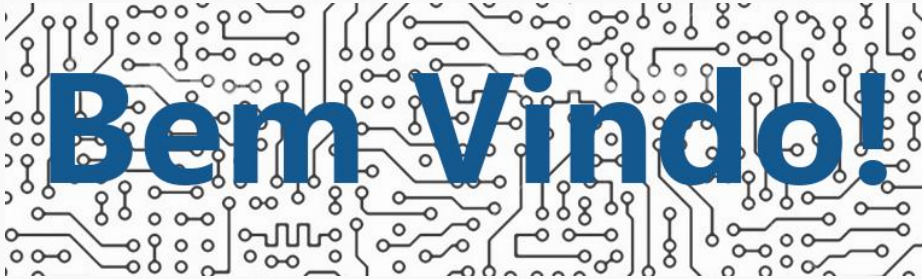


UNISINOS

DESAFIE
O AMANHÃ.

Estrutura de Dados

Vinicius Bischoff

Formação Acadêmica

PUCRS

INSTITUTO IVOTI
Schola semper reformanda

FACCAT

UNIVERSIDADE FEDERAL DE PERNAMBUCO

UNISINOS
DESAFIE O AMANHÃ.

Graduação

- Engenharia
- Pedagogia
- Analise e Desenvolvimento de Sistemas

Pós-Graduação

- Testes de Software
- Computação Aplicada
- Especialização\Mestrado\Doutorado

Atividades Profissionais

- Estagiário
- Operador de Teleprocessamento
- Analista de Sistemas
- Gerente de Projetos
- Desenvolvedor
- Engenheiro de Software
- Pesquisador\Professor

ALUNOS | Apresentações

- Moodle

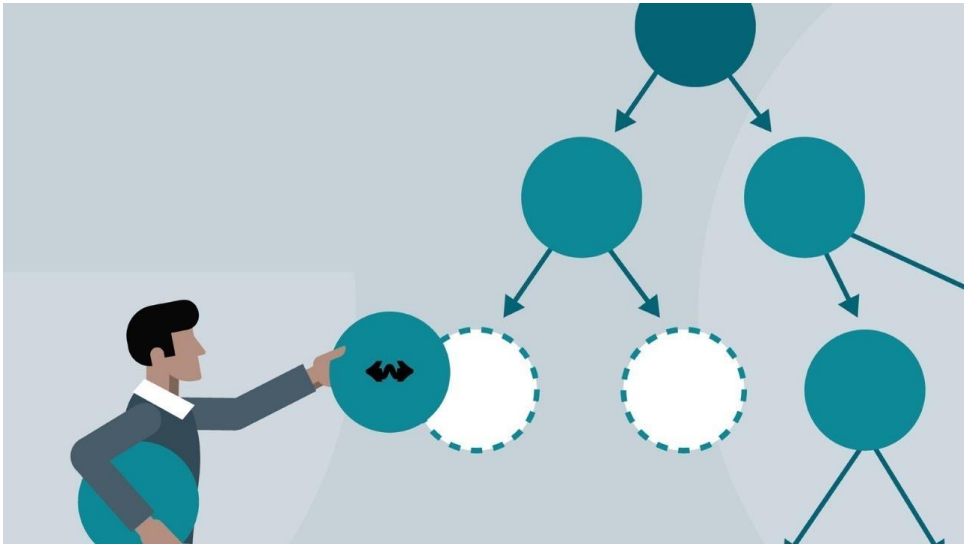
UNISINOS
DESAFIE
O AMANHÃ.

Aula 1

.

UNISINOS
DESAFIE
O AMANHÃ.

O que é uma Estrutura de Dados?



Estrutura de dados é uma maneira organizada de armazenar, gerenciar e acessar dados de forma eficiente.

Elas são fundamentais para a implementação de algoritmos, pois determinam a maneira como os dados são organizados na memória de um computador.

