Estruturas de Informação

Relatório

3º Trabalho Prático

Turma 2DL
Ricardo Mesquita 1190995
Gonçalo Jordão 1190633

isep Instituto Superior de Engenharia do Porto

Requisitos

1. Armazenamento de informação do ficheiro

Neste projeto, o método utilizado para a leitura de informação do ficheiro é denominado *readElements.*

```
void readElements() {
    try {
        BufferedReader reader = new BufferedReader(new java.io.FileReader(Constants.FILE_PATH));
        reader.readLine();
        String line;
        while ((line = reader.readLine()) != null) {
            String[] data = line.split(Constants.FILE_SPLIT);
            Element fullElement = this.elementsList.newElement(data[0].trim(), data[1].trim(), data[2].trim(), data[3].trim(), data[4].trim(), data[5].trim(), data[6].trim(),
                    data[7].trim(), data[8].trim(), data[9].trim(), data[10].trim(), data[11].trim(), data[12].trim(), data[13].trim(), data[14].trim(), data[15].trim(),
                    data[16].trim(), data[17].trim(), data[18].trim(), data[19].trim(), data[20].trim(), data[21].trim(), data[22].trim(), data[23].trim());
            if (!this.elementsList.exitsElement(fullElement)) {
                this.elementsList.registerElement(fullElement);
                this.elementsAtomicNumberTree.insert(fullElement.getAtomicNumber());
                this.elementsNameTree.insert(fullElement.getElement());
                this.elementsSymbolTree.insert(fullElement.getSymbol());
                this.elementsAtomicMassTree.insert(fullElement.getAtomicMass());
                this.elementsConfigurationTree.insert(fullElement.getElectronConfiguration());
            }
        }
```

Neste método, foi utilizada a classe *Element* para que fosse possível criar o objeto completo. Através deste objeto e dos seus respetivos métodos *gets,* foram preenchidas as árvores necessárias para o projeto em questão.

Foram criadas árvores para:

- representação de *atomic numbers:*

```
/**
 * The Elements Atomic Number Tree.
 */
public AVL<Integer> elementsAtomicNumberTree;
```

- representação de *elements name*:

```
/**
 * The Elements Name Tree.
 */
public AVL<String> elementsNameTree;
```

- representação de *elements symbol*:

```
/**
 * The Elements Symbol Tree.
 */
public AVL<String> elementsSymbolTree;
```

- representação de *atomic mass:*

```
/**
 * The Elements Atomic Mass Tree.
 */
public AVL<Double> elementsAtomicMassTree;
```

representação de eletronic configuration:

```
/**
 * The Elements Configuration Tree.
 */
public AVL<String> elementsConfigurationTree;
```

2. Busca de informação

Nos métodos presentes, a busca dos elementos é feita através da árvore em conjunto com algoritmos criados especificamente para o desenvolver deste mesmo exercício.

Nesse algoritmo, é retornada uma lista com a informação que está entre os valores máximos e mínimos inseridos pelo utilizador (remetente para a 1ª questão).

getElementByAtomicNumbers*:*

```
public List<Element> getElementByAtomicNumber(int minimumAtomicNumber, int maximumAtomicNumber) {
    List<Element> listWithElements = new ArrayList<>();

    if (minimumAtomicNumber == maximumAtomicNumber) {
        Element e = this.elementsList.getElementByAtomicNumber(minimumAtomicNumber);
        if (e != null) {
            listWithElements.add(e);
        }
    } else {
        for (Integer atomicNumber : this.elementsAtomicNumberTree.find(this.elementsAtomicNumberTree.root, minimumAtomicNumber, maximumAtomicNumber)) {
            listWithElements.add(this.elementsList.getElementByAtomicNumber(atomicNumber));
        }
    }

    if (!listWithElements.isEmpty()) {
        listWithElements.sort(Element::compareOrder);
    }

    return listWithElements;
}
```

getElementByElement:

```
public List<Element> getElementByElement(String minimumElement, String maximumElement) {
    List<Element> listWithElements = new ArrayList<>();

    if (minimumElement.equalsIgnoreCase(maximumElement)) {
        Element e = this.elementsList.getElementByElement(minimumElement);
        if (e != null) {
            listWithElements.add(e);
        }
    } else {
        for (String element : this.elementsNameTree.find(this.elementsNameTree.root, minimumElement, maximumElement)) {
            listWithElements.add(this.elementsList.getElementByElement(element));
        }

        if (!listWithElements.isEmpty()) {
            listWithElements.sort(Element::compareOrder);
        }
    }

    return listWithElements;
}
```

