UNIVERSIDADE ESTADUAL DA PARAÍBA

CENTRO DE CIÊNCIA E TECNOLOGIA- CCT

CIÊNCIA DA COMPUTAÇÃO

LUCAS ANACLETO BATISTA ALMEIDA

JOÃO GABRIEL DOS SANTOS PEREIRA

PROJETO 2: TEMA 2 - PASSWORDS

Aprimoramento e implementação de novas estruturas de dados no projeto 1.

CAMPINA GRANDE

2024

SUMÁRIO

1. INTRODUÇÃO

Neste relatório, daremos continuidade ao estudo iniciado na primeira etapa do projeto, onde foi realizada uma análise comparativa de diferentes algoritmos de ordenação aplicados a um dataset com mais de 600 mil senhas, consideradas as mais comuns utilizadas. Naquela fase, foram abordados os algoritmos Selection Sort, Insertion Sort, Quick Sort, Merge Sort, Counting Sort, Heapsort e Quick Sort com Mediana 3, todos implementados utilizando exclusivamente a estrutura de dados array.

Na presente fase do projeto, ampliamos nossa abordagem incorporando três novas estruturas de dados além dos arrays, com o objetivo de otimizar a eficiência e a usabilidade dos algoritmos de ordenação. Nosso objetivo é explorar como a introdução de diferentes estruturas de dados pode influenciar a performance dos algoritmos de ordenação em um grande conjunto de dados, proporcionando uma compreensão mais aprofundada de suas aplicações práticas e teóricas.

2. DESENVOLVIMENTO

2.1 Estruturas de dados utilizadas

Neste projeto, implementamos três novas estruturas de dados: Fila, Árvore Binária de Busca e Tabela Hash. A fila, que segue a política FIFO (First In, First Out), é ideal para o processamento sequencial de lotes de dados, otimizando recursos e mantendo o fluxo contínuo. A árvore binária de busca, uma estrutura hierárquica com no máximo dois filhos por nó, permite operações eficientes de busca, inserção e remoção, garantindo listas ordenadas dinamicamente. A tabela hash, que mapeia chaves a valores usando uma função de hash, oferece buscas, inserções e remoções rápidas, sendo ideal para armazenar frequências de dados, buscar duplicatas e filtrar grandes conjuntos de dados antes da ordenação.

2.2 Aplicação e justificativas

Para começar, decidimos melhorar a organização, atendendo também às correções sugeridas pelo orientador, reestruturando o projeto em várias pastas específicas: *dataset, drivers, entities, interfaces, repositories, services, e useCases*. Cada pasta armazena classes que representam ações específicas dentro do código, o que facilita a navegação pela estrutura do projeto.

Além disso, implementamos as estruturas de dados mencionadas anteriormente em locais específicos do código, visando otimizar a execução de diversas operações no projeto. A seguir, discutiremos onde essas mudanças ocorreram, os motivos que nos levaram a optar por essas alterações e as melhorias obtidas como resultado

2.2.1 Primeira Estrutura (Queue)

Uma das principais mudanças implementadas no projeto foi a substituição do uso de arrays pela estrutura de dados fila (Queue) na classe *GetVariables*, especificamente no método *readDataToArray* . Este método é responsável por ler o arquivo .csv contendo as senhas e armazenar esses dados em uma estrutura para processamento posterior. Inicialmente, essa tarefa era realizada utilizando arrays, o que apresentou algumas limitações significativas em

termos de eficiência e flexibilidade. Para abordar esses problemas, decidimos adotar a fila como a estrutura de dados principal para essa operação.

Exemplo de código antes da alteração:

```java
public String[] readDataToArray() {  3 usages  Lucas Anacleto *
    try (BufferedReader br = new BufferedReader(new FileReader( fileName: "dataset/passwords_test.csv"))) {
        br.readLine();
        int numLines = 0;
        String line;
        while ((line = br.readLine()) != null) {
            numLines++;
        }
        String[] dataArray = new String[numLines];
        br.close();
        BufferedReader br2 = new BufferedReader(new FileReader( fileName: "dataset/passwords_test.csv"));
        br2.readLine();
        int index = 0;
        while ((line = br2.readLine()) != null) {
            dataArray[index++] = line;
        }

        return dataArray;
    } catch (IOException e) {
        e.printStackTrace();
        return new String[0];
    }
}
```

