add(at);
    }
    for (ElementAtomicNumber r : l) {
        insert(r);
    }
}
```

Figura 2- Inserção de elementos numa árvore por atomic number

No método insertByAtomicNumber, chamado no método da primeira imagem, está a inserir-se elementos numa árvore binária AVL. A complexidade temporal deste método será de O(log(n)), contudo, antes desta inserção, é realizada uma iteração por uma lista, logo a sua complexidade temporal total de O(n).

```
private Node<Element> findNodeByAtomicNumber(Node<Element> node, int value) {
    if (node == null) {
        return null;
    }

    if (node.getElement().getAtomicNumber() == value) {
        return node;
    }

    if (node.getElement().getAtomicNumber() > value) {
        return findNodeByAtomicNumber(node.getLeft(), value);
    }

    return findNodeByAtomicNumber(node.getRight(), value);

}
```

Figura 3- Método que pesquisa por um nó com o respetivo número atómico

A complexidade deste método chamado pelo método principal é de O(log(n)).

Concluindo, o método **searchAtomicNumber**, como os outros métodos search, vão ter todos uma complexidade total de **O(n),** devido à lista que está a ser criada em cada insert.

Exercício 1.B:

Para resolvermos o que nos foi pedido nesta alínea, criamos o método “searchByRange”, que recebe por parâmetro um intervalo (limite superior “max” e inferior “min”), de massas atómicas. Que usamos para pesquisar e encontra todos os elementos que possuam massas atómicas compreendida dentro desse mesmo intervalo.

```
public List<Element> searchByRange(int min, int max) {
    insertByAtomicMass();
    List<Element> shorterElementsList = new ArrayList<>();
    TreeInRange(root(), min, max, shorterElementsList);
    if (shorterElementsList.isEmpty()) {
        return null;
    }
    Collections.sort(shorterElementsList, new compareByDiscovererAndYearOfDiscovery());
    return shorterElementsList;
}
```

Primeiramente chamamos o método “insertByAtomicMass”, que nos vai criar uma árvore balanceada e ordenada pelos valores de massa atómica. Em seguida criamos uma lista “shorterElementsList” que vai ser preenchida com todos os elementos compreendidos no intervalo já mencionado. A lista é enviada por parâmetro no método “TreeInRange” assim como a “root” da árvore, e o intervalo de valores. É neste método que vamos preencher a “shorterElementsList”, este é um método recursivo com uma travessia “Pre-Order”, uma vez que primeiramente visita a raiz e só depois as subárvores à esquerda e depois à direita.

```
private static void TreeInRange(Node<Element> node, int min, int max, List<Element> ShorterElementsList) {
    if (node == null) {
        return;
    }

    if (node.getElement().getAtomicMass() >= min && node.getElement().getAtomicMass() <= max) {
        if (!ShorterElementsList.contains(node.getElement())) {
            ShorterElementsList.add(node.getElement());
        }
        TreeInRange(node.getLeft(), min, max, ShorterElementsList);
    }
    if (node.getElement().getAtomicMass() > min) {
        TreeInRange(node.getLeft(), min, max, ShorterElementsList);
    }
    if (node.getElement().getAtomicMass() < max) {
        TreeInRange(node.getRight(), min, max, ShorterElementsList);
    }

}
```

Se a lista retornar vazia significa que não temos elementos para aquele intervalo e o nosso método termina, caso contrário partimos a ordenação da lista. Graças a “Collections.sort”, que ordena o conjunto de elementos por “Discoverer” em ordem crescente e “Year of Discovery” em ordem decrescente.

```
private class compareByDiscovererAndYearOfDiscovery implements Comparator<Element> {

