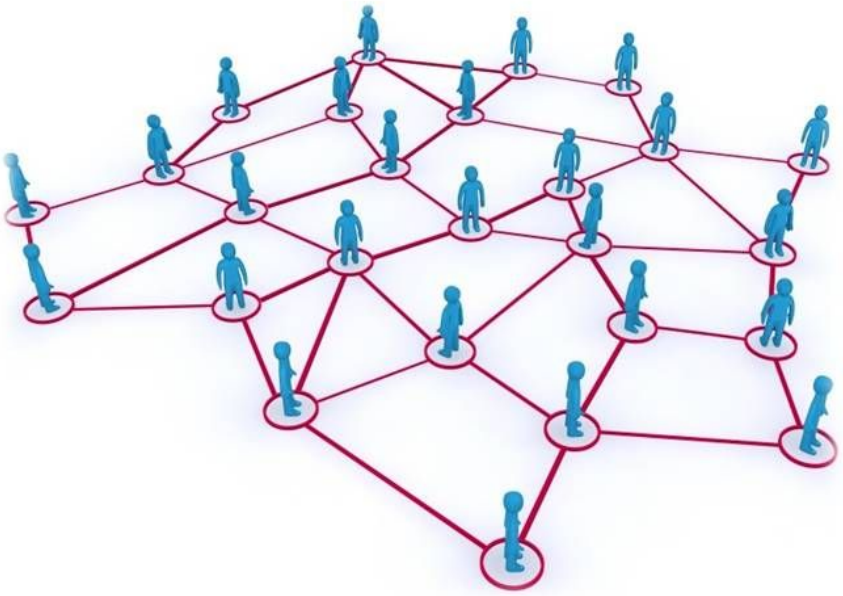
P.PORTO

ESCOLA
SUPERIOR
DE TECNOLOGIA
E GESTÃO

Estruturas de Dados

Trabalho Prático



Grafo Social

Discentes

Célio Ighour de Castro Rodrigues nº 18180016

Jorge Miguel Pinto Araújo nº 8150285

Felgueiras, Janeiro de 2019

Índice

1. Introdução

Atualmente na pesquisa por novos colaboradores para uma empresa, não só as competências profissionais são consideradas, mas também o seu aspeto social. Então para facilitar o encontro entre empresas e utilizadores, poderá existir uma plataforma social de recursos humanos, que permita que facilmente se encontre um colaborador com as competências desejadas.

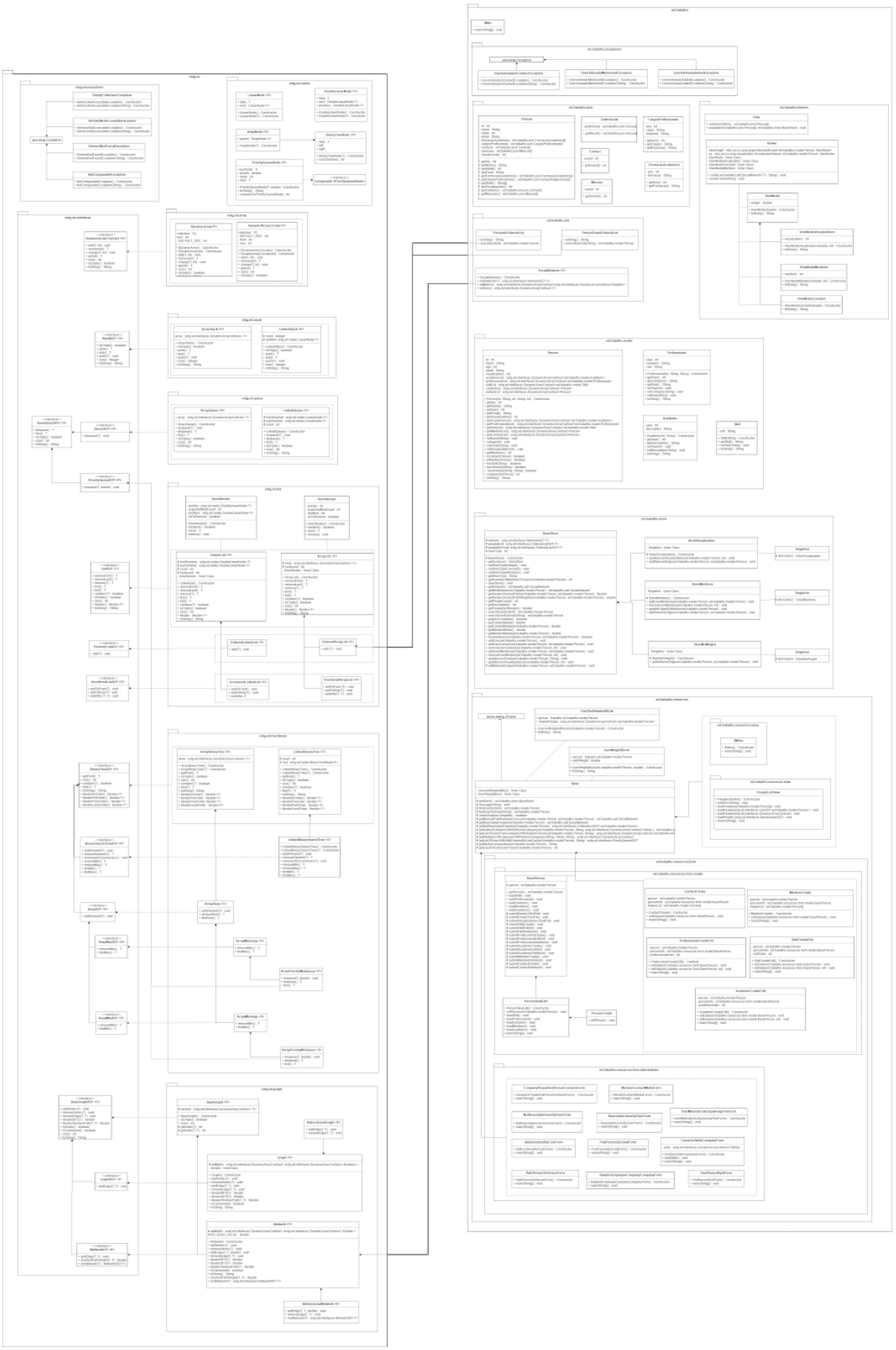
O objetivo principal deste trabalho passa por criar um programa que ajude a encontrar esses candidatos com as competências sociais e técnicas desejadas. Para tal, o programa lê de um ficheiro JSON, que contém perfis de utilizadores e implementa algoritmos adequados para gerar um rede social com os envolvidos.

Com a realização deste trabalho, pretende-se que se cumpram os seguintes objetivos:

- Compreender e dominar os conhecimentos teóricos e práticos sobre Estruturas de Dados e os algoritmos que lhes estão associados;
- Saber escolher, de forma fundamentada e para cada problema específico, qual a estrutura de dados abstrata e implementação mais adequada;
- Implementar algoritmos escaláveis sobre estruturas de dados potencialmente grandes;
- Adquirir competências com vista à resolução de problemas compostos;
- Estimular o trabalho em equipa como elemento essencial do processo de aprendizagem individual.

2. Diagrama de Classes

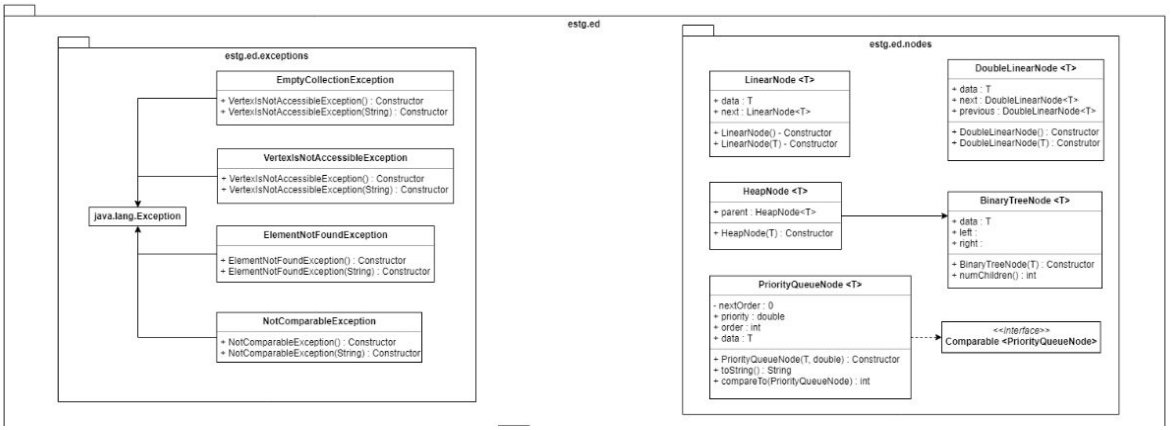
Foram mapeadas todas as classes utilizadas da *library* própria e as do desenvolvimento do trabalho em si, resultando no seguinte diagrama de classes (que será repartido posteriormente para melhor entendimento):



Em síntese, ao lado esquerdo encontram-se as classes da *library* e ao lado direito as utilizadas para desenvolver o *Grafo Social*. As classes também foram organizadas de acordo com os *packages* a que pertencem no trabalho prático.

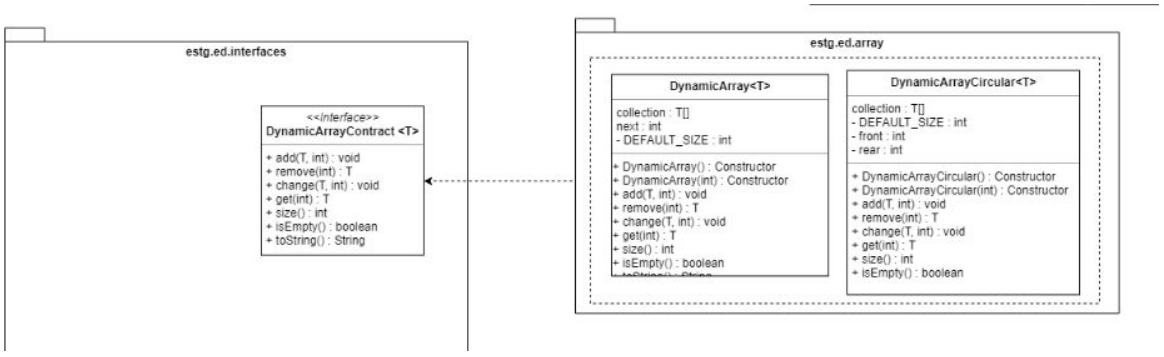
Devido à extensão do diagrama, as setas foram utilizadas apenas para indicar herança (seta comum) e implementação de interface (seta pontilhada). Além disso, os relacionamentos entre as classes não são representados por setas, apenas constando na assinatura do método e/ou atributo a referência completa à classe que é utilizada (contendo o *package*).

2.1. *Library* - *Exceptions* e *Nodes*

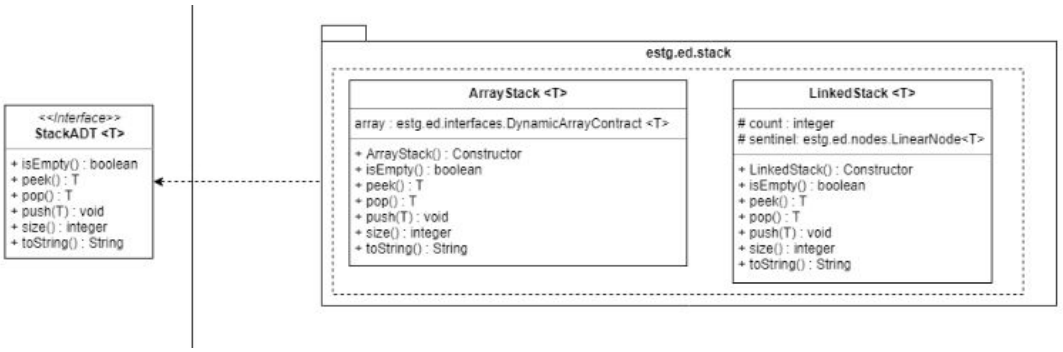


Tratam-se das exceções próprias utilizadas nas estruturas de dados, todas herdando da classe *Exception* do java e dos nós utilizados em listas ligadas e outras.

