Objetivos

- Desenvolver a habilidade de implementar classes genéricas e métodos genéricos.
- Praticar o tratamento de erros via exceções.
- Praticar os conceitos com problemas de estruturas de dados.

Conceitos básicos

O paradigma de orientação a objetos é muito utilizado para a implementação de estruturas de dados. Uma das vantagens do emprego desse paradigma é a boa prática de desenvolver estruturas como Tipos Abstratos de Dados (TADs) por meio das abstrações de encapsulamento e de composição (analogamente ao que fizemos, em aula, com arrays para implementar pilhas). Em geral, começamos com estruturas elementares (e.g. array, nó, lista, heap, etc.), a partir das quais projetamos estruturas mais complexas. Isso permite prover APIs bem-definidas para as estruturas, ao mesmo tempo em que se "oculta" os detalhes da implementação.

No nível conceitual, grande parte das estruturas de dados são agnósticas à tipagem dos dados, porém nos níveis lógico e físico, principalmente quando utilizamos linguagens de tipagem estática (e.g. Java), tal questão precisa ser abordada. Nessas linguagens, o polimorfismo paramétrico (denominado *Generics* em Java), mostra-se valioso para instanciar uma mesma estrutura para diferentes tipos de dados. Tal técnica se contrapõe e traz vantagens em relações a técnicas mais limitadas, que diminuiriam a manutenibilidade (por exemplo, criando uma versão da estrutura para cada tipo de dados) ou comprometeriam a robustez do sistema de tipos (por exemplo, cainda nas "armadilhas" do polimorfismo por subtipos).

Estruturas de dados frequentemente necessitam sinalizar situações de erro de modo não-trivial. Na ausência de recursos para tratamento de exceções, o tratamento de erros costuma trazer recursos indesejáveis, como valores de status no retorno dos métodos (quando possível), parâmetros como referência para sinalizar erro ou a inclusão de "prints" no corpo dos métodos. Tais alternativas diminuem a qualidade da estrutura, pois tornam os métodos em funções impuras[1], as quais dificultam, entre outros fatores, a verificação de corretude via testes de unidade (*unit testing*) e a programação concorrente (*thread-safety*).

Nesta atividade, você praticará o desenvolvimento de estruturas de dados utilizando boas práticas de programação genérica e de tratamento de erros. Em particular, partiremos da implementação prévia de uma lista encadeada para implementarmos uma pilha, uma fila e uma deque. Complementarmente, teremos também um algoritmo para testes em arrays genéricos.

Conceito: lista encadeada

Lista encadeada é uma estrutura de dados que permite armazenar informações de modo não-sequencial na memória. Uma lista encadeada é composta por nós, conectados por meio de ponteiros. Há diferentes modos de representar o encadeamento, dentre os quais citamos o encadeamento duplo não-circular, no qual cada nó possui referências para seu antecessor e para seu sucessor, exceto para o primeiro nó (que não possui antecessor) e para o último nó (que não possui sucessor). A figura 1 ilustra uma lista duplamente encadeada de caracteres.

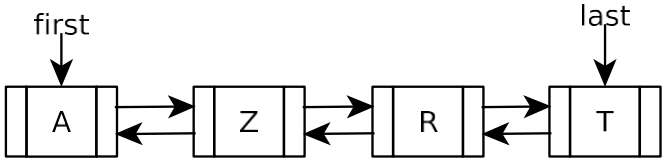

Figura 1: Lista duplamente encadeada de caracteres

[1] https://en.wikipedia.org/wiki/Pure_function#Impure_functions

Nessa estrutura, temos acesso direto apenas aos ponteiros de início (`first`) e fim (`last`) da lista, todas as operações devem partir de um desses pontos. Portanto, as operações no início e no fim da lista são menos custosas (tempo constante) do que as operações nos elementos intermediários (tempo variável).

Em geral, utilizamos listas encadeadas como base para implementar estruturas de dados mais sofisticadas, como listas ordenadas, pilhas, filas, árvores, grafos, etc. As listas encadeadas são um recurso importante para a implementação de estruturas de dados dinâmicas, ou seja, estruturas que alocam ou desalocam memória conforme a necessidade.

Conceito: Pilha

O conceito de pilha implementa a lógica LIFO (*Last In First Out*), já abordado nas video-aulas, inclusive com implementações. Garanta que tenha familiaridade com os conceitos e técnicas dessa estrutura de dados antes de avançar na atividade.

Conceito: Fila

Fila[2] é uma estrutura de dados cujas operações seguem a lógica FIFO (*First In First Out*), ou "o primeiro que entra é o primeiro que sai". O conceito de fila é bastante intuitivo, pois corresponde ao conceito de fila que adotamos no cotidiano:

- os elementos são organizados numa sequência;
- novos elementos entram no final da fila;
- só é possível retirar elementos da frente/início da fila.

