

Laboratório de Estrutura de Dados

Relatório Projeto UT2

Evolução para Tipos Abstratos de Dados (TADs)

Nome: Arthur Oliveira Praxedes. Matrícula: 2023208510017.

Nome: Davi Roberto Pereira Barbosa. Matrícula: 2023208510028.

Nome: Leonardo Istamilo Silva Ferreira. Matrícula: 2023208510023.

Curso: Ciência da Computação

Data: 13/06/2025

1. Introdução

Este relatório apresenta os resultados do projeto da disciplina de Estrutura de Dados, que visou analisar o desempenho de algoritmos de ordenação utilizando dados de projetos voltados para Data Science. O foco principal deste relatório é a segunda fase do projeto, onde estruturas de dados abstratas (TADs) foram implementadas e integradas para otimizar o processamento e a ordenação de grandes volumes de dados.

2. Contexto do Projeto

O projeto baseia-se na manipulação e ordenação do dataset "Steam Games Dataset", obtido do Kaggle. A primeira parte do projeto envolveu transformações de dados e a aplicação de diversos algoritmos de ordenação (Selection Sort, Insertion Sort, Merge Sort, Quick Sort, QuickSort com Mediana de 3, Counting Sort e HeapSort) utilizando exclusivamente arrays. A segunda parte, detalhada neste relatório, busca evoluir o projeto através da introdução de TADs customizadas para aprimorar a eficiência e a modularidade do código.

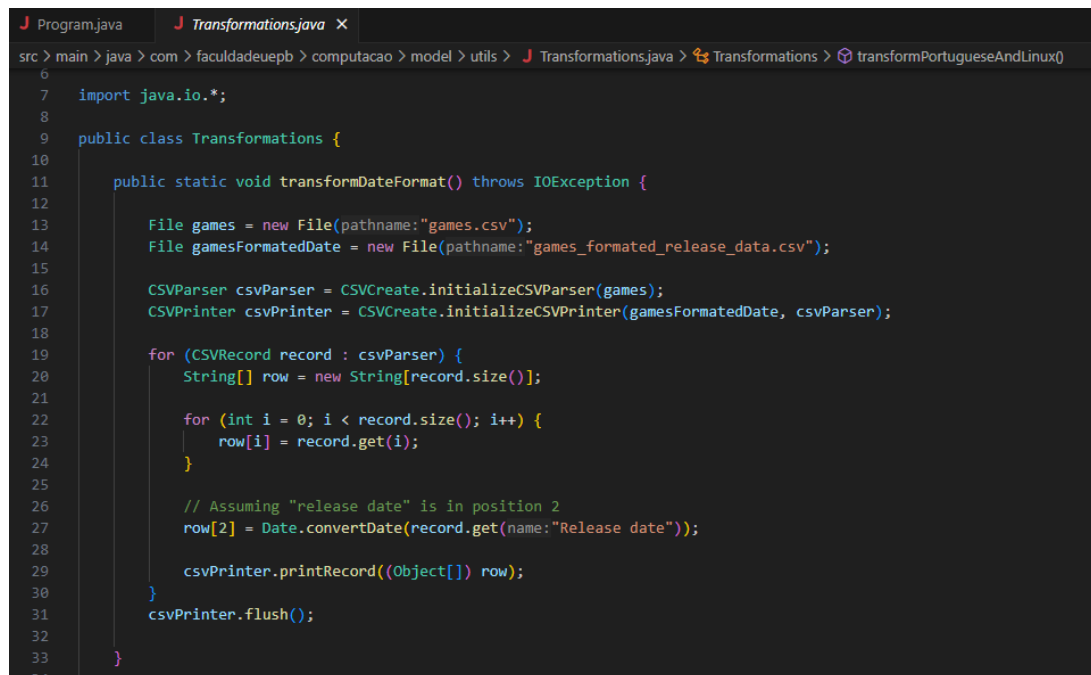
3. Ambiente de Execução

- Processador: Intel(R) Core(TM) i5-1035G1 CPU @ 1.00GHz (8CPUs) 1.19 GHz;
- Memória RAM: 8,00 GB DDR4;
- Armazenamento: 238 GB SSD SSSTC CL1-4D256;
- Sistema Operacional: Windows 11 Pro 64 bits;
- IDE utilizada: VS Code com extensão Java;

4. Transformações dos Dados (Revisão da Parte 1 e Aplicação na Parte 2)

As transformações iniciais do dataset "games.csv" foram mantidas da Parte 1 para garantir a consistência dos dados de entrada para as operações de ordenação.