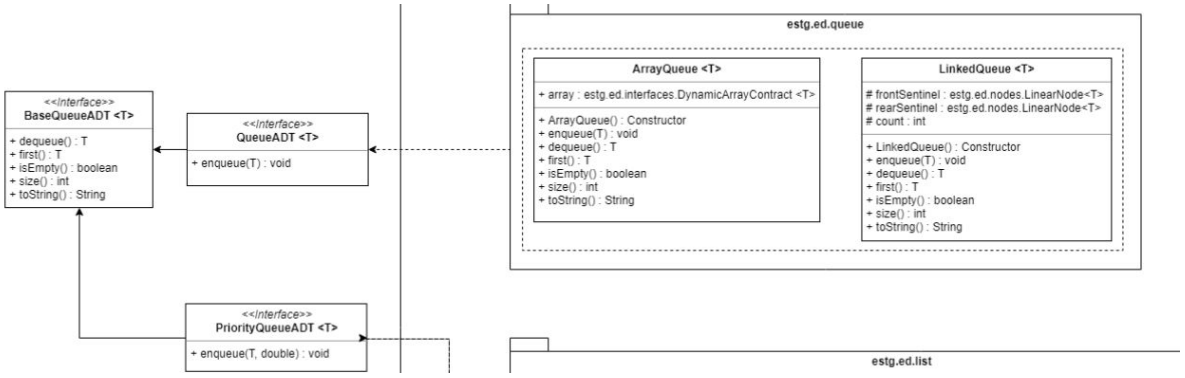
2.2. *Library* - *Array* Dinâmico



2.3. *Library - Stack*



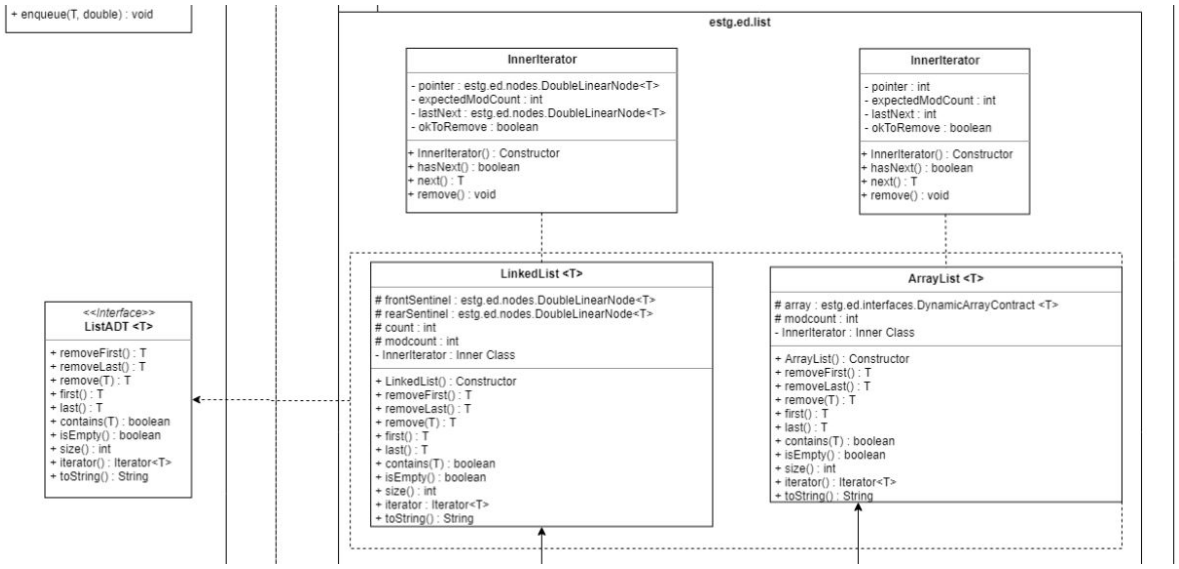
2.4. *Library - Queue* e *PriorityQueueADT*



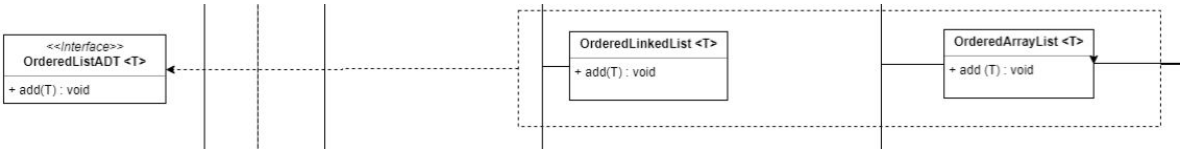
Observe-se que, no caso da *queue*, a interface foi dividida em uma básica (que serve para qualquer tipo de fila) e duas especializadas, uma para filas normais e outra para filas de prioridade.

A implementação da *PriorityQueue* dá-se no *package* de árvores binárias por herdar sua estrutura de um *heap*, apenas observando a interface de uma fila de prioridade.

2.5. *Library* - Base das Listas

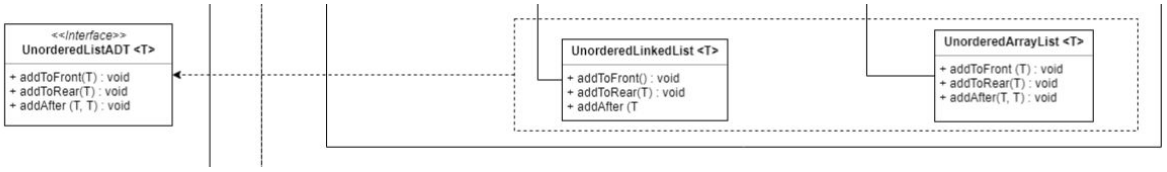


2.6. *Library - OrderedList*



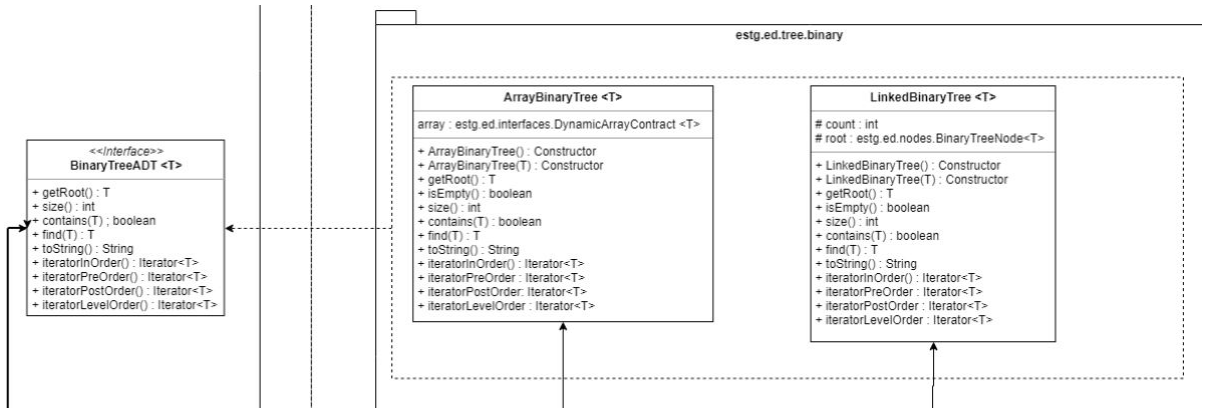
Ambas herdando das respectivas listas base (*LinkedList* ou *ArrayList*).

2.7. *Library - UnorderedList*

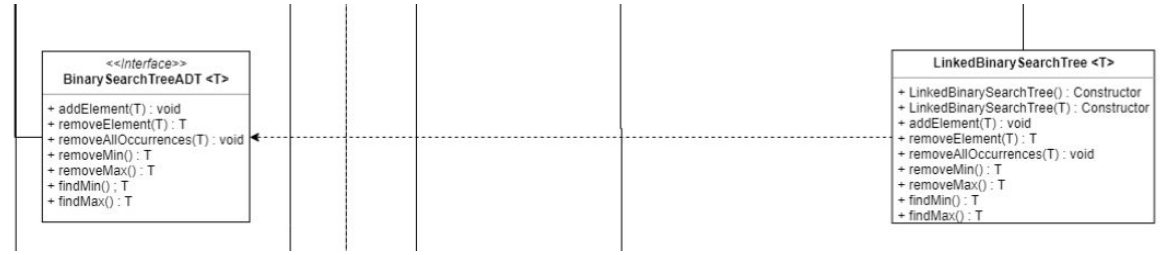


Ambas herdando das respectivas listas base (*LinkedList* ou *ArrayList*).

