Realizado por:

Fábio Fernandes 1191430

Bárbara Pinto 1191507

Síntese

Ao longo do documento iremos demonstrar o modo como gerimos a informação que nos é fornecida através de um ficheiro .csv, bem como a explicação dos nossos métodos para responder aos diferentes requerimentos que a aplicação necessitaria ter, tendo em conta a eficiência, mostrando também a análise de complexidade de todas as funcionalidades implementadas.

Relatório de estruturas de informação

Projeto 3 – Tabela Periódica

**Introdução**

Na unidade curricular de Estruturas de Informação foi-nos proposto, para terceiro projeto a criação de uma biblioteca de classes, que permita gerir a informação relativa à tabela periódica.

Organizamos a informação disponibilizada em ficheiro .csv, contendo separadamente todos os atributos descritivos de um elemento da tabela periódica, aplicando os conhecimentos adquiridos na *Java Collections Framework* e recorrendo a classes genéricas e herança para responder eficientemente aos requisitos que nos foram colocados ao longo deste projeto.

Neste projeto decidimos criar duas classes para os métodos genéricos pertencentes a cada tipo de árvore utilizada, sendo elas, *BinaryTree* contendo os métodos genéricos de uma *Binary Search Tree (BST)* e a classe *BalancedTree* sendo uma extensão da *BinaryTree* contendo os métodos genéricos de uma *Adelson-Velsky and Landis* (*AVL)*.

Optamos por substituir a interface *Comparable* na classe *BinaryTree* (inicialmente fornecido pelos professores) por um *Comparator* que é utilizado como atributo da classe e é inicializado pelo construtor. A razão desta escolha prendeu-se com a possibilidade de construir facilmente várias árvores que, recebendo os mesmos tipos de dados (neste caso objetos da classe *ChemicalElement*) os ordenassem de maneiras distintas.

Em seguida criamos uma classe chamada *PeriodicTable*, que representa uma *BalancedTree* do tipo *Chemical Element*. Utilizamos esta classe para armazenar todo o tipo de métodos mais específicos às questões do projeto, métodos estes que alavancam as funcionalidades genéricas de cada um dos tipos de árvores, fazendo posteriormente os necessários ajustes.

Por fim, e como já referido, criamos uma classe *ChemicalElement*, que modela a informação vinda do ficheiro .csv e armazena todos os atributos de cada um dos elementos que serão utilizados na *PeriodicTable*.

## Requisito nº1 – Pesquisa de elementos por campos: Atomic Number, Element, Symbol ou Atomic Mass.

Para este primeiro procedimento decidimos armazenar a informação recolhida do ficheiro de texto que nos foi disponibilizado para o projeto numa árvore binária designada AVL. A escolha recaiu na maior eficiência de pesquisa que este tipo de árvore oferece – complexidade temporal *O(log n)*, tanto no melhor como pior caso – apesar do sacrifício de eficiência devido aos métodos *insert* e *remove* dotados de maior complexidade face à BST. O sacrifício, no entanto, será mínimo tendo em conta que na tabela periódica o número de elementos é relativamente reduzido e conhecido.

Neste procedimento decidimos utilizar a classe *Chemical Element* para guardar todo o tipo de atributos relativos ao elemento, inserimos os elementos criados na árvore binária *AVL* específica, isto porque, foram criadas quatro árvores contendo um C*omparator* que difere em atributo de comparação.

Iniciando a pesquisa, cria-se um elemento vazio apenas com o atributo que foi inserido pelo utilizador, de forma a que o *Comparator* disponível nessa árvore, consiga comparar *Chemical Elements* e só retorne o elemento assim que encontrar um *Chemical Element* que tenha esse atributo específico.

## Requisito nº2 – Pesquisa por intervalo de valores de Atomic Mass através de dois valores (mínimo e máximo) passados por parâmetro, devolvendo o conjunto de elementos de Atomic Mass nesse intervalo ordenados por Discoverer e Year of Discovery, juntamente com um sumário do número de elementos devolvidos agrupados por Type e Phase.