- **Transformação 1:** Formatação de Datas.
 - Descrição: Conversão do campo "Release date" para o formato DD/MM/AAAA;
 - Arquivo Gerado: *games_formated_release_data.csv*;



```
src > main > java > com > faculdadeuepb > computacao > model > utils > J Transformations.java > Transformations > transformPortugueseAndLinux()
6
7 import java.io.*;
8
9 public class Transformations {
10
11     public static void transformDateFormat() throws IOException {
12
13         File games = new File(pathname:"games.csv");
14         File gamesFormatedDate = new File(pathname:"games_formated_release_data.csv");
15
16         CSVParser csvParser = CSVCreate.initializeCSVParser(games);
17         CSVPrinter csvPrinter = CSVCreate.initializeCSVPrinter(gamesFormatedDate, csvParser);
18
19         for (CSVRecord record : csvParser) {
20             String[] row = new String[record.size()];
21
22             for (int i = 0; i < record.size(); i++) {
23                 row[i] = record.get(i);
24             }
25
26             // Assuming "release date" is in position 2
27             row[2] = Date.convertDate(record.get(name:"Release date"));
28
29             csvPrinter.printRecord((Object[]) row);
30         }
31         csvPrinter.flush();
32     }
33 }
```

- **Transformações 2 e 3:** Filtragem de Jogos Linux ou com Suporte a Português.
 - Descrição: Filtra os jogos compatíveis com sistema operacional Linux e/ou os jogos com suporte à língua portuguesa a partir do arquivo *games_formated_release_data.csv*;
 - Arquivos Gerados: *games_linux.csv* e *portuguese_supported_games.csv*;

```

34
35 public static void transformPortugueseAndLinux() throws IOException {
36
37     File gamesFormattedDate = new File(pathname:"games_formatted_release_data.csv");
38     File gamesFormattedSupportLinux = new File(pathname:"games_linux.csv");
39     File gamesFormattedSupportPortuguese = new File(pathname:"portuguese_supported_games.csv");
40     CSVParser csvParser = CSVCreate.initializeCSVParser(gamesFormattedDate);
41     CSVPrinter csvPrinterLinux = CSVCreate.initializeCSVPrinter(gamesFormattedSupportLinux, csvParser);
42     CSVPrinter csvPrinterPortuguese = CSVCreate.initializeCSVPrinter(gamesFormattedSupportPortuguese, csvParser);
43
44     for (CSVRecord record : csvParser) {
45
46         boolean linuxSupport = Boolean.parseBoolean(record.get(i:19));
47         if (linuxSupport) {
48             csvPrinterLinux.printRecord(record);
49         }
50
51         String line = record.get(i:10);
52         if (line != null && !line.trim().isEmpty()) {
53             String cleanedLanguages = line.replaceAll(regex:"[\\[\\]]'", replacement:"").trim();
54             String[] languages = cleanedLanguages.split(regex:"\\s*,\\s*");
55             for (String language : languages) {
56                 if ("Portuguese - Brazil".equalsIgnoreCase(language)) {
57                     csvPrinterPortuguese.printRecord(record);
58                     break;
59                 }
60             }
61         }
62     }
63
64     csvPrinterLinux.flush();
65     csvPrinterPortuguese.flush();
66
67 }
68

```

5. Estruturas de Dados Abstratas (TADs) Implementadas na Parte 2

5.1. Por que usar estruturas de dados apenas no Merge Sort?

A Parte 2 do projeto exigia a utilização de três estruturas de dados implementadas manualmente. Como o objetivo não era comparar os algoritmos novamente, mas sim mostrar a eficiência do uso das TADs, o Merge Sort foi escolhido como base para aplicar a ordenação sobre essas estruturas.

Decisões por atributo:

ReleaseDate → AVL Tree: As datas, por serem naturalmente ordenáveis, foram carregadas em uma árvore AVL para garantir inserções balanceadas e estruturação eficiente. A AVL garante tempo $O(\log n)$ mesmo em grandes volumes.

