RELATÓRIO DE ESTRUTURAS DE INFORMAÇÃO – SPRINT 1

André Ferreira 1200605 2DO

Diogo Silva 1180611 2DO

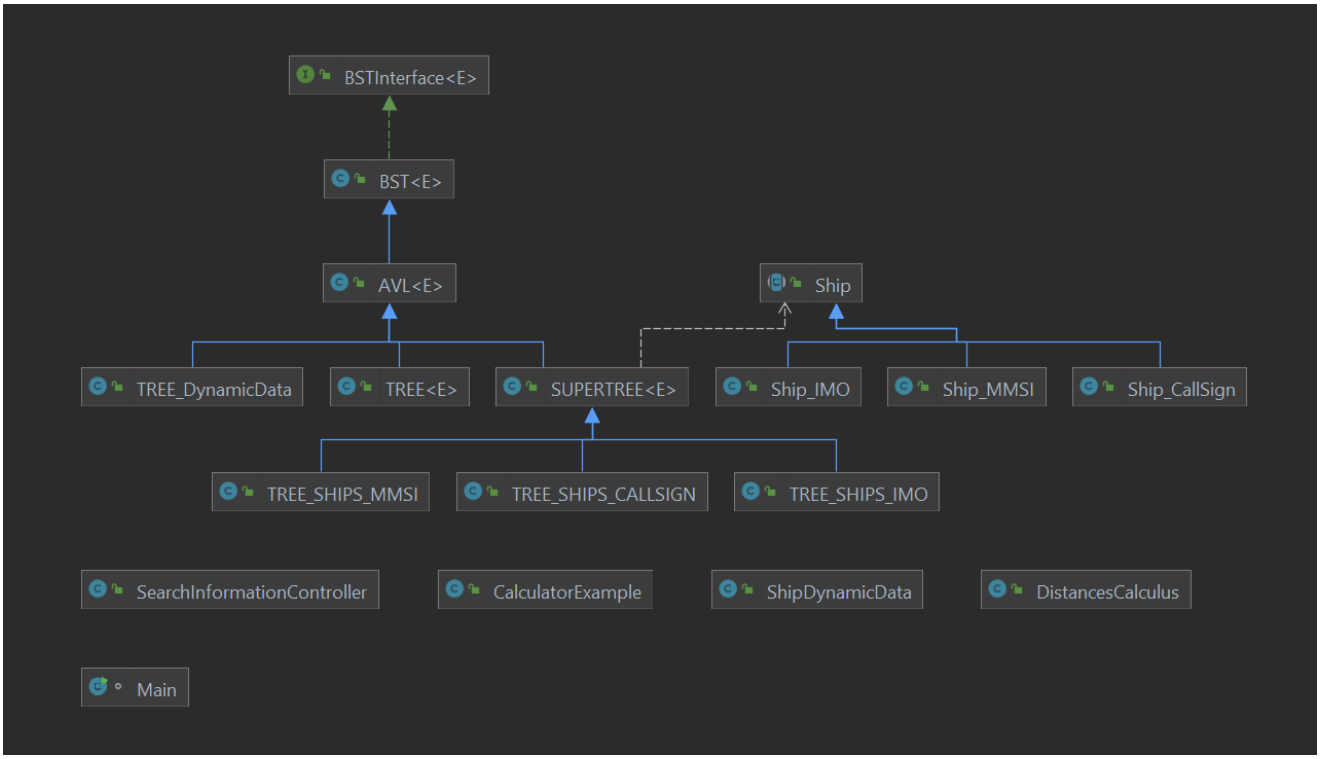
Francisco Ferreira 1180682 2DO

João Ferreira 1190708 2DO

Sérgio Lopes 1200625 2DO

Índice

Diagrama de Classes



Exercício 1

No exercício 1 é necessário guardar os dados estáticos e dinâmicos dos navios dos dados fornecidos nos ficheiros csv (sships ou bships). De forma a conseguir satisfazer o exercício, optamos por criar uma árvore para guardar os dados dinâmicos dos movimentos que os barcos iam fazendo, *TREE_DynamicData,* tais como data / hora da mensagem AIS, a latitude, longitude, dia e hora do movimento do navio, SOG (velocidade sobre o solo), COG (curso sobre o solo, direção relativa ao Norte absoluto (em graus: [0; 359]), rumo do navio em graus, código do navio em reboque e a classe to transciever utilizado no envio dos dados.