getElementBySymbol:

```java
public List<Element> getElementBySymbol(String minimumSymbol, String maximumSymbol) {
    List<Element> listWithElements = new ArrayList<>();

    if (minimumSymbol.equalsIgnoreCase(maximumSymbol)) {
        Element e = this.elementsList.getElementBySymbol(minimumSymbol);
        if (e != null) {
            listWithElements.add(e);
        }
    } else {
        for (String symbol : this.elementsSymbolTree.find(this.elementsSymbolTree.root, minimumSymbol, maximumSymbol)) {
            listWithElements.add(this.elementsList.getElementBySymbol(symbol));
        }
    }

    if (!listWithElements.isEmpty()) {
        listWithElements.sort(Element::compareOrder);
    }

    return listWithElements;
}
```

getElementByAtomicMass:

```java
List<Element> getElementByAtomicMass(double minimumAtomicMass, double maximumAtomicMass) {
    List<Element> listWithElements = new ArrayList<>();

    if (minimumAtomicMass == maximumAtomicMass) {
        Element e = this.elementsList.getElementByAtomicMass(minimumAtomicMass);
        if (e != null) {
            listWithElements.add(e);
        }
    } else {
        for (Double atomicMass : this.elementsAtomicMassTree.find(this.elementsAtomicMassTree.root, minimumAtomicMass, maximumAtomicMass)) {
            listWithElements.add(this.elementsList.getElementByAtomicMass(atomicMass));
        }
    }

    if (!listWithElements.isEmpty()) {
        listWithElements.sort(Element::compareOrder);
    }

    return listWithElements;
}
```

3. Retorno de configurações eletrónicas por ordem decrescente de ocorrências

Neste caso foi criado um algoritmo no âmbito de pesquisar e obter o número de ocorrências repetidas no que toca às configurações eletrónicas, retornando-as num *TreeMap*.
Seguidamente, foi feita a filtragem das configurações que possuíam mais que 1 ocorrência.

```java
Map<String, Integer> getConfiguration() {
    Map<String, Integer> mapWithNecessaryInformation = new LinkedHashMap<>();

    TreeMap<String, Integer> mapAux = this.elementsConfigurationTree.findOccurrences(this.elementsConfigurationTree.root);

    for (String s : mapAux.keySet()) {
        if (mapAux.get(s) > 1) {
            mapWithNecessaryInformation.put(s, mapAux.get(s));
        }
    }

    return mapWithNecessaryInformation.entrySet().stream().sorted(Map.Entry.<String, Integer>comparingByValue().reversed()).collect(Collectors.toMap(Map.Entry::getKey,
            Map.Entry::getValue, (key, content) -> content, LinkedHashMap::new));
}
```

4. **Construção de uma árvore em formato AVL (incluída na BST), com a inserção das configurações eletrónicas realizadas por ordem decrescente do número de ocorrências (acima de 2 ocorrências).**

   Este método, recorre a um outro método criado anteriormente, para obter as configurações eletrónicas e as suas respetivas ocorrências (método *getConfiguration*). Para a lista retornada do método acima referido, é realizada uma filtragem para que se sejam apenas obtidas as configurações que apresentem um número de ocorrências superiores a 2.
   Através destes dados, são criados objetos *EletronicConfiguration*, onde mais tarde serão inseridos numa árvore.

```
void createConfigurationsOccurrencesTree() {

    if (this.elementsConfigurationOccurrencesTree.size() == 0) {

        Map<String, Integer> mapAux = getConfiguration();

        for (String s : mapAux.keySet()) {
            if (mapAux.get(s) > 2) {
                ElectronicConfiguration c = new ElectronicConfiguration(s, mapAux.get(s));
                this.elementsConfigurationOccurrencesTree.insert(c);
            }
        }
    }
}
```