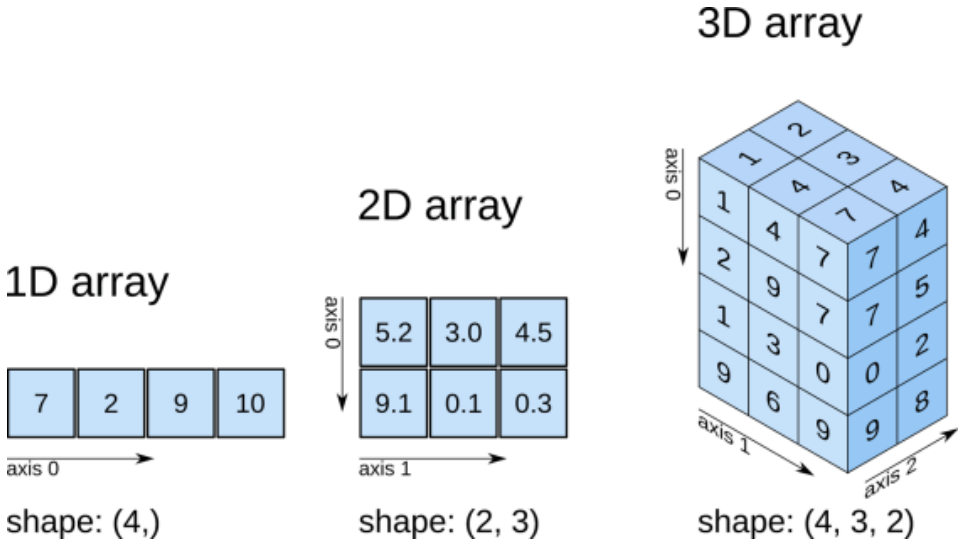
Estruturas de dados definem não apenas o layout físico dos dados, mas também as operações que podem ser realizadas sobre eles, como inserção, remoção, busca, e ordenação.

Nesta aula, vamos explorar a importância das estruturas de dados, focando em como elas influenciam o desempenho de sistemas de software.

Em seguida, vamos introduzir os Tipos Abstratos de Dados (TAD), que oferecem uma maneira de organizar dados de forma abstrata, permitindo a construção de algoritmos mais eficientes e modulares

Tipos Comuns de Estruturas de Dados

Arrays (Vetores): Coleções de elementos ordenados que podem ser acessados por um índice.

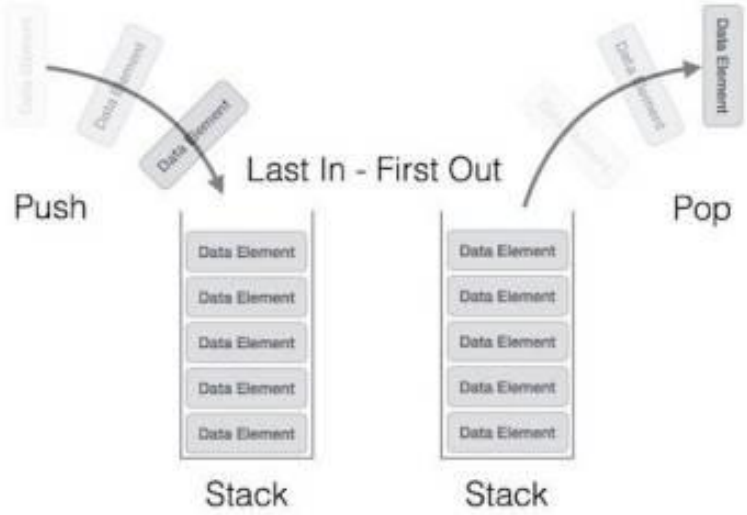


Tipos Comuns de Estruturas de Dados

Listas: Sequências de elementos, que podem ser ligadas (onde cada elemento aponta para o próximo) ou dinâmicas (como em listas encadeadas).

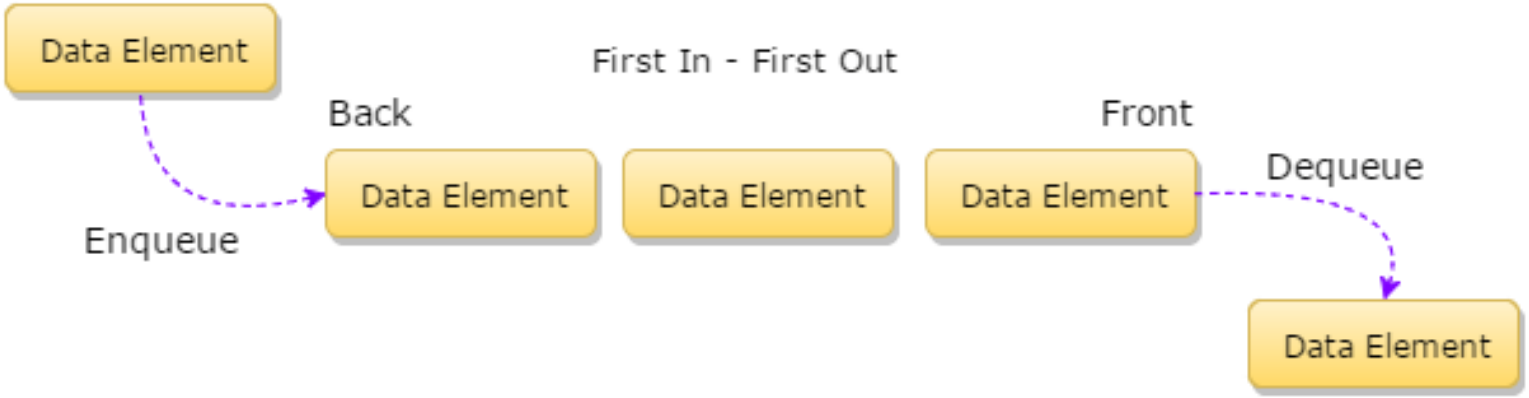
Tipos Comuns de Estruturas de Dados

Pilha (Stack): Estrutura LIFO (Last In, First Out), onde o último elemento inserido é o primeiro a ser removido.