Decidimos separar o processamento deste requisito em 3 partes:

1. Pesquisar os elementos com Atomic Mass compreendida entre o valor mínimo e o valor máximo.  
   Para isso, utilizamos um método recursivo que verifica todas as subtrees dos Nodes compreendidos entre os valores limites, adicionando a uma lista passada por parâmetro.
2. Ordenar a lista, modificada pelo método acima, pelos atributos *Year of Discovery* e *Discoverer*, utilizando mais uma vez a classe C*omparator* que verifica, para cada *Chemical Element*, o atributo *Discoverer*, ordenando-o por ordem alfabética crescente. No caso de “empate”, o *Comparator* procede ainda à verificação do atributo Year *of Discovery*, ordenando os registos do mais recente para o mais antigo.
3. Tendo a lista organizada recorremos à contagem dos elementos devolvidos agrupados por *Type* e *Phase*. Esta ordenação, embora não seja um requisito, é feita de modo a simplificar e tornar mais eficiente o processo de iteração pela lista e extração dos valores. Decidimos utilizar uma matriz – contendo uma coluna para cada um dos atributos *Phase* e uma linha para cada *Pair* de valores, em que a *Key* representa o *Type* e o *Value* armazena um vetor de somas contabilizadas do tipo de elemento para cada estado físico.

## Requisito nº3 – Recorrendo apenas à estrutura árvore binária de pesquisa (BST), devolver por ordem decrescente as configurações eletrónicas com mais que uma repetição, agrupadas por número de repetições.

Neste requisito optamos por recorrer à estrutura de dados *Map*, organizada com o nome da configuração eletrónica, como *Key* e o número de ocorrências como *Value*.

Utilizando o método *getPatterns* verificam-se todos os elementos que estão presentes na tabela periódica e todas as subconfigurações presentes (por exemplo, para uma configuração [Ar] 3d10 4s1, as subconfigurações consideradas são [Ar], [Ar] 3d10 e [Ar] 3d10 4s1). Estas subconfigurações são, por sua vez, armazenadas num *Map* com a estrutura descrita acima.

Após a primeira iteração por todos os elementos da tabela periódica para recolher todas as possíveis configurações, efetua-se uma segunda iteração, desta vez comparando a configuração de cada elemento à lista de configurações e registando as repetições encontradas através do método *put*.

De referir que estas iterações são feitas com recurso ao método *inOrder()* da árvore binária. Adicionalmente, a complexidade temporal deste processo de pesquisa de configurações eletrónicas acaba por totalizar *O(2n)*, devido às duas iterações separadas.

## Requisito nº4 – Construir uma nova BST inserindo por ordem decrescente as configurações eletrónicas com repetição acima de 2 obtidas no requisito anterior.

Recorrendo ao método descrito anteriormente já conseguimos responder corretamente a este requisito, necessitando apenas de restringir o número de repetições, isto é, verificar quais as configurações eletrónicas em que o número de repetições for superior a 2. Tal é feito através da passagem do mínimo de repetições desejado por parâmetro para o método *getPatterns()*.

De referir que a árvore binária gerada é também uma *AVL*, devido às mesmas razões descritas anteriormente.

## Requisito nº5 – Devolver um método que devolva os valores das duas configurações eletrónicas mais distantes na árvore e a respetiva distância.

Neste requisito, utilizamos o conceito de diâmetro de um grafo no método *subTreeHeight*, que calcula para cada Node a profundidade das *subtrees* tanto à esquerda como à direita, sendo que para obter a maior distância entre dois nodes se obtém o Node com a máxima profundidade. Já os Nodes mais distantes obtêm-se também através da profundidade das suas *subtrees*. Neste caso, os Nodes em que este valor é menor serão aqueles nos níveis mais baixos da árvore e, por consequência, os mais afastados. Assim, o método retorna dois Nodes (numa lista passada por parâmetro) que constituem os que estão nos níveis mais inferiores e têm menor altura da respetiva *subtree*.

