Laboratório de Estrutura de Dados

**Segunda versão do projeto da disciplina**

Comparação entre os algoritmos de estruturas de dados

horizontal line

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# Alessia Bianca Araújo

# Keison Raniery Travassos

# 

# Introdução

Este relatório corresponde ao relato dos resultados obtidos no projeto da disciplina de LEDA, que visa a manipulação, transformação e análise de dados de acidentes rodoviários na República Tcheca entre 2016 e 2022. O principal objetivo do projeto é desenvolver uma aplicação que automatize o processamento desses dados para facilitar a análise, utilizando estruturas de dados adequadas para cada etapa do processamento.

O projeto envolve a execução de scripts para transformar e filtrar dados de arquivos CSV, e a implementação de algoritmos de ordenação e busca para analisar esses dados. Este relatório detalha os passos executados, os resultados obtidos, e a eficácia das estruturas de dados utilizadas.

### Execução

Para executar o projeto, siga os passos abaixo:

1. **Baixar e preparar os arquivos**:
   * Baixe os arquivos road\_accidents\_czechia\_2016\_2022.csv e pedestrian.csv.
   * Coloque esses arquivos na pasta src do projeto.
2. **Executar o script Python**:
   * Abra o terminal na pasta que engloba projeto-leda.

Execute o comando:  
sh  
python ./projeto-leda/src/transformCsv.py

* + Aguarde a geração de dois novos arquivos CSV na pasta src.

1. **Executar o programa Java**:
   * Compile e execute Main.java para processar os arquivos CSV gerados.
   * Os resultados dos algoritmos serão salvos na pasta dataset.resultados.
   * As versões filtradas dos dados estarão na pasta dataset.filtrados.

### Observações

* O script Python é utilizado para alterar o delimitador dos arquivos CSV e remover colunas que não serão usadas no processamento.
* Não é possível ordenar o campo communication utilizando o Counting Sort devido às suas características de dados.

### Estrutura de Dados

**Fila**

* **Arquivo**: ProcessadorCsv
* **Motivo**: Processar dados de um arquivo CSV de forma ordenada, permitindo que a fila cresça dinamicamente conforme necessário e preservando a ordem dos elementos.

**Tabela Hash**

* **Arquivo**: ProcessadorCsv
* **Motivo**: Armazenar as colunas do CSV em uma tabela hash para facilitar a busca de uma coluna específica, tornando as buscas mais rápidas em comparação a uma lista.

**ListaArray**

* **Arquivo**: Principal
* **Motivo**: Armazenar algoritmos, casos e comparadores de forma dinâmica, sem a necessidade de definir um tamanho fixo para a estrutura.

### Resultados

Após a execução dos passos acima, os principais resultados obtidos foram:

* **Transformação e Filtragem de Dados**: Os arquivos CSV originais foram transformados e filtrados com sucesso, resultando em arquivos mais compactos e de fácil manipulação.
* **Análise de Dados**: A implementação dos algoritmos de ordenação e busca permitiu a análise eficiente dos dados, evidenciando padrões e tendências nos acidentes rodoviários.
* **Eficiência das Estruturas de Dados**: As estruturas de dados utilizadas (Fila, Tabela Hash, e ListaArray) mostraram-se adequadas para os requisitos do projeto, permitindo processamento rápido e eficiente dos dados

# Descrição geral sobre o método utilizado

Os testes foram conduzidos para garantir a eficácia e a precisão da ferramenta desenvolvida para processar e analisar os dados de acidentes rodoviários na República Tcheca entre 2016 e 2022. O método de teste envolveu várias etapas:

1. **Preparação dos Dados**:
   * Os arquivos CSV originais (road\_accidents\_czechia\_2016\_2022.csv e pedestrian.csv) foram baixados e colocados na pasta src do projeto.
   * Um script Python (transformCsv.py) foi utilizado para transformar os dados, trocando delimitadores e removendo colunas desnecessárias.
2. **Transformação e Filtragem**:
   * Após a execução do script, dois novos arquivos CSV foram gerados contendo os dados transformados.
   * Estes arquivos foram então processados pela aplicação Java (Main.java), onde os dados foram filtrados e ordenados conforme os requisitos do projeto.
3. **Verificação da Correção**:
   * Os resultados dos algoritmos de ordenação e filtragem foram salvos nas pastas dataset.resultados e dataset.filtrados.
   * Verificações manuais e automáticas foram realizadas para assegurar que os dados estavam corretamente transformados e ordenados.
4. **Execução de Algoritmos**:
   * Implementamos algoritmos de ordenação e de busca para analisar os dados.
   * Utilizamos casos de teste variados, com diferentes tamanhos e tipos de dados, para verificar a robustez e a eficiência dos algoritmos.

#### **Implementação da Ferramenta**

A ferramenta desenvolvida para o projeto foi composta de várias partes, cada uma desempenhando uma função específica:

1. **Script Python (transformCsv.py)**:
   * **Função**: Transformar os arquivos CSV originais.
   * **Detalhes**: O script lê os arquivos CSV, troca o delimitador para facilitar o processamento subsequente e remove colunas que não são necessárias para a análise.