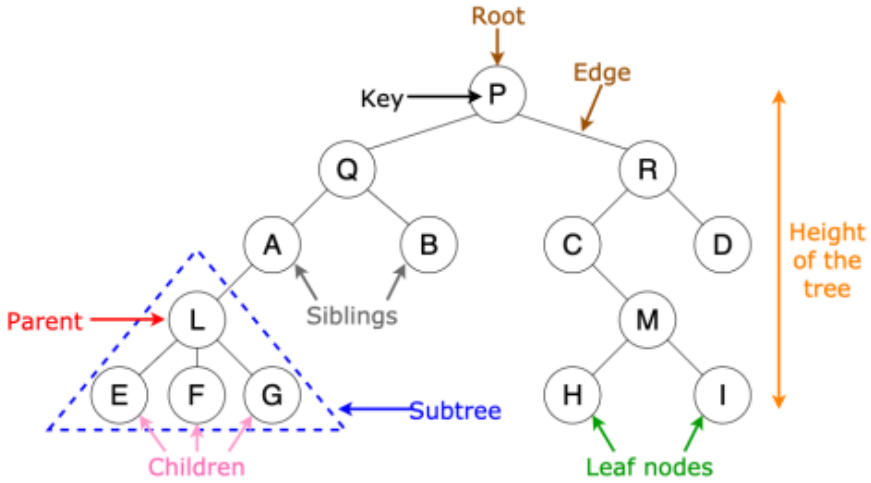
Tipos Comuns de Estruturas de Dados

Fila (Queue): Estrutura FIFO (First In, First Out), onde o primeiro elemento inserido é o primeiro a ser removido.



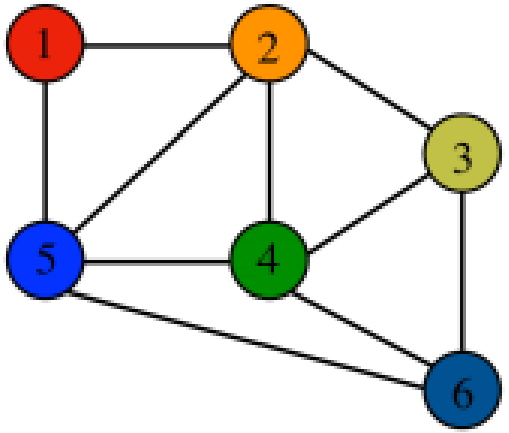
Tipos Comuns de Estruturas de Dados

Árvores: Estruturas hierárquicas que armazenam elementos em "nós", com um nó raiz e nós filhos.

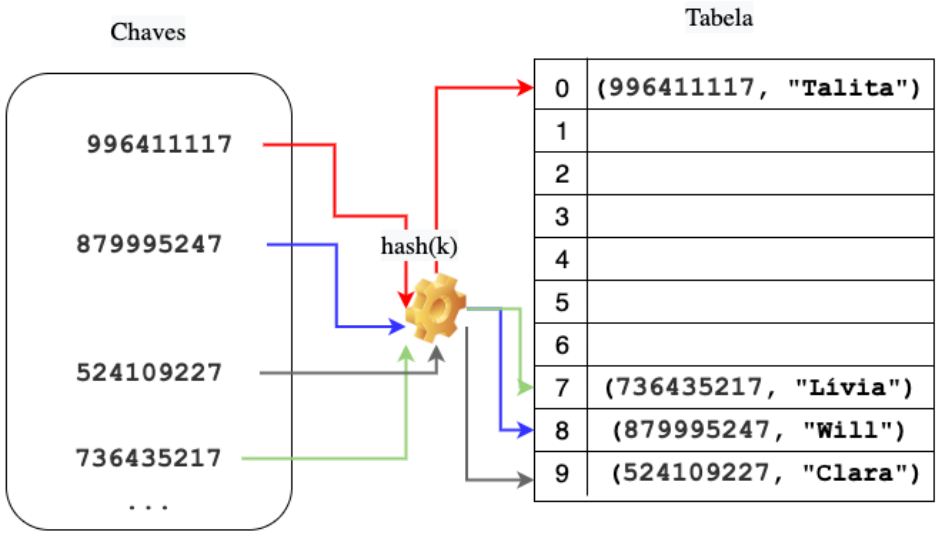


Tipos Comuns de Estruturas de Dados

Grafos: Conjuntos de nós conectados por arestas, utilizados para representar redes e relações complexas.