2.8. *Library - BinaryTree*

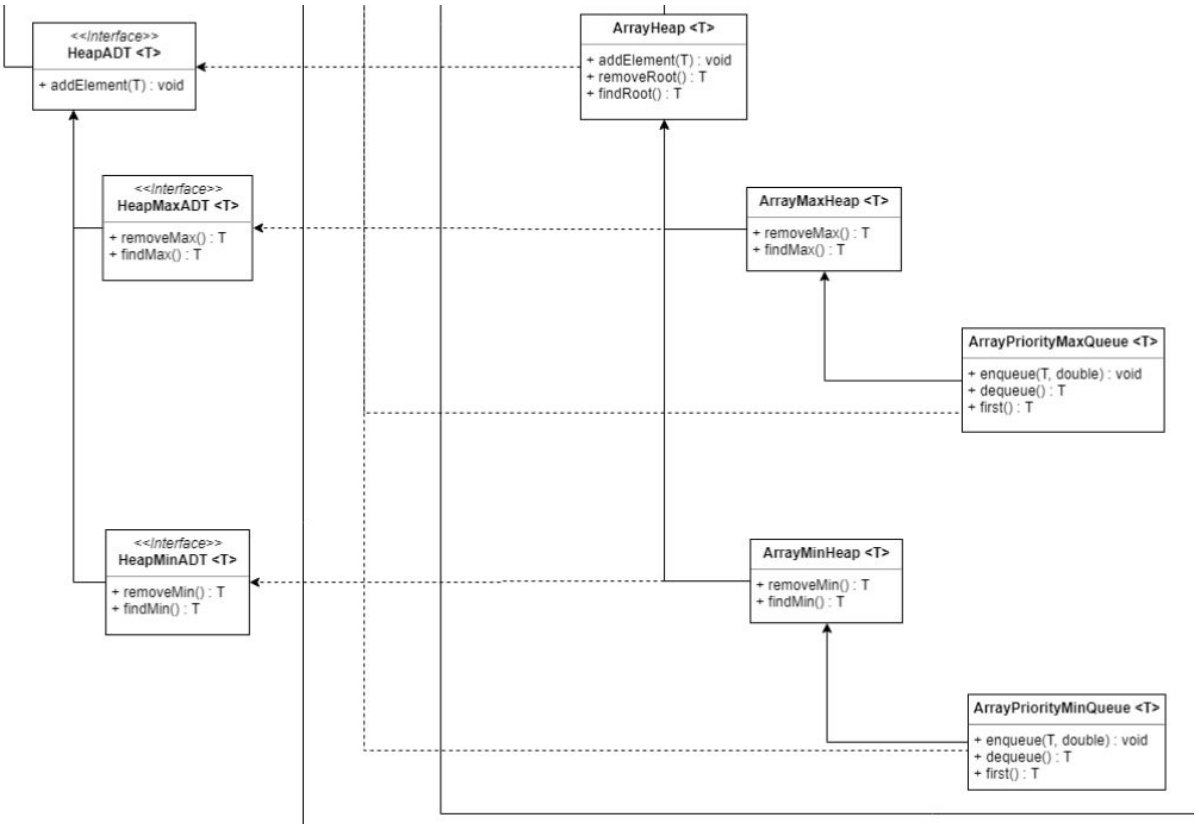


2.9. *Library - BinarySearchTree*



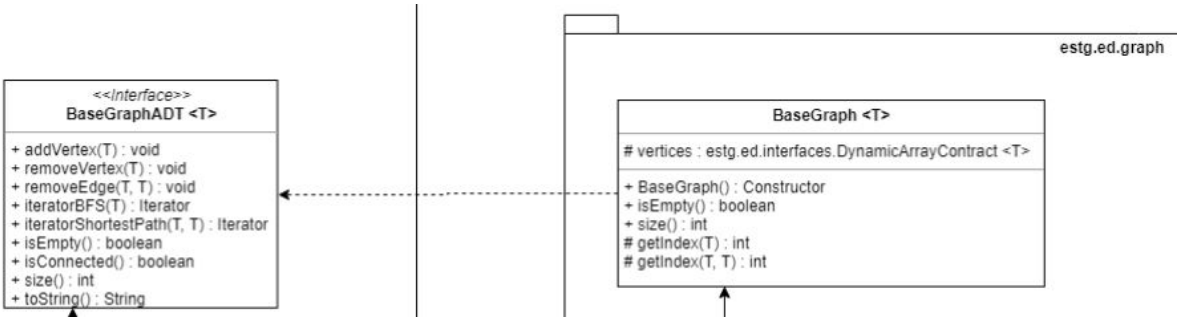
Herdando de *BinaryTree*.

2.10. *Library* - *Heap* e *PriorityQueue*



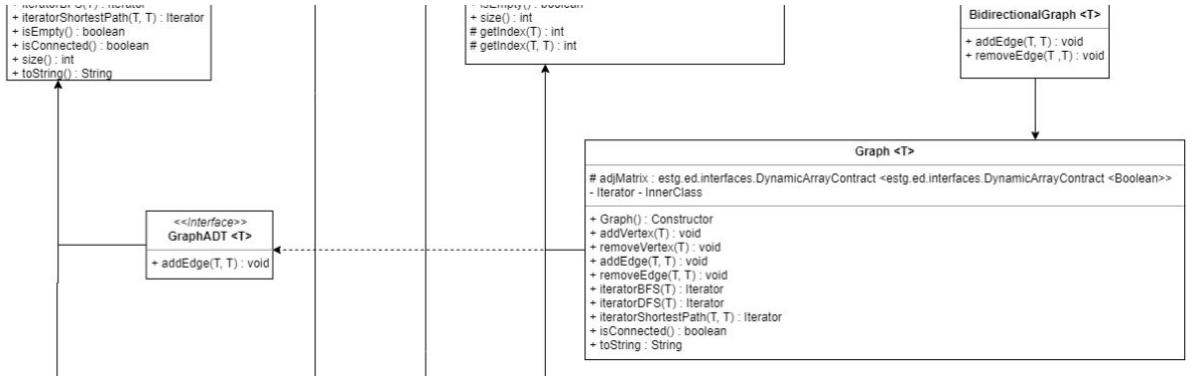
O *heap* herda da *BinaryTree*, podendo ser implementado como *minheap* ou *maxheap* e, consequentemente, fila de prioridade mínima ou fila de prioridade máxima.

2.11. *Library* - Base para *Graph* e *Network*



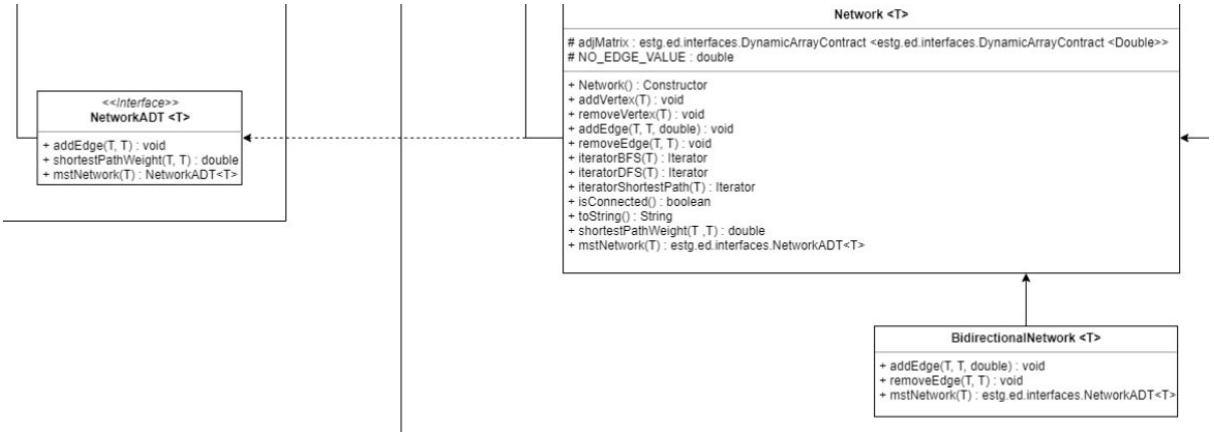
Contém assinaturas que pertencem a ambas as classes *graph* e *network*.

2.12. *Library - Graph*

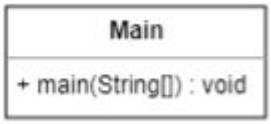


Herdando da base para grafos e redes.

2.13. *Library - Network*

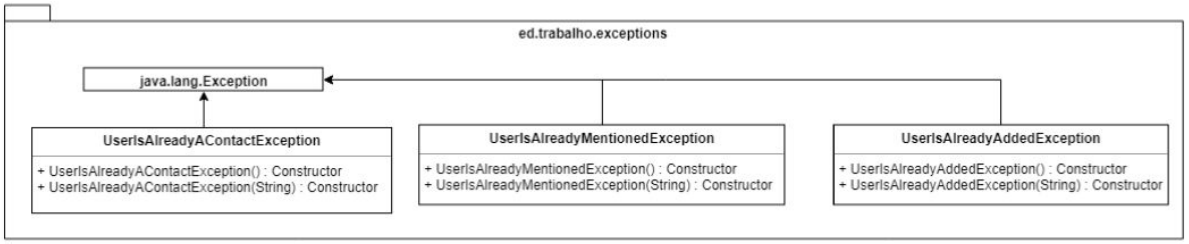


2.14. Grafo Social - *Main*



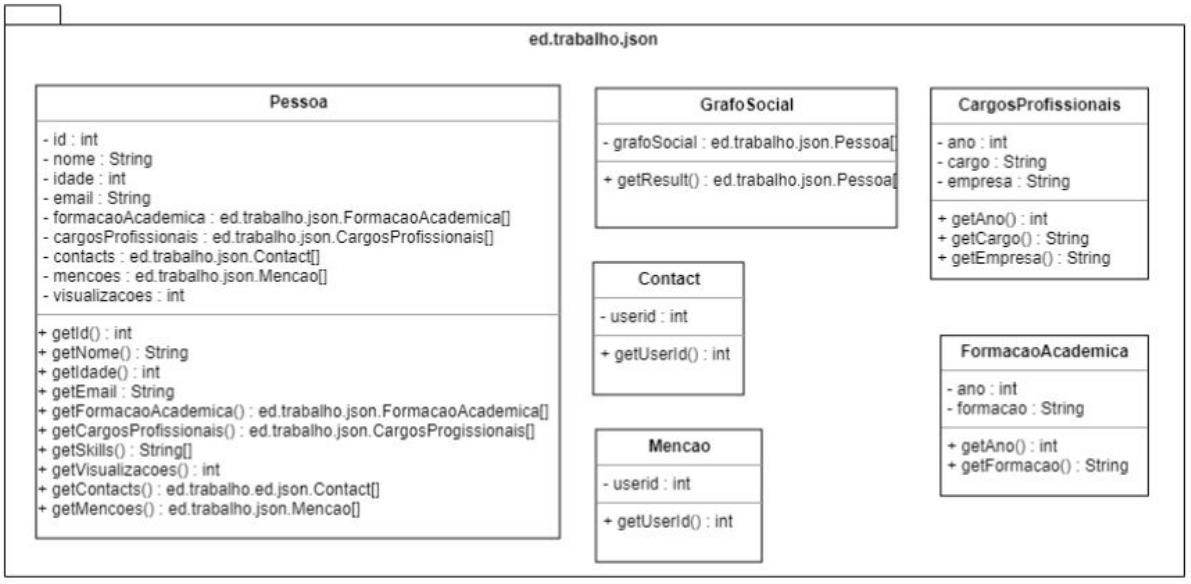
É a classe que inicializa o menu interativo e, por consequência, toda a aplicação.

2.15. Grafo Social - *Exceptions*



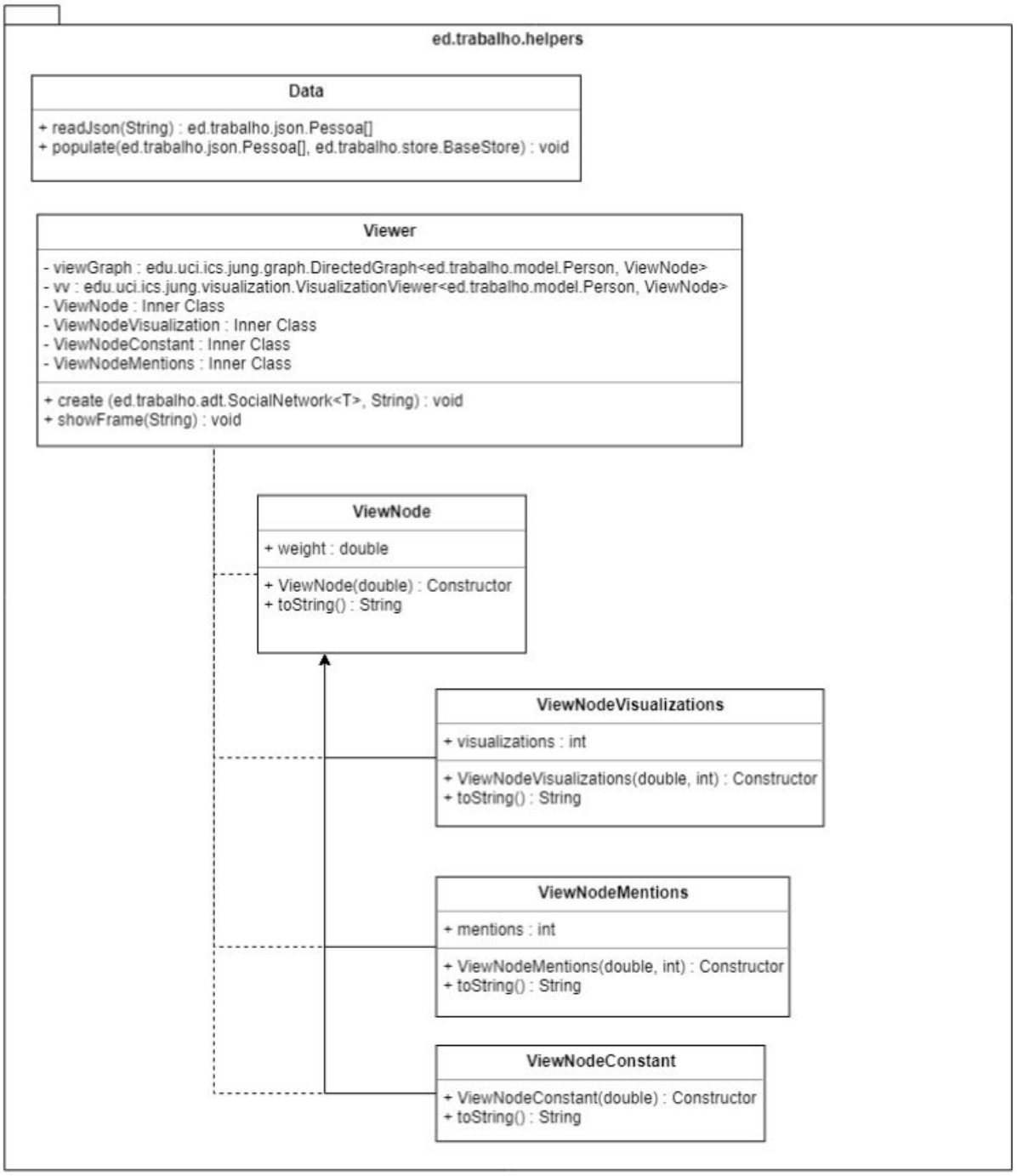
Exceções que são utilizadas no grafo social, ambas herdando da classe *exception* do Java.