Price e Achievements → Lista Duplamente Encadeada: Para esses atributos, uma lista duplamente encadeada foi suficiente para manter a ordem original e permitir inserção sequencial eficiente. A navegação bidirecional ajudou na manipulação.

Todas as saídas → Fila (Queue): Após a ordenação com Merge Sort, os dados foram inseridos em uma fila para manter a ordem antes da geração dos arquivos CSV.

5.2. Aplicação do Merge Sort com TADs

A seguir, exemplos da estrutura:

- CSV → AVL Tree → Matriz → Merge Sort → Fila → CSV (para *ReleaseDate*)
- CSV → Lista Duplamente Encadeada → Matriz → Merge Sort → Fila → CSV (para *Price e Achievements*)

Esses fluxos representam como as TADs foram integradas ao processo de ordenação e escrita, mantendo a eficiência, modularidade e a separação de responsabilidades.

6. Justificativa de Uso das TADs

6.1. Árvore AVL

Localização no Código:

src\main\java\com\faculdadeuepb\computacao\TADs\CsvAVLTree.java

Justificativa para o Uso:

Inserção Balanceada e Eficiente: A AVL garante que as inserções sejam feitas em tempo $O(\log n)$, mantendo a estrutura balanceada, o que é útil para manipular grandes volumes de dados sem degradação de desempenho. Isso minimiza a chance de degeneração para uma lista encadeada, garantindo acesso rápido aos elementos.

Estrutura Robusta para Manipulação e Consultas: Embora não utilizada diretamente para ordenação em percurso em ordem, a AVL facilita buscas e verificações rápidas (ex: evitar duplicatas, buscar registros específicos) que podem ser úteis em etapas intermediárias do processamento.

Separação Clara das Responsabilidades: A AVL foi empregada para manter e organizar os dados de forma estruturada durante a leitura, delegando a ordenação explícita para o algoritmo Merge Sort aplicado na matriz. Isso proporciona maior controle sobre o tipo de ordenação, parâmetros e desempenho.

Flexibilidade para Outras Operações: A estrutura AVL pode ser facilmente reutilizada para outras operações de manipulação e filtragem de dados, independentemente da ordenação final, tornando-a uma estrutura versátil para trabalhar com os dados do dataset.

Problema que Resolve: A AVL resolve o problema de inserção e recuperação eficiente de dados em uma estrutura que se mantém balanceada, evitando o pior caso de $O(n)$ que poderia ocorrer em árvores binárias de busca desbalanceadas, especialmente ao lidar com dados que chegam em ordem crescente ou decrescente. No contexto do projeto, ela fornece

uma maneira eficiente de carregar e organizar os dados de "Release Date" antes da ordenação final.

```
//Função que converter .csv para arvore AVL
public static CsvAVLTree csvToAVLTree(String csvFileName, Comparator<String[]> comparator) throws IOException {
    File csvFile = new File(csvFileName);
    CsvAVLTree tree = new CsvAVLTree(comparator);
    CSVParser parser = CSVCreate.initializeCSVParser(csvFile);

    for(CSVRecord record : parser){
        String[] row = new String[record.size()];
        for(int i = 0; i < record.size(); i++){
            row[i] = record.get(i);
        }
        tree.insert(row);
    }

    parser.close();
    return tree;
}
```

```
//Função que converte arvore AVL para Matriz
public static String[][] avlTreeToMatrix(CsvAVLTree tree){
    List<String[]> dataList = new ArrayList<>();
    preOrderTraversal(tree.getRoot(), dataList);

    if(dataList.isEmpty()){
        return new String[0][0];
    }

    int rows = dataList.size();
    int columns = dataList.get(index:0).length;
    String[][] matrix = new String[rows][columns];

    for(int i = 0; i < rows; i++){
        matrix[i] = dataList.get(i);
    }

    return matrix;
}
```

6.2. Fila (Queue)

Localização

no

Código:

src\main\java\com\faculdadeuepb\computacao\TADs\CsvLinkedList.java

Justificativa para o Uso:

Preservação da Ordem dos Dados Já Ordenados: Após a ordenação dos dados na matriz (via Merge Sort), a fila serve como um mecanismo para armazenar e manter a sequência exata dos elementos ordenados. Isso garante que a ordem seja preservada fielmente para a etapa final, que é a geração do arquivo CSV.