Tipos Comuns de Estruturas de Dados



* hash(k) = ((k % tabela.length) + sondagem) % tabela.length
* sondagem pertence à {0...tabela.length - 1}

Tabelas Hash: Estruturas que mapeiam chaves a valores, permitindo buscas rápidas.

Para que Servem as Estruturas de Dados na Prática?

Na prática, as estruturas de dados são essenciais para o desenvolvimento de software eficiente.

Elas desempenham um papel crucial em diversas áreas da computação.

1. Otimização de Desempenho:

- As estruturas de dados permitem que operações comuns como busca, inserção e remoção sejam realizadas de maneira eficiente. Por exemplo, uma Tabela Hash pode buscar um item em tempo constante (O(1)), enquanto uma lista pode levar tempo linear (O(n)) para encontrar um item.

- Em grandes volumes de dados, a escolha da estrutura de dados correta pode reduzir significativamente o tempo de execução e o uso de recursos.

2 . Gestão de Memória:

- Estruturas de dados ajudam a gerenciar como a memória é utilizada. Algumas estruturas, como listas ligadas, são dinâmicas e podem crescer ou diminuir conforme necessário, enquanto outras, como arrays, têm tamanho fixo e são mais estáticas.

- A escolha da estrutura de dados certa pode minimizar o desperdício de memória e melhorar a eficiência

3. Resolução de Problemas Complexos:

- Estruturas como árvores e grafos são usadas para representar problemas complexos em muitas áreas, incluindo inteligência artificial, redes de computadores e bancos de dados.

- Por exemplo, uma árvore binária de busca é uma estrutura eficiente para ordenar e buscar dados hierárquicos.

4. Organização e Manipulação de Dados:

- Estruturas de dados ajudam a organizar dados de maneira que seja fácil realizar operações específicas. Por exemplo, uma fila de prioridade pode ser usada em algoritmos de escalonamento, onde tarefas com maior prioridade são processadas primeiro.

- Elas também facilitam a implementação de algoritmos, pois fornecem um meio estruturado para armazenar e acessar dados.

5. Implementação de Algoritmos:

- Muitos algoritmos dependem de estruturas de dados específicas para funcionar corretamente. Por exemplo, o algoritmo de Dijkstra para encontrar o caminho mais curto em um grafo usa uma fila de prioridade.

- A escolha da estrutura de dados influencia diretamente a complexidade e a eficiência de um algoritmo.

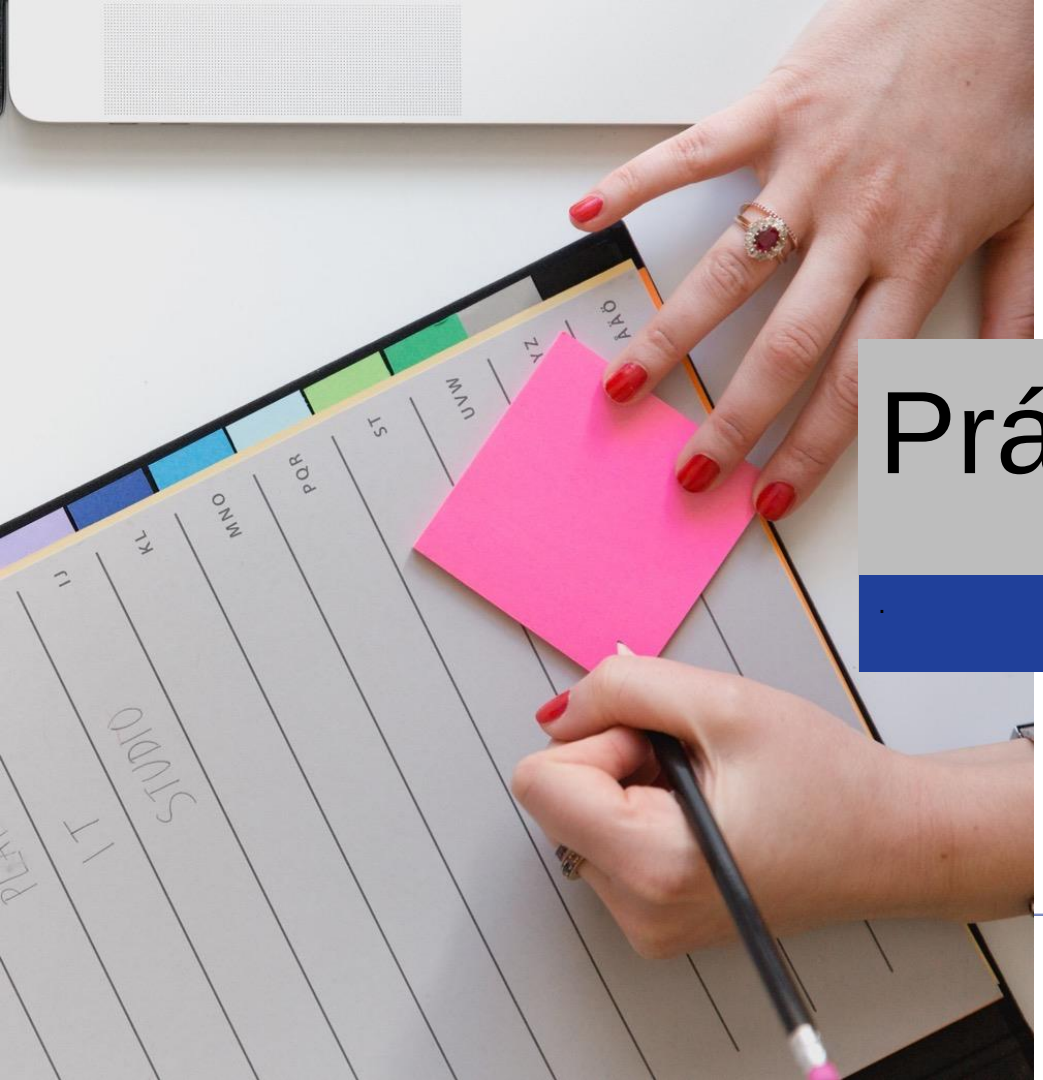
Exemplos Práticos:

•**Desenvolvimento de Jogos:** Usar uma estrutura de dados como uma árvore para representar diferentes estados de um jogo e tomar decisões baseadas nesses estados.

•**Aplicações Web:** Utilizar listas e tabelas hash para armazenar e buscar rapidamente informações de usuários.

•**Redes Sociais:** Grafos para modelar e analisar as relações entre usuários, como amizades ou seguidores.

•**Sistemas de Gerenciamento de Banco de Dados:** Árvores B+ para organizar e buscar registros de maneira eficiente.



Prática

UNISINOS
DESAFIE
O AMANHÃ.

Parte 3: Discussão em Grupo sobre Expectativas da Disciplina

Atividade de Discussão

- Moodle

Objetivo da Discussão:

- Alinhar as expectativas dos alunos com os objetivos da disciplina.
- Entender quais tópicos os alunos consideram mais relevantes para suas carreiras futuras.
- Coletar feedback sobre as primeiras impressões dos alunos sobre o curso.

Método:

- Dividir a turma em pequenos grupos (3 estudantes).
- Cada grupo discute suas expectativas e as anota em um quadro.
- Um representante de cada grupo compartilha as conclusões com a turma

Parte 2: Importância das Estruturas de Dados no Desenvolvimento de Software

Discussão Teórica

- Moodle

UNISINOS
DESAFIE O AMANHÃ.

Por que as Estruturas de Dados são Importantes?

- As estruturas de dados são a espinha dorsal dos algoritmos. Elas permitem que os dados sejam organizados e acessados de forma eficiente, impactando diretamente a performance do software.

- Em aplicações do mundo real, como sistemas de gerenciamento de bancos de dados, motores de busca e redes sociais, as estruturas de dados são usadas para garantir que as operações de busca, inserção e remoção de dados sejam executadas de forma rápida e eficiente.

Impacto no Desempenho

- Estruturas de dados como arrays, listas, filas, pilhas, árvores, grafos, e tabelas hash têm características específicas que as tornam mais ou menos adequadas para diferentes tipos de problemas.

- O uso incorreto de uma estrutura de dados pode levar a ineficiências significativas, como maior tempo de execução ou consumo excessivo de memória, o que pode comprometer a usabilidade e a escalabilidade do software.