Além da *TREE_DynamicData* foi criada também a classe *SUPERTREE*, que armazena os dados estáticos do navio, tais como, MMSI, *VesselName, IMO, CallSign, VesselType, Length, Width, Draft,* sendo esta classe uma superclasse que se encontra herdada por 3 subclasses, *TREE_SHIPS_CALLSIGN, TREE_SHIPS_IMO* e *TREE_SHIPS_MMSI.* Esta herança foi desenvolvida de modo a evitar duplicação de código, visto que estas árvores teriam todos os atributos iguais, menos o código pelo qual se deseja procurar um navio em específico, sendo necessário em cada uma destas subclasses um método de comparação.

Na superclasse *SUPERTREE* foram criados 3 métodos de forma a criar subárvores para os 3 diferentes tipos de códigos para a procura dos movimentos do navio, que são os métodos *createCallSignTree()*, *createIMOTree(), createMMSITree(),* tendo para a criação da árvore o método *createTree().*

Análise de Complexidade

Relativamente à análise de complexidade, temos de analisar a complexidade nos métodos *createMMSITree(), createIMOTree() e createCallSignTree()* sendo que cada um tem uma complexidade de O(n) pois utiliza a estrutura *while,* sendo que os outros métodos apenas são métodos simples como gets. Por consequência, o método *createTree()* tem como complexidade O(n log(n)) + O(n log(n)) + O(n log(n)) = O(log(n)), porque soma todas as complexidades dos métodos que lhe incluem.

Exercício 2

No exercício 2 é necessário retornar informação sobre um navio, especificamente um dos códigos (mmsi, imo, callsign), vesselName, VesselType, BaseDateTime inicial, BaseDateTime final, tempo total dos movimentos, número total de movimentos, MaxSOG, MeanSOG, MaxCOG, MeanCOG, DepartureLatitude, DepartureLongitude, ArrivalLatitude, ArrivalLongitude, TraveledDistance, Delta distance.

Para responder a tal necessidade, foi criado um método na SUPERTREE, chamado *getElementByIdentification(),* que recebe como parâmetro o código do barco, este método chama um outro com o mesmo nome que irá percorrer a árvore até encontrar o elemento. De seguida, como já temos o barco que queremos ir buscar a informação é só fazer um conjunto de *gets* ao mesmo, sendo que a nossa classe *Ship* tem uma árvore de dynamic data e a classe que cria essa árvore, a *TREE_DynamicData* possui os métodos necessários para ir buscar informação à árvore, como por exemplo o *getInicioPartidaDate*() que irá percorrer a metade esquerda da árvore recursivamente para encontrar o node de Dynamic Data mais à esquerda e uma vez que estes nós estão organizados por data, temos que o node mais à esquerda da árvore será o inicial.

Por último, estes métodos são chamados no Controller, no método *searchInformationShip()* e retorna um pair em que a key é o código do navio e o value é uma lista de Strings com a informação do barco.

Análise de Complexidade

Começando pelo método *getElementByIdenfication()* que é um método recursivo que tem complexidade Θ(log(n)) / 2, pois é um método de acesso a uma árvore binária (AVL), que no pior caso tem de aceder a metade da AVL.

De seguida analisemos os métodos gets chamados no controller, pertencentes à classe *TREE_DynamicData*:

- *getInicioPartidaDate()*: este método percorre recursivamente a árvore, comparando os elementos (dynamic data) a partir da data até chegar ao nó mais à esquerda da árvore, sendo que os métodos de acesso da árvore têm complexidade

Θ(log(n)), temos que no pior dos casos este método terá que percorrer metade da AVL, tendo deste modo complexidade Θ(log(n)) /2.