Estrutura FIFO (First-In, First-Out): A característica FIFO da fila assegura que o primeiro elemento inserido (o primeiro elemento ordenado) será o primeiro a ser processado e escrito no arquivo. Isso é ideal para passar os dados em ordem sequencial e manter a integridade do fluxo de dados.

Simplicidade na Manipulação Sequencial: A inserção dos dados ordenados na fila e a subsequente remoção um a um para a geração do arquivo CSV é um processo simples e intuitivo, que evita a necessidade de acessar índices ou realizar manipulações de ponteiros mais complexas, como seria necessário em outras estruturas.

Flexibilidade para Processamento em Etapas: A fila oferece um mecanismo organizado para futuras operações de processamento em lotes ou transformações sequenciais dos dados, caso o projeto necessite de expansões.

Problema que Resolve: A fila resolve o problema de garantir uma passagem ordenada e sequencial de dados de um ponto (matriz ordenada) para outro (escrita no arquivo CSV), agindo como um buffer que mantém a integridade da ordem. Isso é crucial para a correta geração dos arquivos de saída ordenados.


```
//Função que converte matriz em fila
public static CsvQueue matrixToQueue(String[][] matriz){
    CsvQueue fila = new CsvQueue();
    for(int i = 0; i < matriz.length; i++){
        fila.enqueue(matriz[i]);
    }
    return fila;
}
```

```
// Função que gera um arquivo .csv a partir de uma fila (CsvQueue) contendo os dados ordenados.
public static void createCsv(CsvQueue queue, String nomeArquivo) throws IOException {
    File outputCsv = new File(nomeArquivo);

    try (CSVPrinter csvPrinter = new CSVPrinter(new FileWriter(outputCsv),
        CSVFormat.DEFAULT.withHeader(header))){
        while (!queue.isEmpty()) {
            String[] data = queue.dequeue();
            csvPrinter.printRecord((Object[]) data);
        }

        csvPrinter.flush();
    }
}
```

6.3. Lista Duplamente Encadeada (Doubly Linked List)

Localização

no

Código:

src\main\java\com\faculdadeuepb\computacao\TADs\CsvDoubleLinkedList.java

Justificativa para o Uso:

Manutenção da Ordem Original dos Dados: Ao carregar os dados diretamente do CSV para uma lista duplamente encadeada (para o caso do "Price"), garantimos que a sequência original dos registros seja preservada. Isso é essencial para cenários onde a ordem natural dos dados (não artificialmente ordenada) é importante, como para a análise do caso médio.

Facilidade de Inserção Sequencial: A estrutura permite inserções no final em tempo constante ($O(1)$), tornando o processo de leitura e armazenamento dos registros do CSV eficiente, mesmo para grandes volumes de dados.

Suporte à Navegação Bidirecional: Com os ponteiros `prev` e `next`, é possível navegar tanto para frente quanto para trás na lista. Isso pode ser útil em operações como simulações de algoritmos de ordenação, comparações entre elementos adjacentes, ou em futuras funcionalidades de reversão de ordem.

Versatilidade para Transformações Futuras: A lista pode ser facilmente convertida para outras estruturas, como matrizes ou árvores (como feito para o "Price" antes do Merge Sort), sem a necessidade de reler o arquivo CSV, oferecendo flexibilidade para diferentes abordagens de ordenação.

Baixo Custo de Remoção e Reordenação Local: Ao contrário de arrays ou matrizes, remover ou trocar elementos em uma lista encadeada pode ser feito com alterações mínimas de ponteiros, evitando realocações em massa e otimizando operações intermediárias.