ComparacaoListas.java

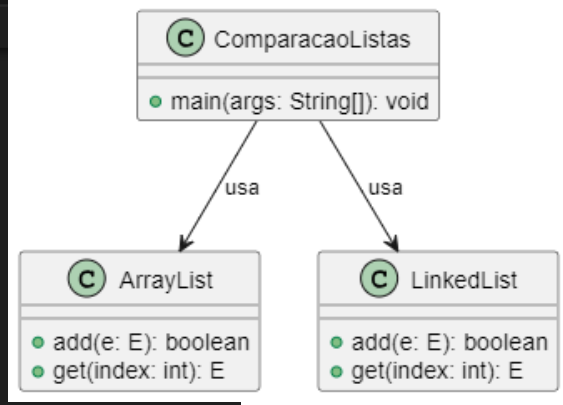
```java

public class ComparacaoListas {

    public static void main(String[] args) {
        // Criando uma instância de ArrayList para armazenar inteiros
        ArrayList<Integer> arrayList = new ArrayList<>();
        // Criando uma instância de LinkedList para armazenar inteiros
        LinkedList<Integer> linkedList = new LinkedList<>();

        // Medindo o tempo de inserção no ArrayList
        long tempoInicio = System.nanoTime();  // Captura o tempo inicial em nanossegundos
        for (int i = 0; i < 100000; i++) {  // Loop para adicionar 100.000 elementos no ArrayList
            arrayList.add(i);  // Adiciona o elemento i ao final do ArrayList
        }
        long tempoFim = System.nanoTime();  // Captura o tempo final após a inserção
        // Calcula e imprime o tempo total gasto na inserção no ArrayList
        System.out.println("Tempo de inserção no ArrayList: " + (tempoFim - tempoInicio) + " ns");

        // Medindo o tempo de inserção no LinkedList
        tempoInicio = System.nanoTime();  // Captura o tempo inicial em nanossegundos novamente
        for (int i = 0; i < 100000; i++) {  // Loop para adicionar 100.000 elementos no LinkedList
            linkedList.add(i);  // Adiciona o elemento i ao final do LinkedList
        }
        tempoFim = System.nanoTime();  // Captura o tempo final após a inserção
        // Calcula e imprime o tempo total gasto na inserção no LinkedList
        System.out.println("Tempo de inserção no LinkedList: " + (tempoFim - tempoInicio) + " ns");
    }
}
```



UNISINOS
DESAFIE O AMANHÃ.

Definição e Conceitos Fundamentais

O que são Tipos Abstratos de Dados (TAD)?

- Um TAD é uma abstração que define um tipo de dado de alto nível e as operações que podem ser realizadas sobre ele, sem expor detalhes de implementação. Essa abstração permite que algoritmos sejam escritos de maneira mais genérica e modular.

- Os TADs fornecem uma interface para operações como inserção, remoção, busca e outras manipulações de dados, enquanto a implementação concreta pode variar.

Exemplos Comuns de TADs

- **Pilha (Stack):** LIFO (Last In, First Out). Exemplo de uso: Desfazer operações em um editor de texto.
- **Fila (Queue):** FIFO (First In, First Out). Exemplo de uso: Gerenciamento de tarefas em filas de impressão.
- **Lista:** Sequência ordenada de elementos, com acesso a qualquer elemento pelo índice.
- **Conjunto (Set):** Coleção de elementos únicos, sem ordem específica.

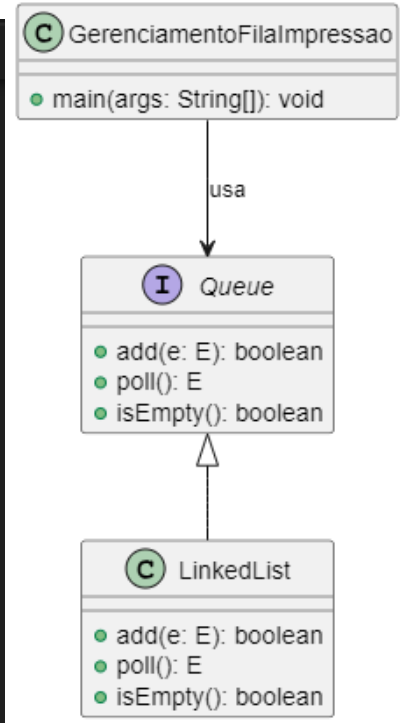
GerenciamentoFilaImpressao.java > ...

```java
public class GerenciamentoFilaImpressao {

    Run | Debug
    public static void main(String[] args) {
        // Criação de uma fila de impressão usando LinkedList
        Queue<String> filaImpressao = new LinkedList<>();

        // Adicionando tarefas de impressão à fila (Enfileiramento)
        filaImpressao.add(e:"Documento1.pdf");
        filaImpressao.add(e:"Documento2.pdf");
        filaImpressao.add(e:"Documento3.pdf");

        // Processando a fila de impressão (Desenfileiramento)
        while (!filaImpressao.isEmpty()) {
            String documento = filaImpressao.poll();  // Retira o primeiro elemento da fila
            System.out.println("Imprimindo: " + documento);
        }
    }
}
```



Ao comparar a complexidade de tempo de inserção entre um **ArrayList** e uma **LinkedList**, estamos interessados no comportamento de **Big-O** (ou Notação Assintótica) para medir como o tempo de execução aumenta conforme o tamanho da entrada aumenta. Vejamos as diferenças:

1. ArrayList:

- **Complexidade de Inserção**: No pior caso, a inserção no final do **ArrayList** tem complexidade **O(n)**. Isso ocorre porque, quando o array atinge sua capacidade máxima, ele precisa ser redimensionado. O redimensionamento envolve copiar todos os elementos para um novo array de tamanho maior, o que leva tempo proporcional ao número de elementos, ou seja, **O(n)**.