Para reforçar o conceito de fila e a lógica FIFO, interaja com essa animação: [https://yongdanielliang.github.io/animation/web/Queue.html]. Adicione e remova elementos para entender como a estrutura se comporta.

Conceito: Deque

A estrutura de dados Deque[3] (acrônimo para *double-ended queue*, ou fila com duas terminações) é uma generalização de fila que permite inserir e remover elementos tanto no final quanto no início. Tal flexibilidade permite à deque intercambiar tanto a lógica LIFO quanto a lógica FIFO, ou seja, pode atuar tanto como fila quanto como pilha.

Para reforçar o conceito de Deque, interaja com essa animação: [https://visualgo.net/en/list]. Escolha "DEQUE" no menu. Interaja adicionando e removendo elementos, no início e no final, para entender a lógica da estrutura.

Questão 1. Classes genéricas

A Figura 2 contém o diagrama de classes para esta questão. Você deve implementar as classes em azul. As classes e interfaces em preto fazem parte do código-base e já estão implementadas. Note que a linha contínua representa um relacionamento de associação[4], significando, em linhas gerais, "um objeto dentro do outro".

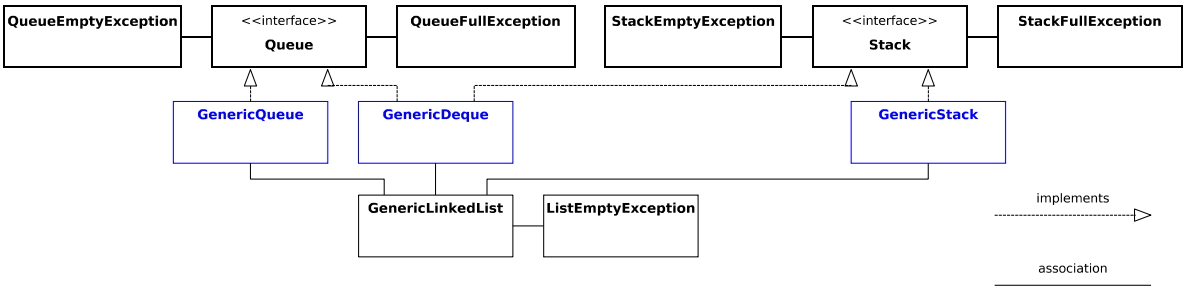

Figura 2: Diagrama de classes UML para a Questão 1

Requisitos adicionais

- Todas as classes devem ter um construtor que recebe um inteiro, correspondente à capacidade da estrutura;
- A capacidade é um inteiro positivo ou zero, valores inválidos devem ser rejeitados via `IllegalArgumentException`.

Código-base

Algumas classes e interfaces já estão prontas para você reusar. **Atenção**: *reusar* significa que não é permitido alterar uma linha sequer nesses componentes. Consulte os JavaDocs das classes e das interafaces para entender o que faz cada um dos seus métodos.

A classe `GenericLinkedList`

Implementação de uma lista duplamente encadeada genérica. Você deve reusá-la, por meio da técnica de composição, para implementar as outras estruturas.

As interfaces `Stack` e `Queue`

APIs contendo as operações clássicas de uma pilha (lógica LIFO) e de uma fila (lógica FIFO), respectivamente. Devem ser implementadas pelas estruturas que aderem às respectivas lógicas.

As exceções customizadas

As interfaces `Queue` e `Stack`, bem como a classe `GenericLinkedList`, dependem de exceções customizadas, todas checadas (*checked*):

- `ListEmptyException`
- `StackEmptyException`
- `StackFullException`
- `QueueEmptyException`
- `QueueFullException`

Nas interfaces, os métodos que disparam exceções estão indicados via `throws`.

Questão 2. Método genérico para teste de ordenação

Código-base: ArraysUtil

Durante a aula, desenvolvemos uma biblioteca de métodos estáticos para processar arrays genéricos. Desenvolvemos, por exemplo, métodos de ordenação e de comparação para arrays genéricos. Você implementará um método adicional nessa biblioteca.

Você deve incluir a classe `ArraysUtil` no projeto e depois implementar o método solicitado.

Especificação da API

- Classe `ArraysUtil`, método estático **isSorted**

  Descrição: verifica se um array genérico está ordenado em ordem crescente

  Parâmetros:

  – `elementos`: array de tipo genérico

  Retorno: `true` caso o array esteja ordenado, `false` caso contrário

[2] https://en.wikipedia.org/wiki/Queue_(abstract_data_type)
[3] https://en.wikipedia.org/wiki/Double-ended_queue
[4] https://en.wikipedia.org/wiki/Object_composition