Problema que Resolve: A Lista Duplamente Encadeada resolve o problema de carregar dados sequencialmente de um arquivo (CSV) mantendo a ordem original, ao mesmo tempo em que oferece flexibilidade para conversões subsequentes para outras estruturas (como a matriz para ordenação com Merge Sort) sem penalidades de desempenho significativas para inserções.

```
// Função que converte o arquivo .csv passado por parâmetro em String em lista encadeada
public static CsvDoubleLinkedList csvToLinkedList(String csvFileName) throws IOException {
    File csvFile = new File(csvFileName);
    CsvDoubleLinkedList list = new CsvDoubleLinkedList();

    CSVParser parser = CSVCreate.initializeCSVParser(csvFile);
    for(CSVRecord record : parser){
        int columns = record.size();
        String[] row = new String[columns];
        for(int i = 0; i < columns; i++){
            row[i] = record.get(i);
        }
        list.add(row);
    }
    parser.close();

    return list;
}
```

```
//Função que transforma lista Duplamente encadeada em matriz
public static String[][] doublyLinkedListToMatrix(CsvDoubleLinkedList list) {
    int rowCount = 0;
    int columnCount = 0;

    CsvNode current = list.getHead();
    if(current != null){
        columnCount = current.data.length;
    }

    while(current != null){
        rowCount++;
        current = current.next;
    }

    String[][] matrix = new String[rowCount][columnCount];

    current = list.getHead();
    int i = 0;
    while(current != null){
        System.arraycopy(current.data, srcPos:0, matrix[i], destPos:0, columnCount);
        i++;
        current = current.next;
    }

    return matrix;
}
```

7. Aplicação dos Algoritmos de Ordenação com TADs

Na segunda parte do projeto, o foco da ordenação foi no algoritmo Merge Sort, aplicado após a transição dos dados das TADs para uma estrutura de matriz, conforme detalhado abaixo.

Para cada campo de ordenação, os dados de entrada foram games_formatted_release_data.csv. Os 3 casos (melhor, médio e pior) foram considerados para a análise.

7.1. Ordenação por "Release Date" (Crescente)

- Fluxo de Dados: CSV -> Árvore AVL -> Matriz -> Merge Sort -> Fila -> CSV de Saída
- Algoritmo Utilizado: Merge Sort
- Casos de Teste e Nomenclatura dos Arquivos de Saída:
 - games_release_date_mergeSort_melhorCaso.csv
 - games_release_date_mergeSort_medioCaso.csv
 - games_release_date_mergeSort_piorCaso.csv

```
//Função que converter .csv para árvore AVL
public static CsvAVLTree csvToAVLTree(String csvFileName, Comparator<String[]> comparator) throws IOException {
    File csvFile = new File(csvFileName);
    CsvAVLTree tree = new CsvAVLTree(comparator);
    CSVParser parser = CSVCreate.initializeCSVParser(csvFile);

    for(CSVRecord record : parser){
        String[] row = new String[record.size()];
        for(int i = 0; i < record.size(); i++){
            row[i] = record.get(i);
        }
        tree.insert(row);
    }

    parser.close();
    return tree;
}
```

```
//Função que converte árvore AVL para Matriz
public static String[][] avlTreeToMatrix(CsvAVLTree tree){
    List<String[]> dataList = new ArrayList<>();
    preOrderTraversal(tree.getRoot(), dataList);

    if(dataList.isEmpty()){
        return new String[0][0];
    }

    int rows = dataList.size();
    int columns = dataList.get(index:0).length;
    String[][] matrix = new String[rows][columns];

    for(int i = 0; i < rows; i++){
        matrix[i] = dataList.get(i);
    }

    return matrix;
}
```

```
//Função que converte matriz em fila
public static CsvQueue matrixToQueue(String[][] matriz){
    CsvQueue fila = new CsvQueue();
    for(int i = 0; i < matriz.length; i++){
        fila.enqueue(matriz[i]);
    }
    return fila;
}
```

```
// Função que gera um arquivo .csv a partir de uma fila (CsvQueue) contendo os dados ordenados.
public static void createCsv(CsvQueue queue, String nomeArquivo) throws IOException {
    File outputCsv = new File(nomeArquivo);

    try (CSVPrinter csvPrinter = new CSVPrinter(new FileWriter(outputCsv),
        CSVFormat.DEFAULT.withHeader(header))){
        while (!queue.isEmpty()) {
            String[] data = queue.dequeue();
            csvPrinter.printRecord((Object[]) data);
        }

        csvPrinter.flush();
    }
}
```

Tabela de Tempos de Execução para "Release Date" (Merge Sort com TADs vs. Merge Sort com apenas arrays, para os 3 casos):