2.16. Grafo Social - *JSON*



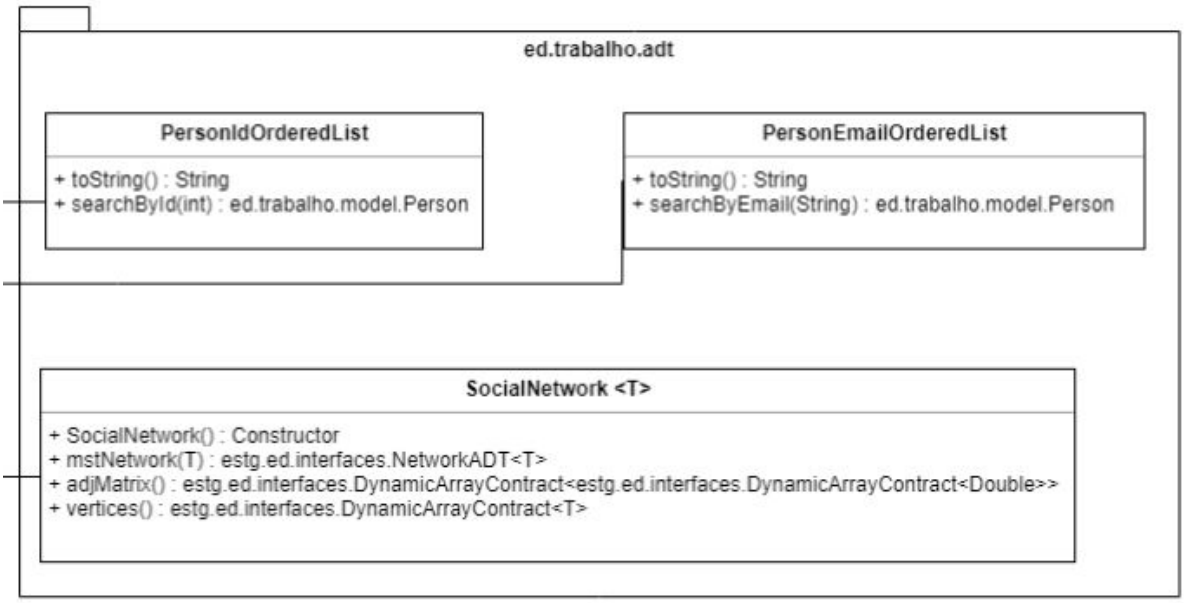
Classes que são utilizadas pela *API* do Gson para mapear um *json* em classes do java. Por esse motivo, o nome dos atributos equivale ao dos atributos recebidos no *json*.

2.17. Grafo Social - *Helpers*



Classes auxiliares para popular o grafo social com os dados recebidos do *json* (após convertidos pelo *Gson*) e para visualizar o grafo com a *API* do *Jung*.

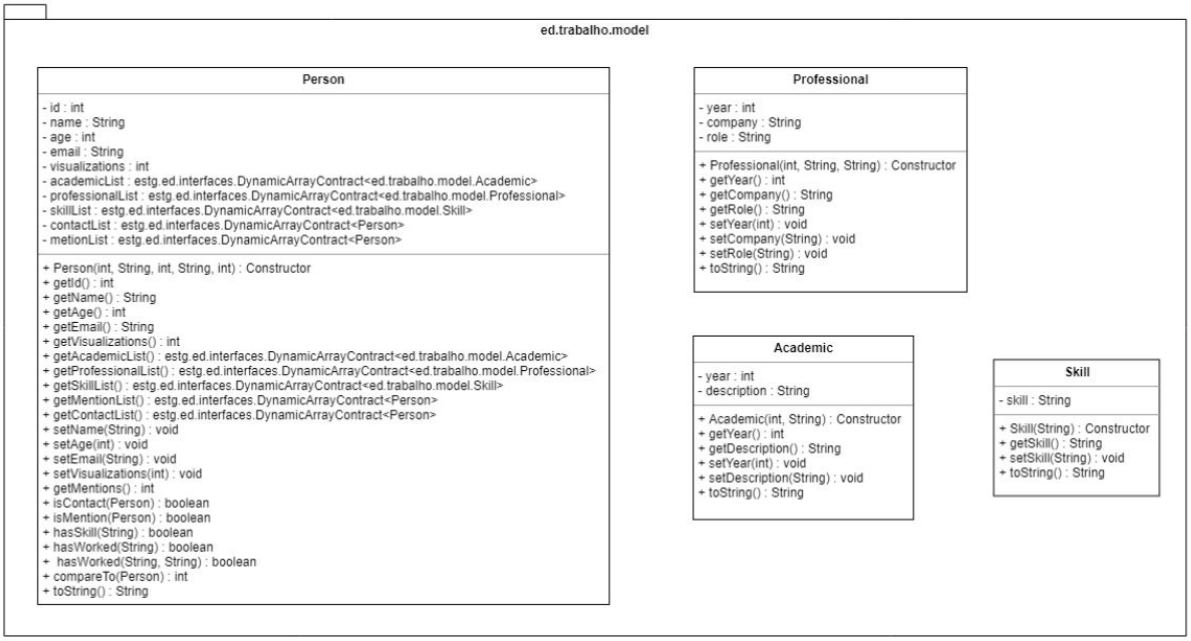
2.18. Grafo Social - Estruturas de Dados Modificadas



Algumas estruturas abstratas de dados da *library* foram modificadas para otimizar a utilização no grafo social.

Dessa forma, as listas ordenadas acrescentam um método para pesquisa de um utilizador por um atributo (id ou email), e a rede acrescenta métodos para a obtenção dos dados brutos dos vértices e matriz de adjacência, para utilização no *Jung* e outros.

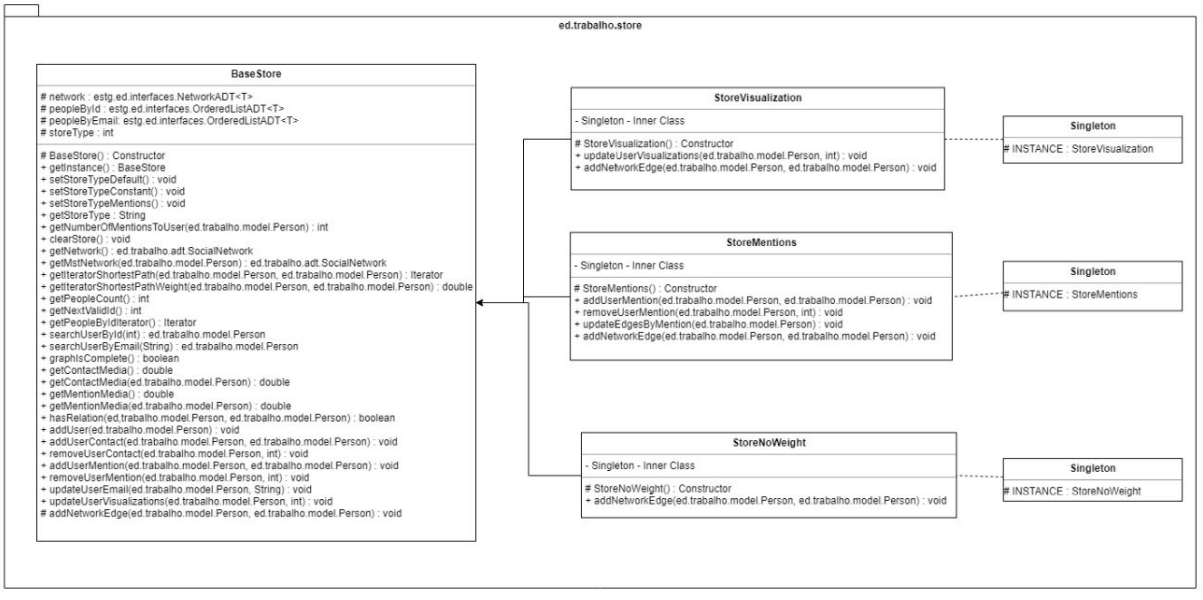
2.19. Grafo Social - *Model*



Contém os modelos representativos dos utilizadores e seus atributos, incluindo uma classe própria para cada histórico profissional ou académico e para as *skills*.

Foram incluídos alguns métodos auxiliares para facilitar a obtenção de informação associada a um utilizador ou seus relacionamentos.