- **Inserção em posições específicas** (não no final): Inserir um elemento no meio ou início do array pode também exigir o deslocamento de todos os elementos subsequentes, o que resulta em complexidade **O(n)**.

2. LinkedList:

- **Complexidade de Inserção**: Inserir no final de uma **LinkedList** é **O(1)**, pois envolve apenas a alteração de um ponteiro para o novo nó, independentemente do tamanho da lista.

No entanto, a inserção em uma posição intermediária ou no início exige percorrer a lista até encontrar a posição correta, o que tem complexidade **O(n)** no pior caso (quando você precisa percorrer toda a lista).

Resumo Comparativo:

- **ArrayList**: **O(n)** para inserção no pior caso devido à necessidade de redimensionamento.

- **LinkedList**: **O(1)** para inserção no final (simplesmente ajusta ponteiros), mas **O(n)** para inserção em qualquer outra posição, pois é necessário percorrer os elementos.

Conclusão:
Se estivermos inserindo **no final** de ambas as estruturas, a **LinkedList** é mais eficiente com **O(1)**, enquanto a **ArrayList** pode atingir **O(n)**.

Contudo, em inserções em posições arbitrárias, ambas podem alcançar **O(n)** devido à necessidade de movimentação ou busca na estrutura

Vamos entender matematicamente o comportamento de **complexidade de tempo** em **ArrayList** e **LinkedList**, utilizando a notação **Big-O** para compará-las. Aqui, focaremos em como o tempo de execução dessas operações cresce à medida que o número de elementos aumenta.

1. ArrayList:

Um **ArrayList** armazena elementos em um array contínuo na memória, com índices que permitem acesso direto. Se a capacidade do array é excedida, o array precisa ser redimensionado para acomodar mais elementos.

Inserção no Final:

- Em condições normais, quando o **ArrayList** tem espaço disponível, a inserção no final do array é feita diretamente e leva **O(1)** (constante), já que não há necessidade de deslocar elementos.

- **Redimensionamento**: Se o array estiver cheio, o **ArrayList** deve ser redimensionado (duplicar o tamanho do array). Esse redimensionamento envolve criar um novo array e copiar todos os elementos do antigo array para o novo.

Matematicamente, o tempo total para redimensionar o array ao longo de várias inserções é **O(n)**, pois, ao longo do tempo, cada elemento será copiado apenas uma vez durante o redimensionamento. Portanto, o tempo médio de inserção **amortizado** é **O(1)**.

Entretanto, o **pior caso** (quando o array é redimensionado) resulta em **O(n)**.

Inserção em uma Posição Arbitrária:

Se a **inserção ocorre no meio ou no início do array, é necessário deslocar os elementos subsequentes para abrir espaço.** Isso leva **O(n)**, onde **n** é o número de elementos que precisam ser deslocados.

2. LinkedList:

Um **LinkedList** é uma estrutura onde cada elemento (nó) armazena um valor e um ponteiro para o próximo elemento na sequência. A vantagem é que a memória não precisa ser contínua, mas as operações como acesso e inserção podem ser mais lentas.

Inserção no Final:

- Como o **LinkedList** simplesmente precisa ajustar os ponteiros (se já tivermos um ponteiro para o último nó), a inserção no final é feita em tempo constante **O(1)**. Não há necessidade de redimensionar ou deslocar elementos.

Inserção em uma Posição Arbitrária:

- Para inserir em uma posição arbitrária, é necessário percorrer a lista até encontrar a posição desejada. Esse percurso tem custo **O(n)**, onde **n** é o número de elementos até a posição de inserção.

Comparação Matemática:
ArrayList (inserção no final):

- **Caso normal (sem redimensionamento)**: **O(1)** (tempo constante).
- **Pior caso (redimensionamento)**: **O(n)**, porque é necessário copiar todos os elementos para um novo array.

LinkedList (inserção no final):

- **Sempre**: **O(1)**, porque apenas os ponteiros são ajustados, sem a necessidade de copiar ou deslocar elementos.

Inserção em uma posição arbitrária:

- **ArrayList**: **O(n)**, porque é necessário deslocar elementos após a posição de inserção.
- **LinkedList**: **O(n)**, porque é necessário percorrer a lista até encontrar a posição.

Comparando as Estruturas:

•Para **inserções no final**, o **LinkedList** tem uma vantagem teórica com **O(1)** no pior caso, enquanto o **ArrayList** pode, no pior caso, levar **O(n)** devido ao redimensionamento.
•Para **inserções em posições arbitrárias**, ambas as estruturas têm um custo de **O(n)**, seja por deslocamento (no caso do **ArrayList**) ou por percurso (no caso do **LinkedList**).

Vamos fazer um exemplo matemático para calcular o tempo de inserção e entender a complexidade **O(n)** e **O(1)** em diferentes cenários, como no **ArrayList** e no **LinkedList**.