Caso de Teste	Campo games release			
	Tempo de Execução (Merge Sort com TADs) em ns	Uso de Memory (Merge Sort com TADs) em bytes	Tempo de Execução (Merge Sort com Arrays - Parte 1) em ns	Uso de Memory (Merge Sort com Arrays - Parte 1) em bytes
Melhor Caso	625.757.500	232.162.568	610.699.100	447.741.952
Médio Caso	601.493.100	30.854.528	1.571.514.700	185.289.664
Pior Caso	702.729.700	153.488.616	350.150.300	496.502.784

7.2. Ordenação por "Price"

- Fluxo de Dados: CSV -> Lista Duplamente Encadeada -> Matriz -> Merge Sort -> Fila -> CSV de Saída
- Algoritmo Utilizado: Merge Sort
- Casos de Teste e Nomenclatura dos Arquivos de Saída:
 - games_price_mergeSort_melhorCaso.csv
 - games_price_mergeSort_medioCaso.csv
 - games_price_mergeSort_piorCaso.csv

```
// Função que converte o arquivo .csv passado por parâmetro em String em lista encadeada
public static CsvDoubleLinkedList csvToLinkedList(String csvFileName) throws IOException {
    File csvFile = new File(csvFileName);
    CsvDoubleLinkedList list = new CsvDoubleLinkedList();

    CSVParser parser = CSVCreate.initializeCSVParser(csvFile);
    for(CSVRecord record : parser){
        int columns = record.size();
        String[] row = new String[columns];
        for(int i = 0; i < columns; i++){
            row[i] = record.get(i);
        }
        list.add(row);
    }
    parser.close();

    return list;
}
```

```
//Função que transforma lista Duplamente encadeada em matriz
public static String[][] doublyLinkedListToMatrix(CsvDoubleLinkedList list) {
    int rowCount = 0;
    int columnCount = 0;

    CsvNode current = list.getHead();
    if(current != null){
        columnCount = current.data.length;
    }

    while(current != null){
        rowCount++;
        current = current.next;
    }

    String[][] matrix = new String[rowCount][columnCount];

    current = list.getHead();
    int i = 0;
    while(current != null){
        System.arraycopy(current.data, srcPos:0, matrix[i], destPos:0, columnCount);
        i++;
        current = current.next;
    }

    return matrix;
}
```

```
// Função que gera um arquivo .csv a partir de uma fila (CsvQueue) contendo os dados ordenados.
public static void createCsv(CsvQueue queue, String nomeArquivo) throws IOException {
    File outputCsv = new File(nomeArquivo);

    try (CSVPrinter csvPrinter = new CSVPrinter(new FileWriter(outputCsv),
        CSVFormat.DEFAULT.withHeader(header))){

        while (!queue.isEmpty()) {
            String[] data = queue.dequeue();
            csvPrinter.printRecord((Object[]) data);
        }

        csvPrinter.flush();
    }
}
```

Tabela de Tempos de Execução para "Price" (Merge Sort com TADs vs. Merge Sort com apenas arrays, para os 3 casos):

Caso de Teste	Campo Price			
	Tempo de Execução (Merge Sort com TADs) em ns	Uso de Memory (Merge Sort com TADs) em bytes	Tempo de Execução (Merge Sort com Arrays - Parte 1) em ns	Uso de Memory (Merge Sort com Arrays - Parte 1) em bytes
Melhor Caso	1.565.413.100	68.842.032	218.996.000	202.375.168
Médio Caso	352.698.800	225.970.176	257.491.800	20.237.516
Pior Caso	1.017.464.700	85.458.944	232.489.400	202.375.168