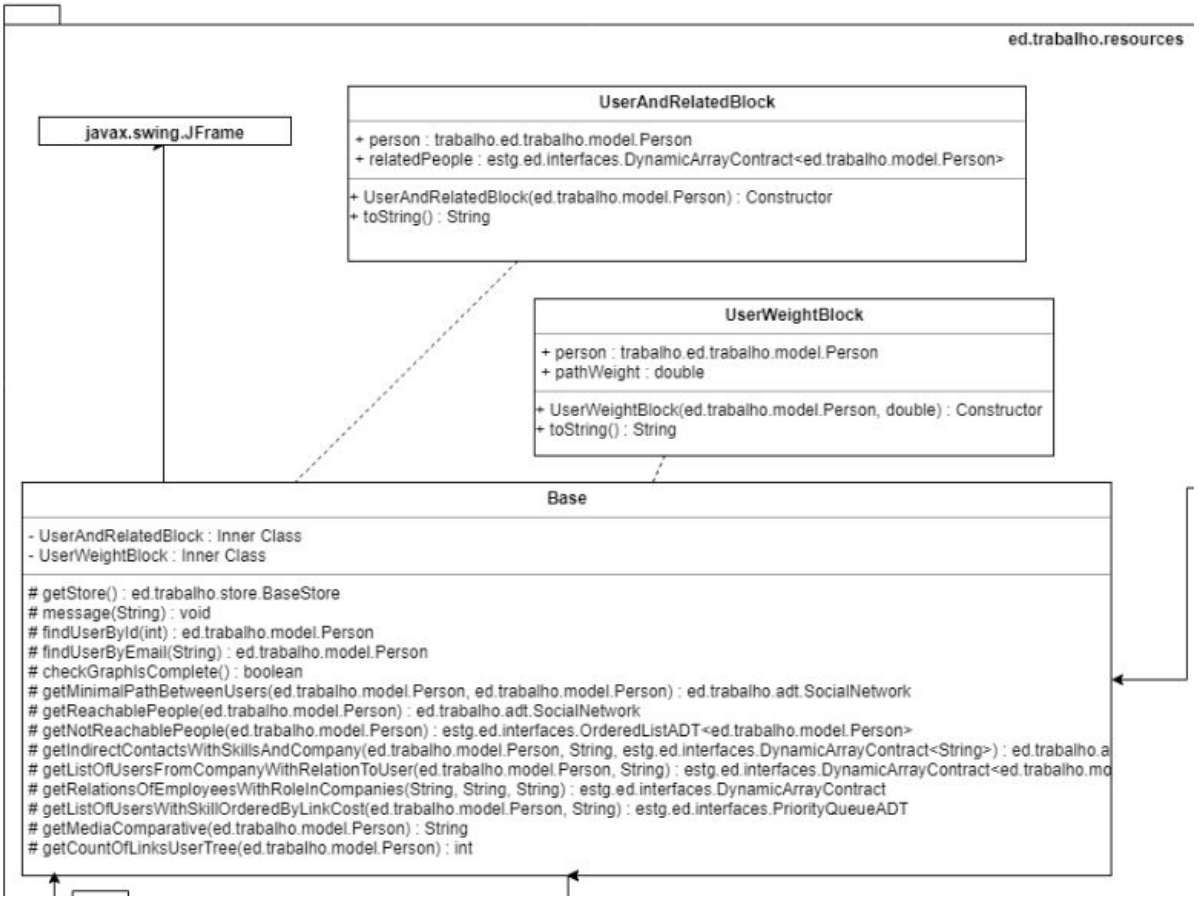
2.20. Grafo Social - *Store*



É a estrutura responsável por armazenar os dados dos utilizadores na rede e em listas ordenadas, contendo a “interface” para a aplicação obter tais informações e manipulá-las, ou seja, tudo se passa por meio da *store*, facilitando a coesão nas mudanças ocorridas durante a execução.

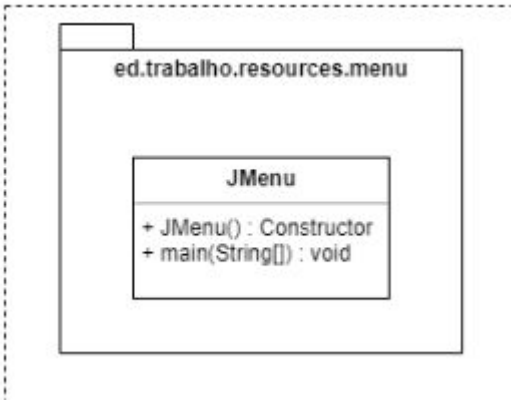
Pode ser subdividida em tipos conforme os tipos de pesos que forem utilizados no grafo (1/visualizaçãoes, 1/menções ou 1), aplicando o padrão *singleton* para o recebimento da instância do tipo em uso.

2.21. Grafo Social - *Resources* - Base



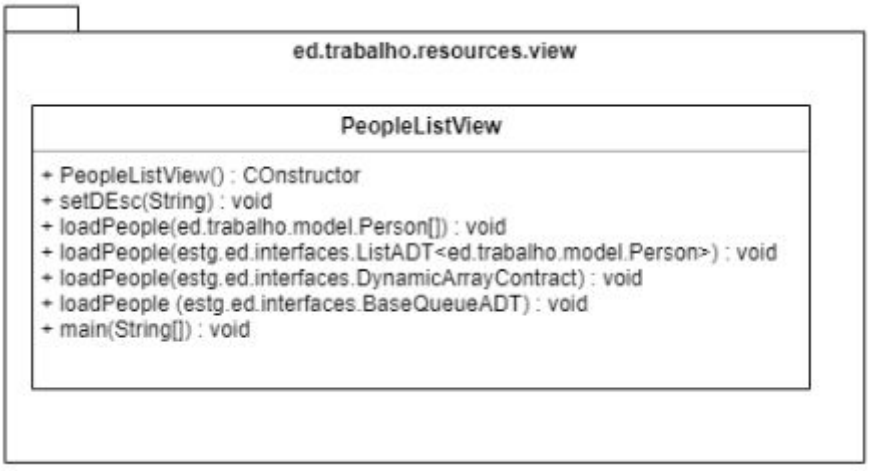
A manipulação do grafo social dá-se por meio de menu interativo construído com *swing*, em que os componentes utilizados herdam de uma classe base, contendo os métodos necessários para a execução das operações desejadas. A classe base herda do JFrame da *API* do swing.

2.22. Grafo Social - *Resources* - Menu



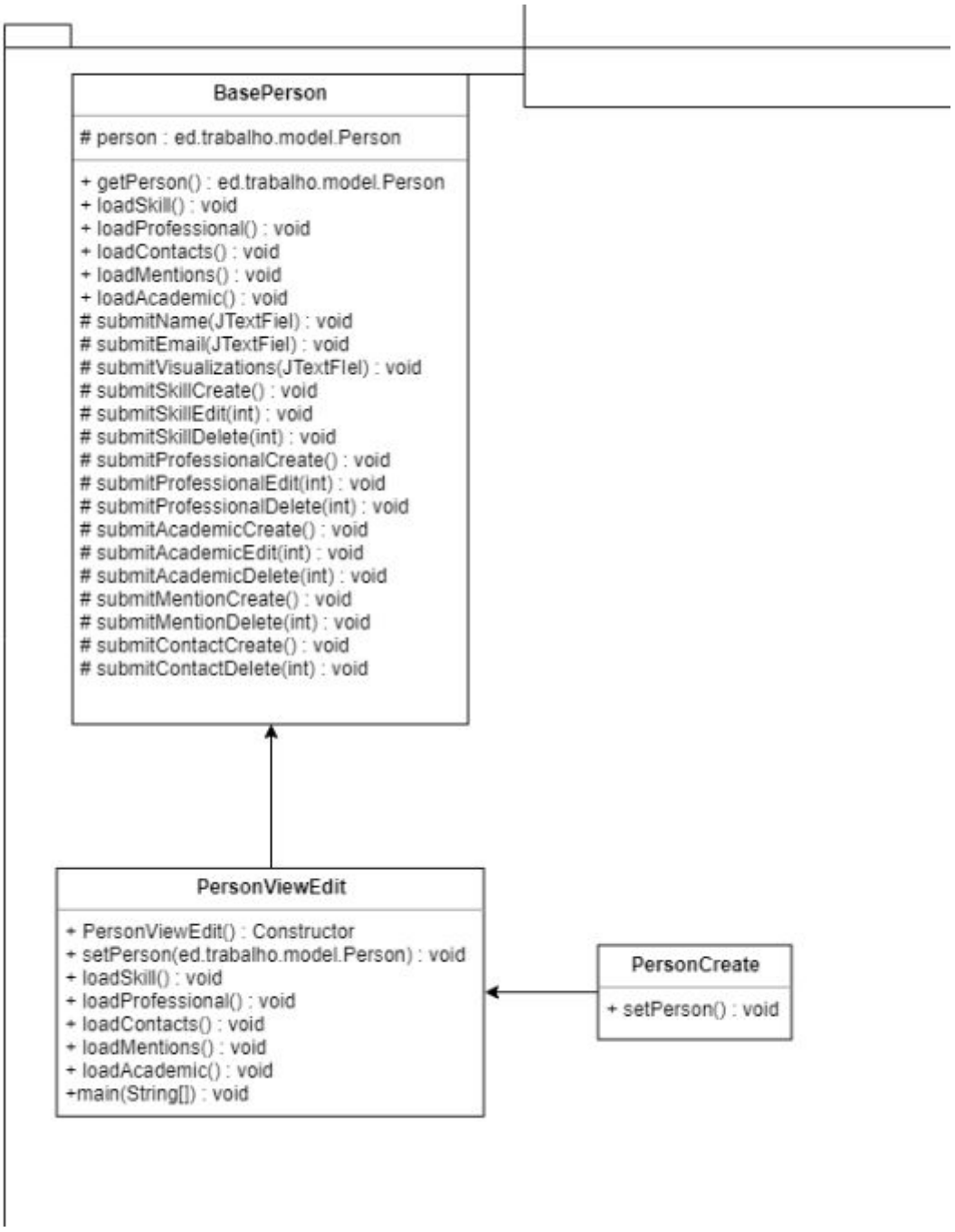
É a classe que renderiza o menu da aplicação, por onde podem ser executadas todas as ações e manipulações possíveis.

2.23. Grafo Social - *Resources* - Lista de Pessoas



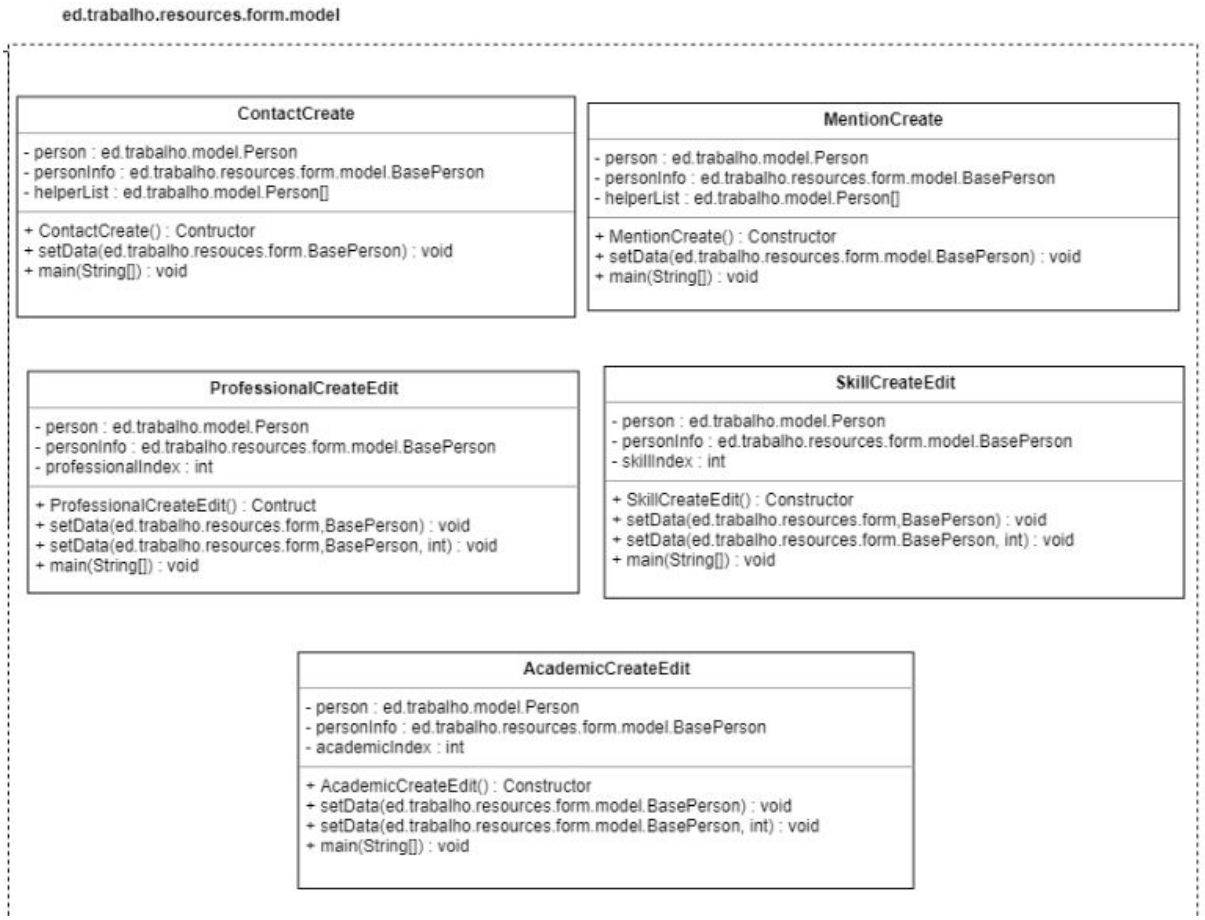
Componente reutilizável para expor uma lista de utilizadores como resultado de uma operação qualquer.