5. **Verificar a distância entre os dois nodes mais afastados da árvore.**

   Neste método, é utilizado um algoritmo implementado especificamente para este exercício para que fosse possível determinar a distância entre as configurações mais distantes(*findMaximumDistance*).
   Assim, é criada uma árvore de ocorrências e seguidamente este método é aplicado. Este, verifica a distância entre pares de *nodes* e retorna um *map* com a maior distância obtida entre os mesmos.

```
Map<List<ElectronicConfiguration>, Integer> distanceBetweenTheTwoFurthestConfigurations() {

    createConfigurationsOccurrencesTree();

    return this.elementsConfigurationOccurrencesTree.findMaximumDistance();
}
```

6. **Transformação da árvore obtida anteriormente numa árvore binária completa, inserindo nesta, configurações eletrónicas únicas.**

   Neste método, foi utilizado um algoritmo capaz de verificar se uma árvore está completa (*isComplete*).
   No caso deste retornar *false* é feita a inserção de configurações eletrónicas únicas numa árvore, até que esta, se possível, fique completa.
   Caso já tenham sido inseridas todas as configurações eletrónicas únicas, e a árvore não tenha ficado completa, o método em questão (*createBinaryCompleteTree*) retorna o valor booleano *false*.

```java
boolean createBinaryCompleteTree() {
    createConfigurationsOccurrencesTree();

    boolean isComplete = this.elementsConfigurationOccurrencesTree.isComplete();

    if (!isComplete) {
        List<String> uniqueElectronicConfigurations = new ArrayList<>();

        TreeMap<String, Integer> mapAux = this.elementsConfigurationTree.findOccurrences(this.elementsConfigurationTree.root);

        for (String s : mapAux.keySet()) {
            if (mapAux.get(s) == 1) {
                uniqueElectronicConfigurations.add(s);
            }
        }

        int count = 0;

        while (count < uniqueElectronicConfigurations.size()) {
            ElectronicConfiguration c = new ElectronicConfiguration(uniqueElectronicConfigurations.get(count),  occurrences: 1);
            this.elementsConfigurationOccurrencesTree = new AVL<>();
            createConfigurationsOccurrencesTree();
            this.elementsConfigurationOccurrencesTree.insert(c);
            count++;
            isComplete = this.elementsConfigurationOccurrencesTree.isComplete();
            if (isComplete) {
                break;
            }
        }
    }
```

7. Outras Classes Utilizadas

Classe Constants:

Esta classe foi utilizada meramente para guardar variáveis que foram consideradas constantes ao longo de todo o projeto.

Assim, caso fosse necessário alterar o valor de alguma destas variáveis, bastava apenas aceder a esta classe.

```java
public class Constants {

    /**
     * The constant FILE_PATH.
     */
    public static final String FILE_PATH = "Files\\PeriodicTableOfElements.csv";
    /**
     * The constant FILE_SPLIT.
     */
    public static final String FILE_SPLIT = ",";
}
```

8. Análise de complexidade

Métodos a analisar	***Complexidade***
readElements	O(n)
getElementByAtomicNumber	*Algoritmo não-determinístico* Melhor caso: O (1) Pior caso: $O(\log_n)$
getElementByElement	*Algoritmo não-determinístico* Melhor caso: O (1) Pior caso: $O(\log_n)$

getElementBySymbol	*Algoritmo não-determinístico* Melhor caso: O (1) Pior caso: $O(\log_n)$
getElementByAtomicMass	*Algoritmo não-determinístico* Melhor caso: O (1) Pior caso: $O(\log_n)$
getConfiguration	$O(n^2)$
createConfigurationsOccurrencesTree	O (n)
distanceBetweenTheTwoFurthestConfigurations	*Algoritmo não-determinístico* Melhor caso: O (1) Pior caso: $O(n^2)$
createBinaryCompleteTree	*Algoritmo não-determinístico* Melhor caso: O (n) Pior caso: $O(n^2)$

9. Conclusões e outras informações relevantes

Foi um trabalho um pouco confuso no que toca à compreensão do enunciado. Desta maneira, não foi possível ter a certeza que os resultados obtidos estejam de acordo com o que é pedido.
Para que a realização de alguns exercícios fosse possível, foram criados algoritmos nas classes relativas à construção e acesso de árvores BST (AVL incluído) disponibilizadas pelos docentes.