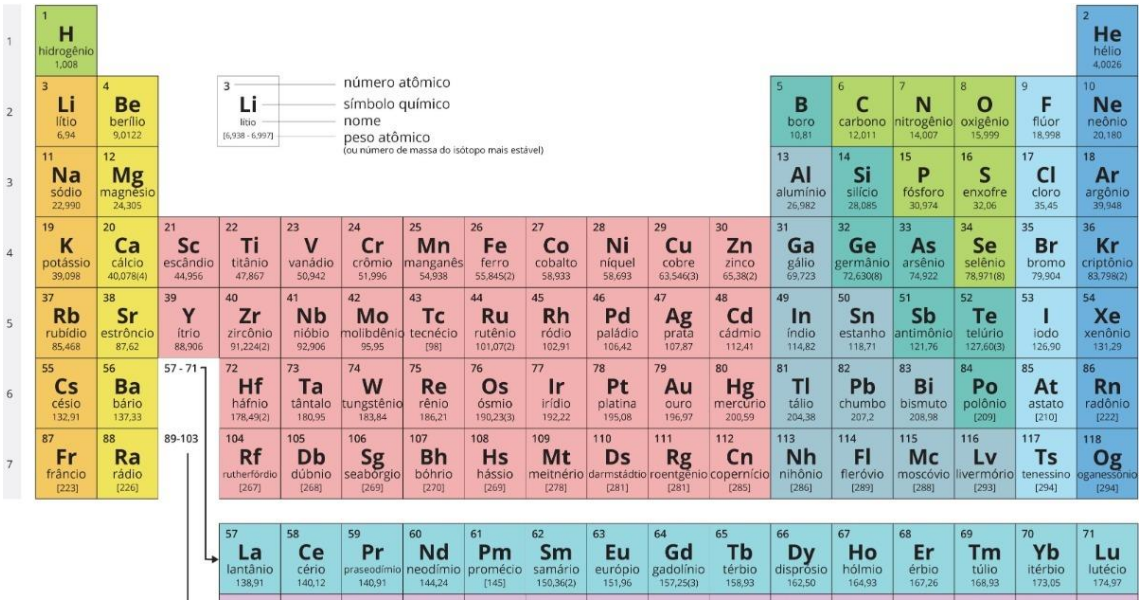
Instituto Superior de Engenharia do Porto

Licenciatura em Engenharia Informática
ESINF – 2020/2021



Relatório Estruturas de Informação

Projeto 3 – Tabela Periódica

Autores:
[1191097] [Tiago Machado]
[1191111] [Tomas Flores]

Turma: 2DL

Data: [22/12/20]

Introdução

Foi-nos proposto, para o segundo projeto da disciplina de Estruturas de Informação, o desenvolvimento de uma biblioteca de classes, respetivos métodos e testes que permitam gerir a informação relativa à tabela periódica. Para tal, aplicamos os conhecimentos adquiridos nesta disciplina sobre a Java Collections Framework e Trees para dar uma resposta mais eficiente aos requisitos do projeto.

Primeiramente, armazenamos a informação recolhida no ficheiro "*Periodic Table of Elements.csv"* em 5 arvores diferentes, 4 delas *avls*, para a procura de um elemento pelo seu numero atómico, elemento, símbolo e massa atómica, e uma *bst*, inserimos as varias configurações eletrónicas que existem.

Complexidade –
readFile – O(nlog(n))

1. Desenvolver a/(s) classe/(s) que permita obter toda a informação relativa a cada um dos elementos da tabela periódica:

a. Pesquisa de elementos por qualquer um dos seguintes campos: Atomic Number, Element, Symbol ou Atomic Mass.

Para este primeiro requisito, foi necessário recorrer á herança e generalização, onde para cada método de pesquisa criamos uma subclasse da superclasse *Element,* que usando o método *compareTo(Element o)* conseguimos definir o método de comparação entre dois elementos. Depois, para cada uma das *avls* existentes inserimos uma subclasse de Element que irá definir como serão feitas as comparações entre elementos da arvore. No caso da *avlAtomicNumber* são adicionados objetos da subclasse *Element_Comparable_AtomicNumber*, para a *avlElement* elementos da subclasse *Element_Comparable_Element* e assim sucessivamente.

Para achar o elemento por qualquer uma das avls utilizamos o metodo *find,* que, começando sempre na *root* vai *node* a *node* comparar o seu valor, se este for maior que o *node*, vamos buscar a *right child* e repetimos o processo, se for menor vamos buscar a *left child*, se for igual retorna o *node* e se percorrer todos os *nodes* da arvore e não encontrar retorna *null*.

Complexidade – Worst Case: *O(log(n))*

b. Pesquisa por intervalo de valores de Atomic Mass através de dois valores (mínimo e máximo) passados por parâmetro, devolver o conjunto de elementos com Atomic Mass nesse intervalo ordenado por Discoverer e Year of Discovery (cresc./decres.) juntamente com um sumário do número de elementos devolvidos agrupados por Type e Phase.

Para este requisito, recorremos primeiramente ao método *getElementsBetween(atomicMassMin, atomicMassMax, map)* recebemos todos os elementos com massa atómica dentro de um intrevalo de valores. Neste método, pesquisamos de forma iterativa todos os elementos da *avlAtomicMass,* até que tanto a direita como a esquerda de um node, contenha um elemento com uma massa atómica fora do intrevalo de valores definido pelo utilizador. Enquanto isto acontece, vão sendo

adicionando somas a um map que é recebido por parâmetro e que será utilizado para a segunda tabela.