2.24. Grafo Social - *Resources* - Perfil do Utilizador



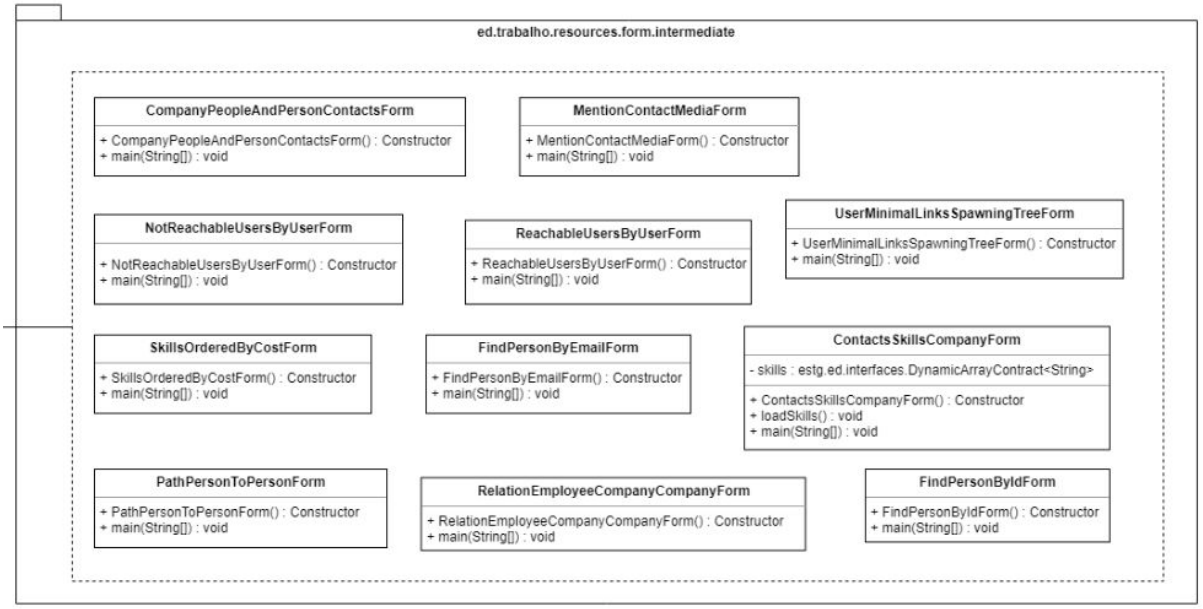
São componentes para visualizar o perfil do utilizador e a sua criação, além da edição dos atributos que não pertençam a outro *model* (nesse caso, apenas instancia o componente responsável para manipular o *model* dependente).

2.25. Grafo Social - *Resources* - Relacionamentos do Utilizador com outros Modelos



São classes que permitem a manipulação das listas associadas a um utilizador (profissão, educação etc), bem como dos contatos e menções.

2.26. Grafo Social - *Resources* - Formulários Intermediários



São *frames* contendo formulários para encaminhamento de uma ação principal. Por exemplo, um formulário que pede o email do utilizador para mostrar o seu perfil; ou um formulário que pede os dois utilizadores para demonstrar qual o caminho mais curto entre ambos.

3. Estruturas de Dados Utilizadas

Os dados obtidos através da leitura de um ficheiro *JSON* são armazenados em três estruturas especiais para o correto funcionamento de toda a aplicação, uma rede (*SocialNetwork*) e duas listas ordenadas (*PersonIdOrderedList* e *PersonEmailOrderedList*), onde estão as referências a todos os utilizadores.

Outras estruturas abstratas de dados são utilizadas também para o cumprimento e execução de alguns requisitos do trabalho prático, em que será necessário processar os dados dos utilizadores ou de eventuais *inputs* de formulários e gerar tais estruturas para suportar o resultado de tal funcionalidade.

Ainda, algumas estruturas utilizam internamente outras estruturas abstratas de dados para o seu correto funcionamento.

Dessa forma, todas as estruturas abstratas de dados utilizadas na aplicação serão discorridas brevemente a seguir.

3.1. *SocialNetwork<T>*

A estrutura abstrata de dados de rede foi pontualmente modificada para atender aos requisitos do trabalho prático. Dessa forma, foi construída uma nova estrutura que herda da NetworkADT<T>.

Tal estrutura acresceu dois métodos auxiliares para a externalização da rede com a *API* do *Jung* (para mostrar visualmente grafos e redes), sendo eles o *adjacencyMatrix()*, que retorna a matriz de adjacência, e o *vertices()*, que retorna a lista de vértices.

Para tanto, também foi necessário realizar um *override* ao método *mstNetwork(T)*, que retorna uma *minimum spanning tree* a partir de um determinado elemento, de modo a construir uma instância de *SocialNetwork*, ao invés de uma *Network*.

Sendo assim, o valor a ser considerado como inexistência de ligação entre elementos é o de *Double.Negative_Infinity*, tendo em vista que o padrão utilizado na estrutura *NetworkADT<T>* era de *Double.Positive_Infinity* e o caso em concreto permite que existam arestas com peso infinito (1/0, quando houver quantidade de visualizações ou menções iguais a 0).

3.2. *PersonIdOrderedList* e *PersonEmailOrderedList*

Tendo em vista que os utilizadores do grafo social seriam intensamente pesquisados pelo seu identificador (ID) e email, agregamos juntamente à rede duas listas ordenadas (por ID e por email) contendo a referência de todos os utilizadores.

Dessa forma, é realizada uma pesquisa na lista para encontrar um utilizador, por meio dos métodos *searchById(id)* e *searchByEmail(email)*, ambos utilizando-se da recursão com pesquisa binária para encontrar o alvo.

Ambas as listas herdam da estrutura de dados *OrderedArrayList*<*T*>, em que o tipo parametrizado *T* é *Person*, classe criada para representar uma pessoa no grafo, e implementam a interface *OrderedListADT*<*T*>.

Foi escolhida a implementação com *arrays* principalmente pela possibilidade permitir a pesquisa binária para encontrar rapidamente um utilizador, pois caso fosse implementada com listas ligadas, seria necessário percorrer todos os nós a cada busca, invalidando a sua principal utilidade.

3.3. *OrderedArrayList<T>*

A implementação da lista ordenada é feita com *array* (motivos explicados no tópico 3.2), implementando a interface *OrderedListADT*<*T*> e herdando da classe *ArrayList*<*T*>, com apenas a inclusão do método de adição de um elemento, momento em que deverá ser escolhida a posição correta para que a lista mantenha-se ordenada.

3.4. *ArrayList<T>*

A implementação de lista é feita com *array* (motivos explicados no tópico 3.2), observando a interface *ListADT*<*T*> e provendo um conjunto de métodos para gestão de uma lista, exceto os de adicionar, que devem ser implementados pela lista ordenada ou pela lista não ordenada.

Internamente os elementos são armazenados em uma estrutura de dados criada para gerir *arrays* de forma dinâmica, o *DynamicArrayCircular*<*T*>, que implementa funções básicas de um *array* (adicionar, remover, ler, alterar valor, tamanho etc), listadas na interface *DynamicArrayContract*<*T*>.

3.5. *DynamicArray<T>* e *DynamicArrayCircular<T>*

Tais estruturas foram criadas simplesmente para facilitar a gestão de operações em um *array* comum, de modo a abstrair a sua necessidade de crescimento (é realizada de forma automática) e facilitar a obtenção de algumas informações do *array* (e.g. verificar se está vazio).

Ambas as estruturas observam uma interface comum, a *DynamicArrayContract<T>*, com os métodos *add(T, index)*, *remove(index)*, *change(T, index)*, *get(index)*, *size()* e *isEmpty()*.

A diferença entre ambas é que uma delas constrói um *array* circular, de modo que a remoção tanto no início como no fim do *array* é de complexidade temporal constante, pois não será necessário empurrar/puxar elementos após tal ato.

Assim sendo, a remoção ou inserção de elementos no meio do *array*, em ambos os casos, ocasiona a deslocação de elementos no *array*.

3.6. *Network<T>*

A estrutura de dados de rede é a base da aplicação, principalmente por ser herdada pela *SocialNetwork<T>*, mas também por fornecer a implementação de diversos métodos utilizados, como *addVertex(T)*, *removeVertex(T)*, *addEdge(T1, T2, weight*), *removeEdge(T1, T2*), *iteratorBFS(T)*, *iteratorDFS(T)*, *iteratorShortestPath(T1, T2)*, *isConnected()*, *shortestPathWeight(T1, T2)* e *mstNetwork(T)*, em observância com a interface *NetworkADT<T>*.

Ela herda da classe *BaseGraph<T>* e implementa tais métodos em observância à interface *NetworkADT<T>*, acrescentando o atributo da matriz de adjacência que, no caso, utiliza como tipo um *double* para armazenar os pesos das arestas.

Tal estrutura utiliza o valor *Double.Positive_Infinity* como marcador para a inexistência de ligação entre vértices e armazena a sua matriz de adjacência em uma estrutura pautada em *array*, de melhor gestão, utilizando a classe *DynamicArrayCircular<T>*, anteriormente explicitada.

A estrutura de dados *UnorderedArrayList<T>* é utilizada pelo método *iteratorBFS()* para armazenar o resultado (lista de vértices), incluindo o auxílio de um *ArrayQueue<T>* para gerir a ordem de armazenamento no resultado.

Dessa forma, o primeiro vértice a entrar na *queue* é o primeiro a entrar no resultado, provendo a iteração no modo largura primeiro (primeiro acessa os vizinhos e depois aprofunda).

A preferência por uma *UnorderedArrayList<T>* ao invés de uma *UnorderedLinkedList<T>* dá-se pelo facto de o resultado ser armazenado sempre ao final da lista, não ocasionando as operações de inserção ou remoção ao meio do *array* (são menos performáticas que em uma lista ligada) e, ao mesmo tempo, necessitar de menor gestão de memória.

Da mesma forma, no caso da *ArrayQueue<T>*, é utilizada internamente um *array* circular, não ocasionando pior performance para a remoção no início do *array*, enquanto que no caso de utilização de uma *LinkedQueue<T>*, seria necessário uma maior gestão da remoção e inserção de elementos na lista ligada.

O método *iteratorDFS(T)* também utiliza uma *UnorderedArrayList<T>* para armazenar o resultado, só que auxiliado pela estrutura *ArrayStack<T>*, tendo em vista que deve atravessar em profundidade primeiro (percorrendo os vizinhos dos vizinhos em sequência).

O *ArrayStack<T>* é preferido ao *LinkedStack<T>* por não ocasionar inserção ou remoção no meio do *array* e necessitar de menor gestão dos elementos (comparado à lista ligada).

No caso do método *iteratorShortestPath<T1, T2>*, além da utilização de uma *UnorderedArrayList<T>*, é utilizada uma *ArrayPriorityMinQueue<>* para armazenar os vértices vizinhos durante a travessia com o custo total até chegar àquele vértice, de modo a ser retirado sempre o caminho acumulado mais curto até determinado vértice.

Tais estruturas também são utilizadas pelo método *mstNetwork(T)*, com a diferença de que a fila de prioridade armazena apenas o custo do *edge* entre os vértices, e não o custo total até chegar no vértice desejado.