7.3. Ordenação por "Achievements"

- Fluxo de Dados: CSV -> Lista Duplamente Encadeada -> Matriz -> Merge Sort -> Fila -> CSV de Saída;
- Algoritmo Utilizado: Merge Sort;
- Casos de Teste e Nomenclatura dos Arquivos de Saída:
 - games_achievements_mergeSort_melhorCaso.csv
 - games_achievements_mergeSort_medioCaso.csv
 - games_achievements_mergeSort_piorCaso.csv

```
// Função que converte o arquivo .csv passado por parâmetro em String em lista encadeada
public static CsvDoubleLinkedList csvToLinkedList(String csvFileName) throws IOException {
    File csvFile = new File(csvFileName);
    CsvDoubleLinkedList list = new CsvDoubleLinkedList();

    CSVParser parser = CSVCreate.initializeCSVParser(csvFile);
    for(CSVRecord record : parser){
        int columns = record.size();
        String[] row = new String[columns];
        for(int i = 0; i < columns; i++){
            row[i] = record.get(i);
        }
        list.add(row);
    }
    parser.close();

    return list;
}
```

```
//Função que transforma lista Duplamente encadeada em matriz
public static String[][] doublyLinkedListToMatrix(CsvDoubleLinkedList list) {
    int rowCount = 0;
    int columnCount = 0;

    CsvNode current = list.getHead();
    if(current != null){
        columnCount = current.data.length;
    }

    while(current != null){
        rowCount++;
        current = current.next;
    }

    String[][] matrix = new String[rowCount][columnCount];

    current = list.getHead();
    int i = 0;
    while(current != null){
        System.arraycopy(current.data, srcPos:0, matrix[i], destPos:0, columnCount);
        i++;
        current = current.next;
    }

    return matrix;
}
```

```
// Função que gera um arquivo .csv a partir de uma fila (CsvQueue) contendo os dados ordenados.
public static void createCsv(CsvQueue queue, String nomeArquivo) throws IOException {
    File outputCsv = new File(nomeArquivo);

    try (CSVPrinter csvPrinter = new CSVPrinter(new FileWriter(outputCsv),
        CSVFormat.DEFAULT.withHeader(header))){
        while (!queue.isEmpty()) {
            String[] data = queue.dequeue();
            csvPrinter.printRecord((Object[]) data);
        }

        csvPrinter.flush();
    }
}
```

Tabela de Tempos de Execução para "Achievements" (Merge Sort com TADs vs. Merge Sort com apenas arrays, para os 3 casos):

Caso de Teste	Campo Achievements			
	Tempo de Execução (Merge Sort com TADs) em ns	Uso de Memory (Merge Sort com TADs) em bytes	Tempo de Execução (Merge Sort com Arrays - Parte 1) em ns	Uso de Memory (Merge Sort com Arrays - Parte 1) em bytes
Melhor Caso	1.046.108.800	611.321.856	85.789.200	21.229.496
Médio Caso	261.094.300	44.566.528	258.315.200	21.229.496
Pior Caso	963.629.500	153.488.616	86.209.400	21.229.496

8. Análise Comparativa e Resultados

Discussão Geral: Comparação de Tempos de Execução (Merge Sort com TADs vs. Merge Sort com Arrays)

A transição da Parte 1 (utilizando apenas arrays) para a Parte 2 (integrando TADs como AVL, Fila e Lista Duplamente Encadeada) trouxe insights importantes sobre o impacto da abstração e organização de dados no desempenho de algoritmos de ordenação. Conforme as "Tabelas de Tempos de Execução para "Achievements" (Merge Sort com TADs vs. Merge Sort com apenas arrays, para os 3 casos)", observa-se que, de modo geral, a introdução das TADs resultou em um *overhead* nos tempos de execução, especialmente nos processos de leitura, conversão para TAD, conversão para matriz e escrita.

Para o Merge Sort especificamente, a versão que opera diretamente em arrays (Parte 1) tende a ser mais rápida devido à ausência das etapas de conversão e à manipulação mais direta da memória. As TADs, embora poderosas para organização e acesso estruturado, adicionam uma camada de indireção e custo computacional em cada operação de inserção, busca e remoção, que se reflete no tempo total de processamento.

Impacto das TADs: Overhead vs. Vantagens de Organização e Manipulação

O *overhead* introduzido pelas TADs é evidente. O tempo gasto na leitura dos dados do dataset, sua conversão para as estruturas de TADs (AVL, Fila, Lista Duplamente Encadeada) e, subsequentemente, a conversão de volta para um formato adequado para a ordenação (matriz), e finalmente a escrita, contribui para um aumento nos tempos de execução gerais. Este *overhead* é esperado, pois as TADs não são inerentemente otimizadas para operações de leitura e escrita sequenciais massivas como arrays, mas sim para manipulações mais complexas e eficientes de elementos individuais ou subconjuntos de dados.