## Requisito nº 6 – Devolva um método que transforme a árvore obtida alínea anterior numa árvore binária completa, inserindo nestas possíveis configurações eletrónicas únicas.

Considera-se uma árvore completa como uma árvore completamente preenchida, em todos os níveis exceto o último, e com todos os Nodes o mais à esquerda possível.

Uma vez que a árvore obtida no requisito 4 não cumpria estas regras, uma vez que possuía um “buraco” no penúltimo nível e apenas um Node não alinhado à esquerda no último nível, tornava-se necessário fazer um ajuste para se tornar uma árvore completa.

A primeira solução desenhada envolvia verificar, em primeiro lugar, calcular o número de Nodes em falta num dado nível da árvore, comparando o resultado do método *nodesByLevel*() com o número desejado de Nodes (2n  onde n corresponde ao nível). Caso o nível estivesse incompleto, era feita uma verificação dos “filhos” de cada Node (os Nodes à direita e esquerda), caso qualquer um estivesse em falta seria inserido o menor/maior (dependendo da falha ser à esquerda ou à direita) valor mais próximo contido na lista de configurações únicas passadas por parâmetro.

No entanto, devido às rotações da *AVL* e à ausência de alguns valores próximos o suficiente de cada Node para uma inserção bem sucedida, não foi possível implementar esta lógica da maneira pretendida.

A solução final a que chegamos consistiu em reinserir todos os elementos da árvore de configurações original por ordem alfabética mas com o preenchimento forçado dos Nodes na árvore da esquerda para a direita, obtendo dessa maneira uma árvore completa sem a necessidade de inserção de novos elementos.

**Complexidades (Resumo)**

Apresentamos abaixo a complexidade temporal estimada de todos os métodos implementados nas classes (exceto *getters*, *setters*, construtores e métodos de *override*).

***BinaryTree:***

Algoritmos:

1. insert: *O(n)*
2. remove: *O(n)*
3. size: *O(log(n))*
4. height: *O(log(n))*
5. smallestElement: *O(n)*
6. find: *O(n)*
7. findInterval: *O(n)*
8. inOrder: *O(n)*
9. inOrderSubtree: *O(n)*
10. preOrder: *O(n)*
11. preOrderSubtree: *O(n)*
12. posOrderSubtree: *O(n)*
13. processBstByLevel: *O(n)*
14. subTreeHeight: *O(n)*
15. maxDistance: *O(n)*

***BalancedTree:***

Algoritmos:

1. balanceFactor: *O(n)*
2. rightRotation: *O(n)*
3. leftRotation: *O(n)*
4. twoRotations: *O(n)*
5. balanceNode: *O(n)*
6. insert: *O(log(n))*
7. remove: *O(log(n))*
8. equals: *O(1)*

***Classe ChemicalElement:***

Algoritmos:

Todos os algoritmos que estão presentes na classe ChemicalElement tem complexidade O(1).

***Classe PeriodicTable:***

Algoritmos:

1. find: *O(n)*
2. searchByInterval: *O(n)*
3. orderByDiscovererAndYear: *O(n)*
4. groupByTypeAndPhase: *O(n2)*
5. getPatterns: *O(n2)*
6. generateElectronConfigTree: *O(n)*

Conclusão

Concluímos que com este trabalho ficamos a compreender as diversas diferenças entre uma BST e uma AVL, conseguindo alcançar o objetivo deste projeto.

Com base na revisão do trabalho anterior, o nosso objetivo para este trabalho era responder aos pontos mencionados com os algoritmos mais eficientes, para não ter um método não funcional se for necessário inserir uma grande quantidade de informação.

Sem falta, mencionar que a utilização da *Java Collections Framework* se mostrou bastante útil para esta etapa, assim como a explicação dos professores aos diferentes requisitos perante o enunciado fornecido.