    @Override
    public int compare(Element e1, Element e2) {
        if (e1.getDiscoverer().compareTo(e2.getDiscoverer()) == 0) {
            if (e1.getYearOfDiscovery() < e2.getYearOfDiscovery()) {
                return 1;
            }
            if (e1.getYearOfDiscovery() > e2.getYearOfDiscovery()) {
                return -1;
            }
            return 0;
        }
        return e1.getDiscoverer().compareTo(e2.getDiscoverer());
    }
}
```

Para o sumário pedido recorremos ainda ao método “summaryListByTypeAndPhase” que agrupa os elementos por “Type” e “Phase”, recebendo por parâmetro a lista ordenada e retornado uma tabela como foi pedido.

Vamos analisar a Complexidade algorítmica por detrás do método “searchByRange”, começando pelo método “TreeInRange”. Este método é determinístico, e apresenta uma complexidade temporal de O(n), em que n é o número total de elementos presente na árvore pois só é necessária uma travessia na árvore. Já no caso do método “insertByAtomicMass” temos uma complexidade de O(n) para os dois ciclos for o que significa que este método apresenta também uma complexidade temporal de O(n). O que nos permite concluir que a complexidade do nosso método “searchByRange” seja de O(n), sendo esta uma complexidade linear em que a performance é diretamente proporcional com o tamanho da informação introduzida. Já no método auxiliar “summaryListByTypeAndPhase”, estamos perante uma complexidade temporal de O(n^2), uma vez que usamos 2 ciclos for para preencher o “summary”, esta complexidade não é tão boa como a linear, mas é a necessaria para o preenchimento desta tabela.

Exercício 2.A:

Para a resolução deste exercício, primeiro será apresentado o método principal, que se encarrega de devolver um TreeMap ordenado pelo número de repetições

```java
/**
 * Método que devolve a árvore binária de pesquisa por ordem decrescente de
 * configurações eletrónicas
 *
 * @return
 */
public TreeMap<Integer, LinkedList<String>> orderedTreeMapByEletronicConfigurationRepetition() {
    createTreeWithSpacesInsideConfiguration();
    TreeMap<Integer, LinkedList<String>> tree = new TreeMap<>(Collections.reverseOrder());
    Map<Integer, List<EletronicConfigurationAssembler>> nodes = nodesByLevel();

    for (Integer i : nodes.keySet()) {
        for (EletronicConfigurationAssembler e : nodes.get(i)) {
            if (e.getOcorrences() > 1 && e.getEletronicConfiguration() != "0") {
                tree.put(e.getOcorrences(), null);
            }

        }

    }

    for (Integer i : tree.keySet()) {
        LinkedList<String> it = new LinkedList<>();
        for (Integer j : nodes.keySet()) {
            for (EletronicConfigurationAssembler e : nodes.get(j)) {
                if (e.getOcorrences() == i) {
                    it.add(e.getEletronicConfiguration());
                }
            }
        }
        Collections.sort(it);
        tree.put(i, it);
    }
    cleanTree();
    return tree;
}
```

Figura 4- Método principal que retorna o TreeMap

Repare-se que o método chama um método **createTreeWithSpacesConfiguration**.

```java
private void createTreeWithSpacesInsideConfiguration() {
    cleanTree();
    Iterator<Element> it = elementsList.iterator();
    while (it.hasNext()) {
        Element current = it.next();
        String splitter[] = current.getElectronConfiguration().split("\\s+");

        String previousValues = "";
        for (int i = 0; i < splitter.length; i++) {
            if (i == 0) {
                previousValues = splitter[i];
                if (splitter[i] != "0") {
                    insertWithIncrement(new EletronicConfigurationAssembler(splitter[i], 1));
                }
            } else {
                if (splitter[i] != "0") {
                    previousValues += " " + splitter[i];
                }
                insertWithIncrement(new EletronicConfigurationAssembler(previousValues, 1));
            }
        }
    }

}
```

Figura 5- Método que cria uma árvore tendo em conta os espaços entre as configurações

O método vai iterar por uma lista que contém todos os elementos (elementsList), e dentro dessa interação, irá percorrer um array que separa as configurações de cada elemento tendo em conta os espaços. Depois, insere para uma nova árvore do tipo EletronicConfigurationAssembler, que incrementa cada configuração eletrónica igual. Este método tem uma complexidade de **O(n^2.log(n))**, por iterar por dois ciclos e por inserir numa árvore. O primeiro ciclo do método orderedTreeMapByEletronicConfigurationRepetition tem uma complexidade de O(n^2) e o segundo ciclo tem uma complexidade de O(n^3*log(n)), por conter também o Collections.sort que apresenta uma complexidade de O(n*log(n)). Por isso, conclui-se que a complexidade do método completo será de **O(n^3*log(n)).**

Exercício 2.B:

```java
public AVL<EletronicConfigurationAssembler> createAVLDescendingOrder() {
    AVL<EletronicConfigurationAssembler> eletronicAVL = new AVL();
    BST<EletronicConfigurationAssembler> copy = new BST();
    List<EletronicConfigurationAssembler> list = new ArrayList<>();

    TreeMap<Integer, LinkedList<String>> tree = orderedTreeMapByEletronicConfigurationRepetition();

    for (Integer i : tree.keySet()) {

        if (i > 2) {
            for (String e : tree.get(i)) {
                EletronicConfigurationAssembler eca = new EletronicConfigurationAssembler(e, i);
                list.add(eca);

            }
        }
    }
    for (EletronicConfigurationAssembler e : list) {
        eletronicAVL.insert(e);
    }
    return eletronicAVL;

}
```

Figura 6- Método createAVLDescendingOrder

Este método tem como objetivo devolver uma tabela ordenada por ordem crescente do número de repetições superiores a 2. Para isso, recorreu-se ao método criado anteriormente **orderedTreeMapByEletronicConfigurationRepetition**, com uma complexidade temporal de **O(n^3*log(n)).** O método tem também um dois ciclos for, de complexidade O(n^2). Concluindo, a complexidade temporal total do método é de O(n^3*log(n)). Este método simplesmente está encarregue de devolver uma árvore binária de busca em que os seus valores são inseridos por ordem decrescente. Sabendo que uma árvore AVL irá balancear os nós inseridos, não seria necessário estarmos preocupados com a ordem de inserção de valores na árvore, apenas seria necessário garantir que apenas as configurações com repetições superiores a 2 eram adicionadas.

Exercício 2.C:

Nesta alínea tínhamos de desenvolver um método que devolva os valores das duas configurações eletrónicas mais distantes na árvore e a respetiva distância, sendo esta distância o número de ramos que distam uma da outra. O método principal é "getMaxDistance" este recebe por parâmetro uma árvore de configurações eletrónicas e para termos uma boa resolução tornamos esta árvore a principal, graças ao "secondTree" e chamamos para retornar o resultado do método "getTwoEletronicConfigurationMaxDistance" que nos vai permitir solucionar esta alínea.

```
// ------------------ EXERCICIO 2 ALINEA C ------------------
/**
 * Método que devolve os valores das duas configurações electrónicas mais
 * distantes na árvore e a respectiva distância.
 *
 * @param tree árvore de pesquisa dos valores das duas configurações electrónicas mais distantes
 * @return a primera coluna da tabela de distancia entre folhas na árvore tree ordenada por distância em ordem desecendente
 */
public Map.Entry<Integer, List<EletronicConfigurationAssembler>> getMaxDistance(AVL<EletronicConfigurationAssembler> tree){
    secondTree(tree);
    return getTwoEletronicConfigurationMaxDistance();
}
```

Figura 7- Método que determina as cidades com mais centralidade

Já no método "getTwoEletronicConfigurationMaxDistance", vamos utilizar o método recursivo "getAllLeafs", que nos preenche a lista "listOfLeafs".

Apos sabermos todas as folhas da árvore, partimos para a combinação destas duas a duas, uma vez que o caminho de maior distância vai ser composto sempre por duas folhas. Esta informação é guardada num "TreeMap", em que a chave é a distância entre as folhas. No final retornamos só a primeira entrada no TreeMap, já que este está ordenado de ordem decrescente de distância, ou seja, o caminho mais distante vai estar sempre na primeira posição.

```
/**
 * Método auxiliar ao getMaxDistance
 *
 * @return a primera coluna da tabela de distancia entre folhas na árvore tree ordenada por distância em ordem desecendente
 */
private Map.Entry<Integer, List<EletronicConfigurationAssembler>> getTwoEletronicConfigurationMaxDistance() {
    List<EletronicConfigurationAssembler> listOfLeafs = new ArrayList<>();
    TreeMap<Integer, List<EletronicConfigurationAssembler>> allPathsSizesBetweenLeafs = new TreeMap<>(Collections.reverseOrder());
    getAllLeafs(root(), listOfLeafs);