Exemplo de Inserção em ArrayList com Redimensionamento
Cenário:
Imagine que temos um **ArrayList** com capacidade inicial de 4 elementos. Quando o array é cheio, ele precisa ser redimensionado para o dobro de sua capacidade.

1.Inserção 1 a 4: Cada elemento é inserido sem precisar redimensionar, ou seja, leva **O(1)** por inserção.
2.Inserção 5: Aqui o array está cheio (capacidade de 4). Um novo array de tamanho 8 é alocado, e todos os 4 elementos são copiados para o novo array. A cópia dos elementos leva **O(4)**, e a nova inserção leva **O(1)**, totalizando **O(4) + O(1) = O(5)**.
3.Inserção 6 a 8: Não há necessidade de redimensionar, então cada inserção leva **O(1)**.
4.Inserção 9: Agora o array está cheio novamente (capacidade de 8). Ele precisa ser redimensionado para 16, copiando os 8 elementos existentes para o novo array. Isso custa **O(8)** para a cópia e **O(1)** para a nova inserção, totalizando **O(9)**.

Vamos fazer um exemplo matemático para calcular o tempo de inserção e entender a complexidade **O(n)** e **O(1)** em diferentes cenários, como no **ArrayList** e no **LinkedList**.

Exemplo de Inserção em ArrayList com Redimensionamento
Cenário:
Imagine que temos um **ArrayList** com capacidade inicial de 4 elementos. Quando o array é cheio, ele precisa ser redimensionado para o dobro de sua capacidade.

1.Inserção 1 a 4: Cada elemento é inserido sem precisar redimensionar, ou seja, leva **O(1)** por inserção.
2.Inserção 5: Aqui o array está cheio (capacidade de 4). Um novo array de tamanho 8 é alocado, e todos os 4 elementos são copiados para o novo array. A cópia dos elementos leva **O(4)**, e a nova inserção leva **O(1)**, totalizando **O(4) + O(1) = O(5)**.
3.Inserção 6 a 8: Não há necessidade de redimensionar, então cada inserção leva **O(1)**.
4.Inserção 9: Agora o array está cheio novamente (capacidade de 8). Ele precisa ser redimensionado para 16, copiando os 8 elementos existentes para o novo array. Isso custa **O(8)** para a cópia e **O(1)** para a nova inserção, totalizando **O(9)**.

Análise de Complexidade:
Embora o redimensionamento cause inserções que possam levar **O(n)** no pior caso, a maioria das inserções será feita em **O(1)**. Isso resulta em uma **complexidade amortizada** de **O(1)**, porque, no longo prazo, o custo do redimensionamento é distribuído entre muitas operações.

Exemplo de Inserção em LinkedList
Um **LinkedList** é formado por nós onde cada nó contém um dado e uma referência para o próximo nó.
Inserir um elemento no final da lista exige:

•Encontrar o último nó.
•Alterar o ponteiro desse nó para o novo nó.
Para fazer isso:
1.Se você já tem uma referência ao último nó, a inserção é feita em **O(1)**, pois você apenas ajusta os ponteiros.
2.Se precisar percorrer toda a lista até o último nó, a inserção leva **O(n)**, pois o tempo depende de quantos nós você precisa percorrer.

Exemplo Matemático:
Vamos calcular o tempo de inserção de 10 elementos para ambas as estruturas:
ArrayList:
•**Elementos 1 a 4**: Cada inserção leva **O(1)**.
•**Elemento 5**: Redimensiona o array (custo de **4** cópias) e insere o elemento (custo **O(1)**).
•**Elementos 6 a 8**: Cada inserção leva **O(1)**.
•**Elemento 9**: Redimensiona o array (custo de **8** cópias) e insere o elemento.

O tempo total é:
$O(1)+O(1)+O(1)+O(1)+O(5)+O(1)+O(1)+O(1)+O(9)=O(21)$

Exemplo Matemático:
LinkedList:
Se for necessário percorrer a lista em cada inserção para encontrar o final, o tempo será:

$$O(1)+O(2)+O(3)+O(4)+O(5)+O(6)+O(7)+O(8)+O(9)+O(10)=O(55)$$

Se você já tiver uma referência ao último nó, o tempo total será **O(1)** para cada inserção, ou seja, **O(10)**

Conclusão:

- Para o **ArrayList**, o redimensionamento causa inserções no pior caso com complexidade **O(n)**, mas em muitos casos a inserção tem **O(1)**, resultando em uma complexidade **amortizada O(1)**.

- Para o **LinkedList**, a inserção no final é **O(1)** se já tiver o ponteiro, mas pode ser **O(n)** se precisar percorrer os elementos.

Obrigado

Bischoff, Vinicius
viniciusbischof@unisnos.br
Unisinos.br