A preferência pelo uso de *array* na fila de prioridade dá-se por utilizar internamente um *minheap*, que armazena a sua estrutura de árvore em um *array*, obedecendo às regras de *heap*, ou seja, a árvore tem de ser completa (equilibrada, por não ter um desnível entre as folhas maior que um, e com os elementos folha mais à esquerda possível).

Com isso, não há um grande desperdício de espaço do *array*. Por outro lado, caso fosse utilizado um *heap* baseado em listas ligadas, a gestão dos nós e das alterações dos mesmos seria mais cara.

3.7. *BaseGraph<T>*

Essa classe abstrata agrega métodos comuns tanto a um grafo como a uma rede, tais como *isEmpty()* e *size()*, observando a interface *BaseGraphADT<T>*. Ademais, o atributo *vertices* (instância de *DynamicArrayCircular*) é inicializado no seu construtor, pois pode ser reaproveitado no grafo e na rede, diferentemente do atributo *adjMatrix*, que é diferenciado.

3.8. *UnorderedArrayList<T>*

A classe de lista não-ordenada é implementada com o uso de *array* (*DynamicArrayCircular<T>*), agregando à *ArrayList<T>* os métodos *addToFront(T)*, *addToRear(T)* e *addAfter(T1, T2)*.

Os motivos da utilização de um *array* ao invés de uma lista ligada é explicitado no tópico 3.6 (*Network<T>*), onde se dá o seu uso.

3.9. *ArrayQueue<T>*

O *ArrayQueue<T>* é implementado com o uso de um *array* circular (*DynamicArrayCircular<T>*), de modo a não impactar negativamente na performance a remoção em uma ponta e a inserção em outra ponta.

Os motivos de sua utilização também são melhor explicitados no tópico 3.6 (*Network<T>*), onde se dá o seu uso.

3.10. *ArrayStack<T>*

O *ArrayStack<T>* é implementado com o uso de um *array* (*DynamicArray<T>*), sendo a inserção e remoção sempre ao final deste, não havendo maiores impactos em sua performance, comparado à implementação com lista ligada, mas reduzindo o custo de gestão do mesmo.

Os motivos de sua utilização também são melhor explicitados no tópico 3.6 (*Network<T>*), onde se dá o seu uso.

3.11. *ArrayPriorityMinQueue<T>*

A fila de prioridade mínima utiliza um *double* como valor de prioridade, armazenado em um nó criado exclusivamente para as filas de prioridade, o *PriorityQueueNode<T>*, que armazena o conteúdo do nó e o valor da prioridade.

Tal classe herda do *ArrayMinHeap<T>*, em que o tipo genérico é implementado como *PriorityQueueNode<T, priority>* e observa a interface *QueueADT<T>*, ou seja, externamente ela é utilizada como qualquer outra *queue*, com os métodos *enqueue(T, priority)*, *dequeue()* e *first()* implementados na própria e os demais herdados.

A utilização de um *heap* com *array* é melhor explicitada no tópico 3.6 (*Network<T>*), onde se dá o seu uso.

3.12. *ArrayMinHeap<T>*

O *ArrayMinHeap<T>* implementa os métodos *removeMin()* e *findMin()* da interface *HeapMinADT<T>*, e ainda realiza um *override* nos métodos protegidos *heapifyUp(index)* e *heapifyDown(index)*, de modo a comparar e elencar os elementos menores acima do *heap*.

A utilização de um *heap* com *array* é melhor explicitada no tópico 3.6 (*Network<T>*), onde se dá o seu uso.

3.13. *ArrayHeap<T>*

O *ArrayHeap<T>* é uma classe abstrata que implementa a interface *HeapADT<T>* e herda da classe *ArrayBinaryTree<T>*, provendo os métodos comuns a ambos os *heaps* mínimo e máximo.

A utilização de um *heap* com *array* é melhor explicitada no tópico 3.6 (*Network<T>*), onde se dá o seu uso.

3.14. *ArrayBinaryTree<T>*

A classe abstrata *ArrayBinaryTree<T>* implementa a interface *BinaryTreeADT<T>* e provê métodos comuns a todas as árvores binárias (iteradores, receber a raiz etc), utilizando internamente um *array* (*DynamicArrayCircular<T>*) para armazenar os elementos.

A utilização da árvore com *array* é melhor explicitada no tópico 3.6 (*Network*<*T*>), onde se dá o seu uso, ou seja, pela utilização de uma fila de prioridade mínima construída com *array*.

3.15. *DirectedGraph<Person, ViewNode>*

Tal estrutura é utilizada única e exclusivamente para converter a *network* construída no trabalho em um grafo da *API Jung*, para posteriormente poder visualizá-lo em uma janela construída com *swing*.

Portanto, a *SocialNetwork*<*T*> provê os métodos para leitura de sua matriz de adjacência e lista de vértices, aos quais são utilizadas para inserir tais dados no grafo do *Jung*, e posteriormente visualizá-lo.

3.16. *Store*

Os dados dos utilizadores (perfil, conexões etc) são armazenados em uma estrutura chamada *BaseStore*, que agrega como atributos instâncias das classes *SocialNetwork*<*Person*>, *PersonIdOrderedList* e *PersonEmailOrderedList*, em que *Person* é a classe representativa de um utilizador do grafo social.

Qualquer manipulação de dados dos utilizadores do grafo social é realizada por meio da *store*, seja para a leitura de dados, alteração, adição ou remoção. Dessa forma, há maior coesão nas mudanças, tendo em vista que algumas podem implicar em alterações extras nos dados.

Por exemplo, a adição de um utilizador à *store*, por meio do método *addUser(Person)*, implica na checagem se o mesmo já foi adicionado (pelo ID) e, caso contrário, inserção na *SocialNetwork* como vértice e nas listas de utilizadores como um novo elemento.

Da mesma forma, quando um utilizador for adicionado à lista de contatos de outro utilizador, por meio do método *addUserContact(Person1, Person2)* da *store*, é checado se já existe tal relação e, caso contrário, adicionado o segundo utilizador à lista de contatos do primeiro e posterior inserção de uma ligação entre ambos na *SocialNetwork*.

Portanto, a manipulação dos dados dos utilizadores por qualquer componente da aplicação sempre deve passar por métodos da *store*, que retornam o resultado desejado e fazem as mudanças necessárias para a realização de tal ato.

Sendo assim, foi implementado o padrão *singleton* para a *store*, de modo que todos os componentes visuais (criados com *swing*) para gerir a aplicação herdam de um componente base, que detém o método *getStore()* para receber a instância da *store* quando precisar manipulá-la, sempre acessando a mesma instância.

Não obstante, de modo a implementar os requisitos opcionais de mudança do valor das arestas entre os utilizadores (1/visualizações, 1/menções ou 1), foi necessário implementar uma *store* básica contendo métodos comuns e os atributos citados anteriormente, e também o método para acesso da instância em execução da *store*, a depender do tipo de valor que está sendo utilizado na aresta.

Assim, quando o valor da aresta a ser utilizado for 1/visualizações, o método *getInstance()* irá retornar a instância única de *StoreVisualization*, uma classe que herda de *BaseStore* e altera apenas os métodos necessários para cumprir os requisitos da mudança de valor nos *edges*.

Por exemplo, nesse caso, o método da *BaseStore updateUserVisualizations*(*Person, visualizations)* necessita de um *override* pela classe *StoreVisualization* para poder executar a atualização das *edges* dos envolvidos com o novo valor de *visualizations*.

Além disso, ambas as classes filhas (*StoreVisualization*, *StoreMentions* e *StoreNoWeight*) implementam um método a ser chamado na adição de um contato na lista de contatos, criando um *edge* com o valor correto (1/visualizações, 1/menções e 1, respectivamente).

4. Conjunto de Funcionalidades Implementadas

4.1. Carregar um Ficheiro JSON

Como funcionalidade básica para a visualização da rede social, é possível importar um ficheiro *JSON* contendo os utilizadores e seus atributos. Após o carregamento do ficheiro, os dados são convertidos em classes Java por meio da *API* do *Gson*.

Logo mais, as referidas classes são lidas e convertidas em classes próprias da aplicação (os *models*, representando cada recurso, tais como *Person* para um utilizador, *Academic* para a lista de registros acadêmicos etc). Isso ocorrer por meio de uma classe auxiliar *Data*, que utiliza dois *arrays* para mapear a inserção dos utilizadores no grafo e nas listas.

Para fins de teste, foram criadas várias amostras de arquivo *JSON* (estão na base do projeto):

a) *sample00.json*: contém mais de quinze utilizadores, requisito do trabalho prático;
b) *sample01.json*: obtido no enunciado do trabalho prático;
c) *sample02.json:* todos os utilizadores estão conectados entre si, para testar se o grafo é completo;
d) *sample03.json*: um utilizador não está conectado a nenhum outro, para testar se o grafo é conexo;
e) *sample04.json*: existem utilizadores com 0 visualizações, para testar como o grafo se comportará com arestas com peso infinito (1/0);
f) *sample05.json*: o menor caminho entre determinados utilizadores quando o peso for *visualizations* é distinto de quando for *constant*, para testar a mudança de pesos;
g) *sample06.json*: para testar se utilizadores de uma companhia podem alcançar determinado utilizador.

4.2. Visualizar o Grafo

Por meio da utilização da *API* do *Jung*, é possível visualizar o grafo (seus vértices e suas ligações) em uma janela própria.

4.3. Verificar se o Grafo é Completo

É possível verificar se o grafo atual é completo, ou seja, se todos os utilizadores estão conectados com os demais.

4.4. Calcular Melhor Caminho entre Utilizadores

É possível calcular o melhor caminho entre dois utilizadores, por meio da utilização de método já implementado na estrutura de dados abstrata *Network*, que gera uma lista contendo o caminho traçado entre ambos.

O resultado é mostrado com a utilização da *API* do *Jung* para otimizar a sua visualização, de modo que a lista é reconstruída na forma de uma rede, podendo se observar também o custo de cada ligação no caminho.

4.5. Utilizadores alcançáveis por um utilizador

É possível ver quais utilizadores são alcançáveis por um determinado utilizador, por meio da utilização de método já implementado na *Network*, que gera a *minimum spanning tree* a partir de um elemento, ou seja, gera um novo grafo contendo todos os demais utilizadores que podem ser alcançados.

O resultado é mostrado com a utilização da *API* do *Jung* para otimizar a sua visualização como grafo.

4.6. Utilizadores não alcançáveis por um utilizador

É possível ver uma lista contendo todos os utilizadores que determinado utilizador não consegue alcançar na rede, obtendo-se a *minimum spanning tree* do utilizador e removendo-os da lista total de utilizadores.

4.7. Criar um Novo Utilizador

É possível criar um novo utilizador, que iniciará com o próximo *ID* válido (primeiro inteiro seguinte ao último utilizado ou 0 quando não houver pessoas no grafo) e com atributos vazios.

4.8. Gerir o Perfil de um Utilizador

É possível visualizar e alterar o perfil de um utilizador, inclusive adicionar ou remover novos contatos, menções etc.

4.9. Utilizadores de uma Empresa que se Relacionam com Outro Utilizador

É possível pesquisar quais utilizadores que trabalham (ou trabalharam) em determinada empresa estão relacionados com um outro utilizador, ou seja, que conseguem alcançá-lo pelo grafo.

4.10. Utilizadores com Determinada *Skill* Ordenados pelo Menor Custo de Ligação a Partir de Outro Utilizador

É possível listar utilizadores que detenham determinada *skill*, alcançáveis por um determinado utilizador e ordenados pelo custo de ligação, de modo que o menor custo aparecerá primeiro.

4.11. Utilizadores com Determinadas *Skills* e que Trabalharam em Determinada Empresa e são Contato dos Contatos de um Utilizador

É possível listar utilizadores que detenham determinadas *skills* (todas), trabalharam em determinada empresa e são contato dos contatos de um determinado utilizador.