    for (int i = 0; i < listOfLeafs.size() - 1; i++) {
        for (int j = i + 1; j < listOfLeafs.size(); j++) {
            int distance = findDistance(root(), listOfLeafs.get(i), listOfLeafs.get(j));
            List<EletronicConfigurationAssembler> twoElements = new ArrayList<>();
            twoElements.add(listOfLeafs.get(i));
            twoElements.add(listOfLeafs.get(j));
            allPathsSizesBetweenLeafs.put(distance, twoElements);
        }
    }
    return allPathsSizesBetweenLeafs.firstEntry();
}
```

Figura 8- Método getTwoEletronicConfigurationMaxDistance

Para termos a distância entre os elementos da árvore é utilizado o método "findDistance", este por sua vez recorre ao "lowestCommonAncestor" que encontra o elemento que liga estas duas subárvores. Quando tivermos esta nova raiz que liga estas duas folhas calculamos a distância desta a cada uma das folhas e somamos, recorremos ao método "findLevel" para calcular esta distância.

Calculemos a complexidade algorítmica por detrás do método "getMaxDistance", calculando primeiramente todos os métodos auxiliares.

- **SecondTree:** Preenche uma árvore logo complexidade O(nlog(n)), uma vez que não a tem de ordenar.
- **GetAllLeafs:** Percorre uma árvore logo complexidade O(n).
- **LowestCommonAncestor:** Este método faz-nos uma simples travessia logo O(n).
- **FindLevel:** Método não determinístico, pior caso O(log(n)), melhor caso O (1).
- **GetTwoEletronicConfigurationMaxDistance:** Devido ao preenchimento do "TreeMap" este método fica com uma complexidade temporal de O (n^2), sobrepondo-se a todas as outras.

Logo o método "getMaxDistance", assume a complexidade temporal O (n^2).

Exercício 2.D:

Para elaboração desta alínea, serão necessários recorrer a vários métodos exteriores.

A árvore que seria necessário completar, era a árvore criada na alínea b, recorrendo a configurações únicas, pertencentes a uma árvore criada no início do exercício 2. É de reparar que, para os valores fornecidos pelo ficheiro do texto, não foi possível completar recorrendo a esses valores. É possível que na execução no exercício hajam falhas em condições que permitam a identificação possíveis, contudo foi perdido mais de 60% do tempo do trabalho na tentativa de executar esta alínea com sucesso.

```
public AVL<EletronicConfigurationAssembler> completeTheTree() {
    AVL<EletronicConfigurationAssembler> incompleteTree = createAVLDescendingOrder();
    int height = incompleteTree.height();

    createTree();

    completeTreeWithSingleEletronicConfigurations(incompleteTree, incompleteTree.root(), height);

    return incompleteTree;

}
```

Figura 9- Método principal para completar a árvore

O método instância uma árvore AVL, recorrendo ao método createAVLDescendingOrder, da alínea B, com uma complexidade temporal de **O(n^3*log(n)).** Recorre também ao método createTree():

```
public void createTree() {
    cleanTree();
    for (Element e : elementsList) {
        if (e.getElectronConfiguration() != "0") {
            String text = e.getElectronConfiguration();

            EletronicConfigurationAssembler eca = new EletronicConfigurationAssembler(text, 1);
            insert(eca);
        }
    }
}
```

Figura 10- Método create tree

O método createTree tem complexidade temporal de **O(n*log(n)),** por percorrer uma lista de elementos- complexidade de O(n)- e inserir valores uma árvore- O(log(n)).

Depois, recorre-se também ao método completeTreeWithSingleEletronicConfigurations. Este método está encarregue de inserir valores na árvore, enquanto ela não for completa. Ele recorrer também a métodos exteriores que serão explicados.

```java
public void completeTreeWithSingleEletronicConfigurations(AVL<EletronicConfigurationAssembler> incompleteTree,
        Node<EletronicConfigurationAssembler> node,
        int height) {
    if (node == null) {
        return;
    }

    if (node.getLeft() == null && node.getRight() == null && nodeDepth(node, incompleteTree) < height) {
        if (esquerda(incompleteTree, node.getElement()) != null) {
            incompleteTree.insert(esquerda(incompleteTree, node.getElement()));
        }
        if (direita(incompleteTree, node.getElement()) != null) {
            incompleteTree.insert(direita(incompleteTree, node.getElement()));
        } else {