Feita a busca dos elementos, no método *sortElementsBetween,* recebemos a lista de elementos e ordenamos a lista por Discoverer e Year of Discovery (cresc./decres.).

Mais tarde, esta lista é recebida no método *printUmB* imprime a lista ordenada na consola, e com o mapa anteriormente passado por parâmetro imprime também um sumário do número de elementos devolvidos agrupados por Type e Phase.

Complexidade – Worst Case: *O(nlog(n))*

2. Observando a configuração electrónica dos elementos (coluna Electron Configuration), verifica-se a existência de repetição de padrões.

a. Recorrendo apenas à estrutura árvore binária de pesquisa (BST), devolva por ordem decrescente as configurações eletrónicas com mais do que uma repetição, agrupadas por número de repetições.

Para este requisito, como já referido anteriormente, utilizamos uma *bst*. Isto porque a inserção é mais eficiente que uma *avl* e, visto que irá ser preciso percorrer a arvore por completo, não precisamos de uma grande eficiência de pesquisa. Criamos então uma *bst* de Strings contendo as varias configurações eletrónicas que não sejam null.

Para achar as repetições, criamos o método *getElectronConfigurationCount* onde percorremos uma lista de todos os elementos da *bst* por ordem crescente. Depois, para cada configuração eletrónica dividimo-la em partes vamos contando as vezes que cada parte se repete. Por exemplo, na configuração [He] 2s2 2p1, vamos dividir esta por varias partes: “[He]”, “[He] 2s2” e “[He] 2s2 2p1”. De seguida, vamos parte a parte e adicionamos uma repetição a cada parte. No fim iremos obter o numero de repetições que cada parte tem.

Para isto recorremos a um TreeMap<Integer,LinkedList<String>>, onde usamos o número de repetições como chave e um LinkedList com as configurações que tem o mesmo número de repetições que a chave. Para saber qual o numero de repetições de cada configuração usamos um TreeMap auxiliar TreeMap<String, Integer>, onde na chave temos a String da configuração e o *value* é o numero de repetições que esta configuração tem. Para efeitos futuros, vamos recolhendo também as configurações que aparecem apenas uma vez.

Por fim, para imprimir as configurações na consola, utilizamos o método *printDoisA().* Neste, recebemos a lista do metodo getElectronConfigurationCount(), que, sendo um TreeMap, tem as chaves organizadas em ordem crescente. Assim, recorremos ao *map.descendingKeySet()* que irá imprimir as repetições por ordem decrescente.

Complexidade – Worst Case: *O(n)*

b. Construa uma nova BST inserindo por ordem decrescente as configurações eletrónicas com repetição acima de 2 obtidas na alínea anterior.

Para este requisito começamos com o método *avlDecr()*, que vai invocar o método *getElectronConfiguration()*, que vai retornar um mapa em que a *key* é o número de vezes que o elemento esta repetido, e o *value* é uma *list* que contem todas as configurações eletrónicas que repetem tantas vezes como o valor da chave.
Depois, para cada *key* do mapa, vão ser adicionados todos os valores da lista contida no *value* associado a uma avl, com à exceção ao *value* quando a *key* é 1.

Complexidade – Worst Case: *n*log(n)*

c. Desenvolva um método que devolva os valores das duas configurações electrónicas mais distantes na árvore e a respectiva distância.

Para este requisito temos o método *findLongestPat(),* que vai invocar o método *longestPath()* para o *node* esquerdo da root, e para o *node* direito da root. O método *longestPath()* vai recursivamente encontrar o caminho mais longo a partir do node fornecido, retornando um *ArrayList* de *Strings*, que vai conter o percurso e em que o primeiro item da *list* vai ser o ultimo *node*. Assim, ao invocarmos o método *longestPath() para* o node esquerdo e direito da root, vamos obter o percurso mais longo para cada um deles. Depois, extraímos o primeiro valor do *Array* de cada um (que contem o percurso), obtendo assim o primeiro e último elemento do percurso, e depois calculamos o tamanho dos dois arrays de modo a obter o tamanho do percurso de cada um, e junta mos ambos os valores obtidos, e juntamos, obtendo assim a distância percorrida. Isto pode ser feito pois estando a usar uma avl, o caminho mais longo vai ter de passar sempre pela *root.*

Complexidade – Worst Case: *log(n)*

d. Desenvolva um método que transforme a árvore obtida alínea anterior numa árvore binária
completa, inserindo nestas possíveis configurações electrónicas únicas.

Para este requesito temos o método *completeBinaryTree,* que vai começar por criar duas listas, a lista *listUnique* e a lista *listElementosRepetidos*, que vão ambas ser preenchidas através do método *getElementList()*, que por sua vez invoca o método *getElectronConfiguration(),* e utiliza o mapa retornado por este para preencher as respetivas listas com todas as configurações eletrónicas não repetidas, para a *listUnique,* e as repetidas para a *listElementosRepetidos*. Depois, vamos obter um mapa criar um mapa *elementsPerLevel* através do método *nodesByLevel*, que vai conter o número de elementos por cada nível da avl (que foi do método 2b). De seguida, para o penúltimo nível, é adicionado um *node* sempre que o os *nodes* do nível superior, não tem elemento esquerdo ou direito.
Isto é repetido para o ultimo nível, mas desta vez apenas são adicionados *nodes* de modo a manter o arvore completamente cheia mas preenchida o mais à esquerda possível.

Complexidade – Worst Case: n^2