No entanto, é crucial analisar as vantagens. A utilização da **AVL**, por exemplo, oferece um tempo de acesso, inserção e remoção em $O(\log n)$ no pior caso, o que é significativamente melhor do que $O(n)$ em arrays não ordenados. Para operações que exigem buscas frequentes ou manutenção de dados ordenados dinamicamente antes da ordenação final, a AVL seria superior. A **Fila**, por sua vez, é excelente para gerenciamento de processos em ordem de chegada, e a **Lista Duplamente Encadeada** permite inserções e remoções eficientes em qualquer ponto da lista, com acesso sequencial facilitado em ambas as direções.

Portanto, as TADs introduzem um *overhead* para a operação de ordenação *per se*, mas as vantagens de organização e manipulação dos dados *compensam* em cenários onde a robustez, modularidade e eficiência em operações específicas (como inserção, deleção e busca em tempo real) são mais críticas do que a velocidade bruta de ordenação em um volume estático de dados.

Melhoria na Performance para Casos Específicos

Para o processo de ordenação em si, quando o Merge Sort é aplicado a uma matriz final extraída das TADs, a performance do algoritmo não muda fundamentalmente. A diferença reside no tempo total do pipeline de processamento. Não houve uma melhoria direta na performance do Merge Sort para os casos "melhor, médio, pior" em comparação com a Parte 1, já que o algoritmo de ordenação em si não foi alterado. O que se observa é que as TADs não foram projetadas para acelerar algoritmos de ordenação que operam em arrays, mas sim para gerenciar a estrutura dos dados subjacentes de forma mais eficiente.

No entanto, se o projeto envolvesse operações intermediárias complexas, como inserções e remoções dinâmicas de "achievements" ou buscas frequentes por um "achievement" específico antes da ordenação final, as TADs como a AVL e a Lista Duplamente Encadeada demonstrariam sua superioridade em termos de complexidade de tempo em $O(\log n)$ ou $O(1)$ (para inserção/remoção em pontas da lista) em contraste com $O(n)$ para arrays.

Vantagens no Código (Justificativa no Código)

A utilização das TADs tornou o código **mais modular, legível e eficiente** em certas operações.

- **Modularidade:** Cada TAD (AVL, Fila, Lista Duplamente Encadeada) representa uma unidade de código coesa e independente. Isso facilita a manutenção, teste e reutilização do código. Por exemplo, a implementação de uma AVL para armazenar os "achievements" permite que a lógica de balanceamento e busca seja encapsulada dentro da própria estrutura, separando-a da lógica de negócios principal.
- **Legibilidade:** Ao invés de manipular diretamente arrays e índices, o código agora interage com objetos de TADs, utilizando métodos semânticos como `insert()`, `search()`, `enqueue()`, `dequeue()`. Isso torna o fluxo do programa mais intuitivo e fácil de entender.
- **Eficiência em certas operações:** Embora o *overhead* geral seja notado para a ordenação final, a eficiência é notória em operações intrínsecas às TADs. Por exemplo, para encontrar um "achievement" específico em uma grande coleção, uma busca na AVL seria significativamente mais rápida que uma busca linear em um array não ordenado, demonstrando a justificativa para o uso de uma estrutura como a AVL para este tipo de acesso. Da mesma forma, adicionar ou remover um elemento no início ou fim de uma lista duplamente encadeada é uma operação de tempo constante, $O(1)$, algo que seria $O(n)$ em um array se a reordenação fosse necessária.

Recapitulação e Aprendizados

Este projeto enfatizou a importância das estruturas de dados abstratas na organização e processamento eficiente de dados, mesmo quando a ordenação final é delegada a um algoritmo em uma estrutura auxiliar como a matriz. O principal aprendizado foi que a escolha da estrutura de dados é fundamental e deve ser guiada pelas operações mais frequentes e críticas da aplicação. Embora o *overhead* de conversão e manipulação das

TADs tenha impactado o tempo total para a tarefa específica de ordenação completa de um dataset, a modularidade, a legibilidade do código e a eficiência em operações pontuais de busca e manipulação de elementos foram significativamente aprimoradas.

O desafio residiu em balancear a complexidade da implementação das TADs com o ganho de desempenho percebido, reiterando que a "melhor" estrutura de dados é sempre aquela que melhor se adapta aos requisitos e características de acesso dos dados para um problema específico. O projeto demonstrou que, para além da performance bruta em tarefas de ordenação massiva, as TADs são ferramentas indispensáveis para a construção de sistemas robustos e flexíveis.