        }
    }

    if (node.getLeft() == null && nodeDepth(node, incompleteTree) < height) {

        if (esquerda(incompleteTree, node.getElement()) != null) {
            incompleteTree.insert(esquerda(incompleteTree, node.getElement()));
        }
    }
    if (node.getRight() == null && nodeDepth(node, incompleteTree) < height) {

        if (direita(incompleteTree, node.getElement()) != null) {
            incompleteTree.insert(direita(incompleteTree, node.getElement()));
        }

    }
    if (node.getLeft() != null) {
        completeTreeWithSingleEletronicConfigurations(incompleteTree, node.getLeft(), height);
    }

    completeTreeWithSingleEletronicConfigurations(incompleteTree, node.getRight(), height);

}
```

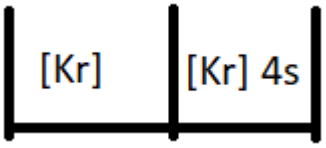
Figura 11- Método completeTreeWithSinglesEletronicConfigurations

O método direita e esquerda estão encarregues de indicar um valor possível a adicionar a um nó, dependendo do sítio onde ele precisa de ser inserido. Se um nó não tiver um ramo para a esquerda, e precise de ter para poder tornar a árvore completa, este tentará encontrar um possível nó recorrendo ao método esquerda, o mesmo acontecerá para o lado direito, recorrendo ao método direita. Ambos terão a mesma complexidade, por isso só será demonstrado o método direita:

```java
private EletronicConfigurationAssembler direita(AVL<EletronicConfigurationAssembler> incompleteTree,
        EletronicConfigurationAssembler ele) {
    List<EletronicConfigurationAssembler> list = list(incompleteTree);

    if (incompleteTree.smallestElement().equals(ele)) {
        List<EletronicConfigurationAssembler> l = sendPossibleValues(this, incompleteTree);
        for (EletronicConfigurationAssembler k : l) {
            if (k.getEletronicConfiguration().compareTo(ele.getEletronicConfiguration()) > 0 && !list.contains(k)) {
                return k;
            }
        }
    }
    if (biggestElement(incompleteTree).equals(ele)) {

        List<EletronicConfigurationAssembler> l = sendPossibleValues(this, incompleteTree);
        for (EletronicConfigurationAssembler k : l) {
            if (k.getEletronicConfiguration().compareTo(ele.getEletronicConfiguration()) > 0 && !list.contains(k)) {
                return k;
            }
        }
    }

    for (int i = 0; i < list.size() - 1; i++) {
        if (ele.getEletronicConfiguration().equals(list.get(i).getEletronicConfiguration())) {
            List<EletronicConfigurationAssembler> l = sendPossibleValues(this, incompleteTree);
            for (EletronicConfigurationAssembler k : l) {

                if (k.getEletronicConfiguration().compareTo(ele.getEletronicConfiguration()) > 0) {
                    if (k.getEletronicConfiguration().compareTo(list.get(i + 1).getEletronicConfiguration()) < 0 && !list.contains(k)) {
                        return k;
                    }
                }

            }

        }
    }

    return null;
}
```

Figura 12- Método direita que encontra valores à direita de uma configuração

O método direita, precisará de encontrar elementos, recorrendo à lista in-order da árvore a ser completa. O endereço eletrónico do nó em que é pretendido adicionar um endereço terá que ser menor que o endereço adicionado. Para isso, será explicada a ordem de pensamento do exercício na seguinte ilustração:



Imaginando que [Kr] precisa de um elemento à direita, e sabendo que a travessia in-order devolve os elementos da árvore por ordem crescente, o valor a ser adicionado terá que ser maior que [Kr] e menor que [Kr] 4s. No caso de ser pretendido adicionar um nó à esquerda, o elemento anterior ao nó que precisa de um ramo, na travessia in-order, terá que ser maior que o elemento a adicionar, e o elemento a adicionar terá que ser menor que o nó onde se pretende adicionar. No método direita e esquerda, recorre-se ao método smallest element, e consequentemente o biggest element (criado na classe), que têm uma complexidade de **O(n)** e a um método chamado sendPossibleValues que têm complexidade temporal de O(n) também (percorre a árvore completa). Ao todo, a complexidade temporal do método é de **O(n^3).**

Ao todo, o método principal terá uma complexidade de O(n^3), sendo que esta é a maior das complexidades de todos os métodos envolves na realização da alínea.