4.12. A Relação entre Utilizadores de Determinado Cargo em uma Empresa com Utilizadores do Mesmo Cargo em Outra Empresa

É possível verificar qual a relação entre utilizadores de determinado cargo em uma empresa e utilizadores do mesmo cargo em outra empresa. A lista resultante mostra quais utilizadores na segunda empresa são relacionados ou nenhum (quando não houver).

4.13. A Média de Menções e Contatos da Rede Social Comparados à Média da Lista de Utilizadores Alcançáveis por Determinado Utilizador (Inclusive)

É possível comparar a média de menções e contatos da rede social com a média de determinado utilizador, considerando-se apenas os utilizadores que são alcançáveis pelo mesmo.

4.14. A Quantidade Mínima de Ligações para Conectar um Utilizador aos outros Utilizadores (Alcançáveis)

É possível calcular qual a quantidade mínima de ligações para conectar um utilizador aos demais utilizadores (que sejam alcançáveis).

O resultado é obtido por meio da utilização da *minimum spanning tree* do utilizador, recebendo o tamanho da mesma (quantidade de vértices) e reduzindo em uma unidade, para encontrar a quantidade de arestas.

4.15. Alterar o Custo da Relação entre Utilizadores

É possível alterar o custo das arestas no grafo, sendo o padrão 1/visualizações do contato e como opção a mudança para 1/menções do contato ou 1 (constante).

Após a mudança do peso das arestas, os dados dos utilizadores são reciclados e aproveitados no novo grafo, de modo a permitir a mudança constante sem perda de dados.

5. Algoritmos Mais Complexos

5.1. Caminho Mais Curto

A estrutura de dados abstrata *Network* detém um método para calcular o caminho mais barato (menos custoso) entre dois utilizadores, o método *iteratorShortestPath(T1, T2)*.

Em síntese, o método recebe como *input* dois utilizadores e retorna um *Iterator* contendo o caminho mais rápido no grafo entre ambos. Caso não seja possível alcançar o destino, o *Iterator* retornado contém 0 elementos.

Tal método é similar ao *iteratorBFS()*, com a diferença que considera o caminho total até cada vizinho (e não apenas o da ligação atual), preferindo por aquele que seja menor (não necessariamente o primeiro que entrou vai ser o primeiro a sair).

O resultado do método utiliza um *UnorderedArrayList<T>* (linha 304) para armazenar os vértices passados no caminho mais rápido (e retornar o seu iterador ao final).

Para auxiliar durante a travessia do grafo, é utilizada uma *ArrayPriorityMinQueue<>* (linha 311), onde serão inseridos os vizinhos do vértice atual na travessia com o seu peso (valor acumulado da peso da origem até o mesmo).

Para tal foi necessário a utilização de três *arrays* do tamanho da quantidade dos vértices:

a) *visited[]*: marca se o vértice já foi visitado na travessia, quando o valor for *true* (linha 314);
b) *pathLength[]*: marca o tamanho acumulado do caminho até determinado vértice (linha 319). Caso o valor seja -1, significa que o vértice não foi visitado ainda;
c) *antecessor[]*: guarda a referência do vértice anterior ao em comento, que pode mudar durante a travessia (linha 324), Caso o valor seja -1, significa que o vértice não foi visitado ainda.

A seguir é descrito o passo-a-passo do algoritmo, após a instanciação das variáveis e classes auxiliares:

1) O vértice da pessoa de origem é inserida na fila de prioridade e o seu valor em *pathLength[]* marcado como 0, pois não tem custo;

2) Inicia-se a recursão para a travessia no grafo;
3) É retirado o elemento de menor peso da fila de prioridade:
    a) Caso ele já tenha sido visitado, inicia-se a recursão novamente, retornando ao passo 2;
4) O vértice é marcado como visitado;
5) Pesquisa-se os vizinhos do vértice que não tenham sido visitados ainda;
    a) Caso não haja vizinhos a visitar e a fila de prioridade tenha elementos, inicia-se a recursão novamente, retornando ao passo 2;
6) É calculado o custo total do caminho até o vizinho, utilizando o custo do seu *edge* entre o vértice atual e o vizinho somado ao valor salvo em *pathLength[]* para o vértice atual (significa o custo total até chegar nele);
7) O vizinho é adicionado à fila de prioridade com o custo acumulado até ele;
8) Caso ainda não exista valor do vizinho em *pathLength[]* ou o custo total atual é menor que o que foi obtido anteriormente (o novo caminho é melhor), atualiza-se o valor em *pathLength[]* e *antecessor[]* para a nova rota;
9) Encerrando-se a recursão (não há mais elementos na fila de prioridade), confere-se o valor do vértice de destino no *array antecessor[]*:
    a) Caso seja -1, significa que não foi possível atingir o vértice, retornando um *Iterator* vazio;
    b) Caso contrário, é adicionada à frente da lista cada elemento a partir do destino, obtendo-se o anterior a partir da referência em *antecessor[]*, até atingir um índice com valor -1, que significa ser a origem do caminho, retornando o iterador.

5.2. Custo do Caminho Mais Curto

Tal algoritmo também está inserido na estrutura de dados abstrata *Network*, com uma pequena variação em comparação ao do caminho mais curto, tendo em vista que a recursão e estruturas auxiliares utilizadas são idênticos.

A diferença reside após a obtenção do caminho até o vértice de destino, em que o presente algoritmo apenas retorna o valor constante em *pathLength[]* para o vértice de destino, ou seja, ali está armazenado o custo total para chegar ao mesmo.

5.3. *Minimum Spanning Tree*

Há também um método em *Network* para calcular a *minimum spanning tree* a partir de um utilizador.

Em síntese, o método recebe como *input* o utilizador de início e retorna uma *Network* contendo todos os utilizadores alcançáveis pelo mesmo com o menor custo total de arestas.

Tal método é similar ao *iteratorBFS()*, com a diferença que o próximo vértice a ser selecionado é sempre o de menor custo atual (considerando-se apenas o vértice atual e o vizinho).

O resultado do método utiliza uma *Network*<*T*> (linha 546) para armazenar os vértices passados no caminho e os seus *edges*.

Para auxiliar durante a travessia do grafo, é utilizada uma *ArrayPriorityMinQueue*<> (linha 550), onde serão inseridos os vizinhos do vértice atual na travessia com o seu peso (considerando apenas o custo do vértice atual para o vizinho).

Para tal foi necessário a utilização de dois *arrays* do tamanho da quantidade dos vértices:

a) *visited[]*: marca se o vértice já foi visitado na travessia, quando o valor for *true* (linha 314);
b) *antecessor[]*: guarda a referência do vértice anterior ao em comento, para poder adicionar as *edges* entre ambos quando for necessário.

A seguir é descrito o passo-a-passo do algoritmo, após a instanciação das variáveis e classes auxiliares:

1) O vértice da pessoa de origem é inserida na fila de prioridade;
2) Inicia-se a recursão para a travessia no grafo;
3) É retirado o elemento de menor peso da fila de prioridade:
    a) Caso ele já tenha sido visitado, inicia-se a recursão novamente, retornando ao passo 2;
4) O vértice é marcado como visitado;
5) Adiciona-se o vértice ao resultado (grafo);
6) Caso o vértice tenha um antecessor (*antecesso[]*), adiciona uma *edge* no grafo entre ambos;
7) Pesquisa-se os vizinhos do vértice que não tenham sido visitados ainda;

a) Caso não haja vizinhos a visitar e a fila de prioridade tenha elementos, inicia-se a recursão novamente, retornando ao passo 2;

8) O vizinho é adicionado à fila de prioridade com o custo do vértice atual até ele;
9) Atualiza-se o valor do *antecessor[]* do vizinho para o vértice atual;
10) Encerrando-se a recursão (não há mais elementos na fila de prioridade), retorna-se o grafo recém construído.

5.4. Populando a *Store* com o *JSON*

A classe *Data* (ed.trabalho.helpers.*Data*) contém o método *populate()* que serve para popular a *store* (grafo e listas) com os dados recebidos pelo *JSON*.

Inicialmente, a *store* é limpa (grafo e listas são limpos), criando-se um *array* temporário de *Person* para contrastar com o array de *Pessoa* recebidos do *JSON*, de modo a manter o mesmo índice em ambos os *arrays*.

Prossegue-se à criação de utilizadores (*Person*) com base em cada *Pessoa*, com os atributos primários e não relacionais a outros utilizadores (id, nome, email, lista de formação acadêmica etc).

Posteriormente, são adicionadas as conexões entre utilizadores *Person*, sejam elas do tipo contato ou menção. Para tanto, o utilizador armazena em sua lista de contatos/menções as referências aos demais utilizadores, de modo a facilitar o fluxo para obter um ou outro em qualquer sentido.

Nesse intuito, após a criação de um utilizador com os dados primários, são adicionadas as relações com outros utilizadores e, caso o utilizador a ser adicionado ainda não exista, ele é criado previamente (apenas com os atributos primários e posteriormente implementa-se as relações).

5.5. Copiar o Conteúdo dos Utilizadores para *Stores* Distintas

Após a implementação de três espécies de *stores*, uma para cada requisito de peso de aresta diferente, é possível mudar o tipo de *store* atual no menu da aplicação.

Para tanto, implementou-se um algoritmo que copia os dados dos utilizadores entre as *stores*, tanto as listas como o grafo, apenas alterando os valores contidos nas arestas.

Inicialmente, é selecionada a nova *store* (por meio do padrão *singleton*) e limpos todos os dados da mesma.

Logo mais, são copiadas as referências das listas e do grafo da *store* antiga para a nova e alterados os valores dos pesos da aresta por meio do acesso direto à matriz de adjacências e vértices de ambas.

6. Conclusão

Este trabalho permitiu aumentar o conhecimento acerca das Estruturas de Dados e como usá-las de forma eficaz e eficiente, especialmente com o aprendizado de como implementá-las com a linguagem java.

Foi possível vivenciar a criação de uma estrutura do inicio, desde o seu planeamento até à sua implementação, e justificação da razão da escolha de determinada estrutura, tendo em vista as implicações que a mesma terá no desempenho do algoritmo.

Com isso, aprendeu-se a analisar criticamente um problema e solucioná-lo com o uso de ferramentas corretas para o caso concreto, avaliando-se as implicações das complexidades algorítmicas temporal e espacial das implementações realizadas e decidindo por uma solução que seja razoável e ótima para a sua execução.

7. Referências

Conteúdo Disponibilizado para as Aulas de Estruturas de Dados no Moodle do IPP. Disponível em: <https://moodle.estg.ipp.pt/login/index.php>. Acesso em: 20 dez. 2019.

Guava Release 23.0. Disponível em: <https://github.com/google/guava/wiki/Release23>. Acesso em: 16 jan. 2019.

Java serialization/deserialization library to convert Java Objects into JSON and back. Disponível em: <https://github.com/google/gson>. Acesso em: 13 jan. 2019.

Java Singleton Design Pattern Best Practices with Examples. Disponível em: <https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples>. Acesso em: 20 jan. 2019.

Java Swing Tutorial. Disponível em: <https://www.tutorialspoint.com/swing/>. Acesso em: 15 jan. 2019.

JUNG: Java Universal Network/Graph Framework. Disponível em: <https://github.com/jrtom/jung>. Acesso em: 16 jan. 2019.

SWING Tutorial. Disponível em: <https://www.tutorialspoint.com/swing/>. Acesso em: 15 jan. 2019.

The Graph Abstract Data Type - Chapter 13. Disponível em: <https://slcc.instructure.com/courses/232958/files/30982396/download>. Acesso em: 22 dez. 2019.