- getFimPartidaDate: este método é similar ao método acima referido só que ira retornar a data do nó mais à direita. Logo a sua complexidade é Θ(log(n)) /2.
- getNumeroTotalMovimentos: retorna o total de nós da árvore, que irá corresponder ao total de movimentos. Deste modo, navega a AVL toda, logo a sua complexidade será de Θ(log(n)).
- getMaxSog(): este método navega por todos os nodes da AVL e retorna o valor máximo da SOG. A complexidade é Θ(log(n)).
- getMeanSog(): este método navega por todos os nodes da AVL e retorna o valor medio da SOG. A complexidade é Θ(log(n)).
- getMaxCog(): este método navega por todos os nodes da AVL e retorna o valor máximo da COG. A complexidade é Θ(log(n)).
- getMeanCog(): este método navega por todos os nodes da AVL e retorna o valor medio da COG. A complexidade é Θ(log(n)).
- getDepartureLatitude(): este método retorna a latitude da partida (do início), logo retorna a latitude do node mais à esquerda da AVL, logo a sua complexidade é Θ(log(n)) /2.
- getDepartureAltitude(): este método retorna a altitude da partida (do início), logo retorna a latitude do node mais à esquerda da AVL, logo a sua complexidade é Θ(log(n)) /2.
- getArrivalLatitude(): este método retorna a latitude da chegada (do fim), logo retorna a latitude do node mais à esquerda da AVL, logo a sua complexidade é Θ(log(n)) /2.
- getArrivalAltitude(): este método retorna a altitude da chegada (do fim), logo retorna a latitude do node mais à esquerda da AVL, logo a sua complexidade é Θ(log(n)) /2.
- getTravelledDistance(): este método retorna a travelledDistance a sua complexidade é Θ(log(n)).
- getDeltaDistance(): este método retorna a DeltaDistance a sua complexidade é Θ(log(n)).

Por último analisemos a complexidade do método do controller, searchInformationShip().

- Este método constrói um pair<String,List<String>> e adiciona-lhe os dados referentes aos gets acima referidos, deste modo como temos que a inserção numa

List é de complexidade Θ(n). Esta inserção é repetida 16 vezes, porém na notação Θ temos que as constantes se removem da complexidade. Logo a inserção será apenas Θ(n). Porém estas inserções chamam os métodos dos gets referidos e estes métodos têm complexidade diferente. Deste modo, que a complexidade dos métodos irá ser:

(Θ(n)+Θ(log(n)) /2) *6+ (Θ(n)+Θ(log(n)) *6 =

=6 Θ(n) + 3 Θ(log(n)) + 6 Θ(n)+6Θ(log(n)) =

=12 Θ(n)+9 Θ(log(n)) = Θ(n)

Exercício 3

Para a obtenção das finalidades requeridas no exercício 3 (“Devolver para todos os navios o (*MMSI)*, o número total de movimentos (*NumeroTotalMovimentos)*, (TraveledDistance) e (DeltaDistance) ordenado por TraveledDistance (ordem decrescente) e número total de movimentos (ordem crescente).”), relativamente à estrutura a devolver o grupo optou por um mapa em que a chave é o *MMSI* e os valores são uma *Lista de Double,* em que se colocou na lista o *NumeroTotalMovimentos*, *TravelledDistance* e *DeltaDistance.*

Na class (*SUPERTREE)* temos o método (“*shipsMovementsAndDistance()*”), que, em primeiro lugar vai buscar os elementos de todos os nós da Árvore AVL de *Ships* e vai colocá-los numa lista (ArrayList). De seguida, usamos um *comparator* para organizar a lista por *TravelledDistance* e, no caso de as *TravelledDistances* serem iguais, vai chamar outro *comparator* que vai “desempatar” organizando por (*NumeroMovimentos)* Ordem Crescente.