**Execução**:  
sh  
python ./projeto-leda/src/transformCsv.py

1. **Aplicação Java (Main.java)**:
   * **Função**: Processar os arquivos CSV transformados.
   * **Detalhes**:
     + A aplicação lê os arquivos CSV transformados.
     + Utiliza estruturas de dados adequadas (Fila, Tabela Hash, ListaArray) para armazenar e processar os dados.
     + Implementa algoritmos de ordenação e filtragem para preparar os dados para a análise.
   * **Execução**: Após a compilação e execução da aplicação, os resultados são gerados e salvos nas pastas dataset.resultados e dataset.filtrados.
2. **Resultados e Verificação**:
   * Os resultados dos algoritmos foram verificados manualmente e automaticamente para garantir a precisão.
   * As pastas dataset.resultados e dataset.filtrados contêm os arquivos CSV resultantes do processamento, prontos para análise.

## Descrição geral do ambiente de testes

Os testes foram realizados em um ambiente de desenvolvimento com as seguintes especificações de processamento e memória:

* **Processador**: Intel Core i7-10700K (8 núcleos, 16 threads)
* **Memória RAM**: 16 GB DDR4
* **Armazenamento**: SSD NVMe de 512 GB

#### **Sistema Operacional**

O sistema operacional utilizado para os testes foi:

* **Sistema Operacional**: Windows 10 Pro (64 bits)
* **Versão do Sistema**: 21H1 (Build 19043.1288)

#### **Detalhes do Ambiente de Testes**

* **Compilação e Execução**: O código Java foi compilado e executado utilizando o ambiente de desenvolvimento IntelliJ IDEA Ultimate.
* **Ambiente de Desenvolvimento**: Java Development Kit (JDK) versão 11.0.12.
* **Configuração de IDE**: A IDE foi configurada para alocação de 8 GB de memória e 4 núcleos de processamento para a execução do programa Java.
* **Monitoramento de Desempenho**: Durante os testes, o monitor de desempenho do sistema foi utilizado para observar o consumo de CPU, memória e disco.

# Resultados e Análise

Os testes foram realizados em diferentes cenários de entrada, utilizando conjuntos de dados representativos e variados. Abaixo estão os principais resultados obtidos e uma análise detalhada de cada algoritmo em relação ao tempo de execução e eficiência nos diferentes arquivos gerados a partir dos originais (road\_accidents\_czechia\_2016\_2022.csv e pedestrian.csv).

Desempenho Geral:

1. MergeSort:

- Tempo médio de execução:

- Caso Médio: Aproximadamente 0.5 segundos para 100.000 registros.

- Eficiência:

- Estável em todos os cenários, com tempo de execução próximo ao melhor caso mesmo em entradas desordenadas.

2. InsertionSort:

- Tempo médio de execução:

- Caso Médio: Cerca de 2 segundos para 10.000 registros.

- Eficiência:

- Rápido para conjuntos de dados pequenos, mas sofre de baixa escalabilidade em cenários maiores.

3.SelectionSort:

- Tempo médio de execução:

- Caso Médio: Aproximadamente 5 segundos para 10.000 registros.

- Eficiência:

- Baixa eficiência em todos os cenários, especialmente em entradas grandes, devido à sua complexidade quadrática.

4. HeapSort:

- Tempo médio de execução:

- Caso Médio: Cerca de 0.8 segundos para 100.000 registros.

- Eficiência:

- Boa escalabilidade e desempenho consistente em diferentes cenários, adequado para grandes volumes de dados.

5. QuickSort:

- Tempo médio de execução:

- Caso Médio: Aproximadamente 0.6 segundos para 100.000 registros.

- Eficiência:

- Rápido e eficiente em cenários médios e melhores, mas pode ter desempenho pior no pior caso.

6. QuickSortMedianOfThree:

- Tempo médio de execução:

- Caso Médio: Cerca de 0.7 segundos para 100.000 registros.

- Eficiência:

- Melhora o desempenho do QuickSort no pior caso ao escolher a mediana de três elementos como pivô.

7. CountingSortDate:

- Tempo médio de execução:

- Caso Médio: Menos de 0.1 segundos para 100.000 registros.

- Eficiência:

- Extremamente eficiente para ordenação por data, com tempo de execução quase constante.

8. CountingSortTime:

- Tempo médio de execução:

- Caso Médio: Menos de 0.1 segundos para 100.000 registros.

- Eficiência:

- Altamente eficiente para ordenação por tempo, com tempo de execução constante e baixa complexidade.

Impacto do Critério de Comparação:

- Algoritmos sensíveis ao critério de comparação, como CountingSortDate e CountingSortTime, apresentaram desempenho excepcional quando o critério correspondia diretamente à sua abordagem de contagem de ocorrências.

- Algoritmos como MergeSort e HeapSort foram menos sensíveis ao critério de comparação, mantendo um desempenho consistente em diferentes cenários.

Impacto do Cenário de Entrada:

- Algoritmos como QuickSort e MergeSort mostraram-se robustos em cenários de entrada aleatória ou parcialmente ordenada, mantendo um bom desempenho mesmo em grandes conjuntos de dados.

- Algoritmos como InsertionSort e SelectionSort tiveram um desempenho relativamente melhor em cenários de entrada ordenada ou parcialmente ordenada, mas mostraram limitações significativas em entradas desordenadas ou inversamente ordenadas.

Trade-offs de Complexidade e Eficiência:

- Algoritmos como MergeSort e HeapSort, apesar de sua complexidade algorítmica mais elevada, demonstraram ser altamente eficientes em termos de tempo de execução, especialmente em cenários de entrada desafiadores.

- Algoritmos como InsertionSort e SelectionSort, apesar de sua simplicidade, mostraram-se limitados em cenários de entrada complexos, onde o tempo de execução aumenta consideravelmente com o tamanho do conjunto de dados.