	Universidade do Estado do Rio de Janeiro	Período: 2025.1
	Algoritmos e Estruturas de Dados	Turma 1
	Professor Rodrigo Mafort	02/07/2025

Segundo Trabalho

O objetivo principal desse trabalho é a aplicação prática dos quatro pilares da Programação Orientada a Objetos (Abstração, Encapsulamento, Herança e Polimorfismo) no projeto e implementação de um conjunto de estruturas de dados clássicas em C++, criando um sistema de classes coeso, reutilizável e robusto, seguindo as melhores práticas do orientação a objeto usando o C++ como linguagem de programação.

Parte I: Implementações com Alocação Sequencial

Você deverá criará uma hierarquia de classes para os elementos que serão armazenados nas estruturas. Para isso, implemente uma classe abstrata chamada "Elemento". Essa classe será usada como ancestral para todos os elementos que serão inseridos nas estruturas de dados.

Ela deve conter um atributo protegido chamado ID (do tipo int ou long), que servirá como chave primária. O acesso externo a este atributo deve ser feito exclusivamente através de um método público getID() const.

A classe deve ser abstrata, impedindo a instanciação de objetos. Adicione também pelo menos uma outra função virtual pura, como virtual void imprimirInfo() const = 0;, que forçará todas as classes concretas a implementar uma forma de se "apresentar".

Classes que herdam de "Elemento": Crie pelo menos duas classes instanciáveis que herdem de Elemento. Cada classe deve implementar todos os métodos virtuais puros herdados de Elemento e possuir seus próprios atributos e métodos específicos. Essas classes deverão ser usadas como demonstração das estruturas criadas nessa primeira parte do trabalho.

Após implementar os "elementos", defina as estruturas de dados. Elas não devem ser vistas como programas isolados, mas como classes de "contêineres" projetadas para gerenciar objetos do tipo "Elemento". Todas as estruturas de dados deverão gerenciar a memória dos objetos Elemento que armazenam.

Considerando os fundamentos discutidos em sala em relação à alocação sequencial de memória, implemente:

Lista Não Ordenada: Implemente os métodos InserirNoInicio, InserirNoFinal, RemoverPrimeiro, RemoverUltimo, RemoverPeloId, BuscarPeloId e AlterarPeloId. Analise e comente a complexidade de cada operação;

Lista Ordenada: Implemente os mesmos métodos da lista não ordenada. Contudo, as operações deve manter a ordem dos elementos pelo ID. A busca deve ser otimizada, utilizando busca binária para alcançar complexidade $O(logn)$;

Pilha: Implemente as classes Pilha utilizando composição. Em vez de reimplementar a lógica, sua classe Pilha deve ter um atributo privado que seja uma das listas que você já implementou (escolha a estrutura apropriada). Exponha apenas os métodos para empilhar, desempilhar, consultarTopo, pilhaCheia e pilhaVazia. Mapeie essas operações para as funcionalidades da sua lista subjacente;

Fila: Implemente uma fila seguindo os mesmos fundamentos apresentados para as pilhas. Observe que essa estrutura apresentará uma manipulação menos eficiente do que o necessário (estude os métodos de manipulação). Proponha uma segunda implementação que corrija o problema encontrado.

Parte II: Implementações com Alocação Encadeada:

Encapsulamento do Nó: Para cada estrutura, a definição dos "nós" deve ser uma classe privada dentro da própria classe da estrutura de dados, escondendo completamente este detalhe de implementação do usuário final.

Implemente as classes ListaSimplesmenteEncadeada, ListaDuplamenteEncadeada e ListaDuplamenteEncadeadaCircular robustas e com todas as operações de manipulação (inserirNoInicio, inserirNoFim, removerPeloId, buscarPeloId, etc.).

Assim como na parte sequencial, implemente Pilha, Fila e Deque como "adaptadores" que utilizam, por composição, uma instância de uma de suas classes para listas encadeadas. Escolha a estrutura mais adequada para reduzir ao máximo a complexidade computacional das operações de manipulação.

Implemente também a classe ArvoreBinariaBusca. Assim como realizado anteriormente, esconda completamente o uso de uma classe para os nós da árvore do usuário final. Mantenha a propriedade fundamental das árvores binárias de busca baseada no atributo ID do nó.

Implemente os métodos Inserir, BuscarPeloId e RemoverPeloId. Implemente também os três tipos de percurso: emOrdem, preOrdem e posOrdem. Os percursos devem imprimir o atributo ID de cada nó.

Parte III: Requisitos de Avaliação e Demonstração dos Princípios

Um programa main.cpp deve ser criado para servir como base de testes e demonstração. Para as estruturas baseadas em alocação estática, ele deve instanciar objetos de suas diferentes classes derivadas de Elemento e adicioná-los a uma mesma estrutura de dados. Depois, percorrer a estrutura e invocar o método imprimirInfo() de cada elemento, demonstrando que a versão correta do método é chamada para cada tipo de objeto. Observe que isso implica que uma mesma lista deverá conter objetos de diferentes classes. Para as estruturas baseadas em alocação dinâmica, basta demonstrar que as estruturas apresentam o comportamento esperado.

O código do arquivo main.cpp só pode interagir com as estruturas implementadas através de seus métodos públicos. Qualquer tentativa de acessar membros internos deve resultar em erro de compilação.

Código Limpo e Moderno: Seu código deve ser bem comentado e ter uma organização clara de arquivos (.h para declarações, .cpp para implementações).

Análise de Complexidade: A implementação de cada método de manipulação nas classes de estrutura de dados deve ser acompanhada de um comentário indicando sua complexidade de tempo computacional.

Entregáveis: Todos os arquivos de código-fonte (.h e .cpp), o arquivo main.cpp de demonstração e um arquivo README.md explicando as principais decisões de design tomadas (ex: "Optei por composição para implementar a Pilha para maximizar o reuso de código da classe Lista...") e uma tabela resumindo a complexidade de todas as operações implementadas. O arquivo README.md deverá listar TODOS os integrantes do grupo.

Forma de Entrega: Um membro do grupo deverá indicar um repositório público no GitHub. Esse repositório deverá conter os arquivos indicados anteriormente. Caso o link postado seja inválido ou o repositório seja privado, será atribuído valor zero ao trabalho. Serão considerados apenas *commits* realizados na branch *main* (ou *master*, na ausência da anterior) até a data anterior a entrega do trabalho.