De seguida, iteramos sob a lista de *Ship* com um ciclo forEach e colocamos num mapa o *MMSI* (caso não exista ainda no mapa) e, de seguida, colocamos o *NumeroMovimentos*, *travelledDistance* e *DeltaDistance* associados a cada *Ship* na Lista associada à chave

No Controller colocamos um método (*shipsMovementsAndDistanceController()*) que chama o método (*shipsMovementsAndDistance())* e itera sob o mapa devolvido por esse método e para cada chave imprime os seus valores.

Análise de Complexidade

Ciclo "for" da SUPERTREE para colocar todos os elementos da arvore numa lista tem uma complexidade de O(n);

Comparator tem complexidade O(n), pois chama o método "getTravelledDistance()" que tem complexidade O(n) e o método "compareByMovement()" tem complexidade O(log(n)), somando tudo dá uma complexidade de O(n);

Ciclo "for" para inserir informação no mapa tem complexidade n^3logn, pois, “getTravelledDistance” complexidade O(n), “getDeltaDistance” tem complexidade O(n) e “getNumeroTotalMovimentos” tem complexidade O(log(n)), logo, O(n)*O(n)*O(log(n)) = O(n^3log(n));

Método do controller tem complexidade O(n).

Logo,

O(n) + O(n) + O(n) + O(n^3log(n)) = 3O(n)+O(n^3log(n)) = O(n^3log(n))

Exercício 4

Para a obtenção das finalidades requeridas no exercício 4 ("*Obter os top-N navios com mais quilómetros percorridos (TraveledDistance) e respetiva velocidade média (MeanSOG) num período tempo (BaseDateTime inicial/final) agrupado por tipo de navio (VesselType).*"), relativamente a estrutura a devolver, o grupo optou por um mapa por cada tipo de navio (*VesselType*), onde as *keys* do mapa são referentes ao objeto "navio"(Ship) e os *values* do mapa as respetivas "velocidades médias"(Double) no intervalo de tempo introduzido pelo utilizador.

Na classe *SearchInformationController* o método *topNShipsController* limita-se a criar uma lista para todos os *VesselTypes* existentes na Tree (neste caso só utilizamos a Tree referente ao *MMSI*, mas daria com qualquer uma). Depois para cada *VesselType* chama o método *topN* da classe *SUPERTREE* e recebe o respetivo mapa referido anteriormente dando "Print" do mesmo e da *TravelledDistance* entre essas datas apenas para garantia da correta organização do método.

O método *topN* recebe como parâmetro um inteiro "*n*" (número de navios para aparecer no Top), as datas inicial e final do respetivo período de tempo a verificar e o respetivo *vesselType* (referido anteriormente). Começa por adicionar a uma lista auxiliar (*listShip*) todos os navios que tenham o mesmo vesselType que o a comparar e que tenham pelo menos duas mensagens no intervalo de tempo introduzido (*hasTwoMovementsBetweenDates* é o método que faz essa verificação retornando um *boolean*). De seguida, é através de um *comparator* relacionado com a *TravelledDistance* (*getTravelledDistanceBetweenDates* é o método que calcula a distância percorrida para o intervalo de tempo introduzido) que a lista auxiliar de navios fica organizada por ordem decrescente da distância percorrida. Finalmente o método apenas adiciona ao mapa a retornar os "n" primeiros navios da lista já organizada e a sua respetiva velocidade média (*getMeanSogBetweenDates* é o método que calcula a velocidade média daquele navio para o intervalo de tempo introduzido).

Análise de Complexidade

Relativamente à complexidade do método *TopN* da classe *SUPERTREE*, quando este adiciona à lista apenas os navios com o respetivo *vesselType* e com pelo menos duas mensagens no período de tempo introduzido tem uma complexidade de O(n^2) pois é feito numa estrutura de repetição (*for*) para todos os navios (n) e chama o método para verificar as mensagens que também é feito com base num *for*. De seguida o *comparator* e organização da lista também utiliza uma estrutura de repetição (*for*) para o tamanho total da lista chamando o método para calcular a distância para cada navio, este que tem complexidade O(n) pois utiliza um *While*. Assim, nesta fase também temos O(n) * O(n) = O(n^2). Finalmente para adicionar ao mapa os navios e a média são também utilizados uma estrutura de repetição (*for*) e chama-se o método para calcular a média no intervalo de tempo referido que também possui um *for*. Também nesta fase se tem uma complexidade de O(n^2). Assim, em conclusão, o método *TopN* tem uma complexidade de O(n^2) +O(n^2) +O(n^2) = 3O(n^2) = O(n^2).

Exercício 5

Para dar resposta à funcionalidade de obtenção dos pares de navios com rotas com coordenadas de partida/chegada próximas e com diferentes *TraveledDistances*, ordenados pelo código MMSI do 1º navio e por ordem decrescente da diferença de *TraveledDistance* recorreu-se à implementação descrita de seguida.

Na classe *SUPERTREE* encontra-se o método *naviosPares* responsável por devolver a lista dos pares de navios que respeitam as regras referidas acima. Para isso, começa-se por criar uma lista de todos os navios com *TraveledDistances* superiores a 10 quilómetros, através da obtenção de todos os navios recorrendo ao método *inOrder* da classe BST, garantindo não só a regra de não consideração de navios com *TraveledDistances* inferior a 10 quilómetros, como também a ordenação pelo código MMSI do 1ºnavio.

De seguida, procede-se à construção da lista dos pares de navios que obedecem aos restantes parâmetros ainda não verificados. Assim, através da iteração de dois ciclos da lista de navios criada no passo anterior, compara-se todos os pares de navios possíveis. Para cada um dos pares de navios possíveis são calculadas as distâncias de partida e chegada, recorrendo ao método de cálculo da distância da classe *DistancesCalculus* com as latitudes e longitudes de partida/chegada de cada um dos navios. Posteriormente, é verificada se essas distâncias são menores que 5 quilómetros e em caso afirmativo é criado um par de navios e consequentemente adicionado à lista dos pares de navios que obedecem a todas as regras. O procedimento anterior é realizado para todos os pares de navios possíveis.

Por último, resta apenas proceder à ordenação da lista previamente criada por ordem decrescente da diferença de *TraveledDistance*. Assim sendo, recorreu-se à implementação de um *comparator* que compara o módulo da diferença das *traveledDistances* de cada par de navios da lista, permitindo a ordenação por ordem decrescente da lista que será retornada.

No método *paresIguais* da classe *SearchInformationController* é feita a chamada do método naviosPares da classe SUPERTREE que retorna a lista dos pares de navios. Além disso, este método é responsável pela formatação da mostragem da lista resultante.

Análise de Complexidade

A análise de complexidade do método *naviosPares* da classe *SUPERTREE* obedece à seguinte estrutura.

Primeiramente o método *inOrder* da BST, utilizado para obter todos os elementos da árvore por ordem crescente possui uma complexidade O(log(n)). De seguida são iterados todos os elementos dessa lista recorrendo a um ciclo "for", o qual é iterado pelos n elementos da lista, perfazendo uma complexidade de O(n).

Por conseguinte são iterados dois ciclos "for" o que origina uma complexidade O(n^2), ou seja, O(n) *O(n). Dentro destes ciclos é calculada a distância de partida e chegada, necessitando dos valores da longitude e latitude, tendo estes métodos de busca uma complexidade O(log(n)). A inserção dos pares na lista tem uma complexidade O(1), assim como o "if" de verificação dos parâmetros da distância de chegada e partida.

Por último, o *comparator* apresenta uma complexidade O(N), pois chama o método "*getTraveledDistance*", o qual possui essa mesma complexidade. O método do *controller* apresenta uma complexidade O(1).

Logo,

O(log(n)) + O(n) + [O(n^2)*(O(log(n)*O(log(n))] + O(1) + O(n) + O(1) = O(log(n)) + O(n) + O(n^2) = O(n^2).

Contribuição de cada membro do grupo

Todo o trabalho (estrutura geral, user stories e testes) contou com a cooperação de toda a equipa para a resolução do mesmo, sendo que cada membro não se preocupou apenas em fazer a sua tarefa designada, mas também em ajudar na resolução de todo o trabalho.