Exemplo de código depois da alteração:

```java
public String[] readDataToArray() {  3 usages  Lucas Anacleto
    CustomQueue queue = new CustomQueue();
    String line;
    try (BufferedReader br = new BufferedReader(new FileReader( fileName: "./src/dataset/passwords_test.csv"))) {
        br.readLine();
        while ((line = br.readLine()) != null) {
            queue.insert(line);
        }
        br.close();
        return queue.toArray();
    } catch (IOException e) {
        e.printStackTrace();
        return new String[0];
    }
}
```

Antes, o método utilizava um array para armazenar as senhas lidas do arquivo .csv, exigindo duas leituras do arquivo: uma para contar o número de linhas e outra para preencher o array. Após a mudança, a estrutura de dados fila (Queue), representada pela classe *CustomQueue*, foi implementada. A fila permite a inserção dinâmica de elementos durante uma única leitura do arquivo, eliminando a necessidade de redimensionamento de arrays.

Essa estrutura permite uma melhor gestão de memória, evitando a necessidade de redimensionar arrays quando o número de elementos cresce, além de suportar a inserção

dinâmica de elementos, ajustando-se automaticamente ao tamanho do dataset. Sua semântica garante que a ordem de inserção dos elementos seja preservada, essencial para o correto processamento dos dados lidos.

Essas mudanças foram fundamentais para melhorar a eficiência e a organização do projeto, permitindo uma manipulação mais eficaz dos dados e facilitando futuras expansões e manutenções.

2.2.2 Segunda estrutura (Árvore Binária de Busca)

Uma significativa melhoria foi implementada nos arquivos dentro da pasta *useCases*, onde a funcionalidade é gerar arrays do pior, médio e melhor caso com base em critérios como mês, tamanho da senha e data. Inicialmente, cada chamada para obter o array exigia a ordenação dos dados, resultando em uma complexidade de O(n log n) a partir do método *Sort*. Para otimizar esse processo, substituímos a abordagem anterior pela implementação de uma Árvore Binária de Busca (BST).

A BST foi introduzida para inserir os dados uma única vez, permitindo a obtenção dos arrays em ordem crescente ou decrescente através de travessias in-order na árvore.

Exemplo de código antes da alteração:

```
4   public class CreateCasesByLength {  2 usages  Lucas Anacleto
5       GetVariables csvToArray = new GetVariables();  1 usage
6       String[] data = csvToArray.readDataToArray();  5 usages
7
8       Comparator<String> comparatorLengthPasswordCrescent = Comparator.comparingInt((String str) -> {...});
18      Comparator<String> comparatorLengthPasswordDecreasing = Comparator.comparingInt((String str) -> {...}).reversed();
28
29      public String[] bestCase() {  1 usage  Lucas Anacleto
30          String[] dataTransforming = Arrays.copyOf(data, data.length);
31          Arrays.sort(dataTransforming, comparatorLengthPasswordDecreasing);
32          return dataTransforming;
33      }
34      public String[] mediumCase() {  1 usage  Lucas Anacleto
35          return data;
36      }
37      public String[] worstCase() {  1 usage  Lucas Anacleto
38          String[] dataTransforming = Arrays.copyOf(data, data.length);
39          Arrays.sort(dataTransforming, comparatorLengthPasswordCrescent);
40          return dataTransforming;
41      }
42
43  }
```

Exemplo de código depois da alteração:

```java
public class CreateCasesByLength {  2 usages  Lucas Anacleto *

    public GetVariables csvToArray = new GetVariables();  3 usages
    public   String[] data;
    public BinarySearchTree treeData;  4 usages

    public CreateCasesByLength() {  1 usage  Lucas Anacleto
        this.treeData = new BinarySearchTree(this::compareLength);
        this.data = csvToArray.readDataToArray();
        treeData.insertAll(data);
    }
    public int compareLength(String s1, String s2) {  1 usage  Lucas Anacleto *
        int length1 = csvToArray.getTamanhoSenha(s1);
        int length2 = csvToArray.getMesData(s2);
        return Integer.compare(length1, length2);
    }

    public String[] bestCase() {  1 usage  Lucas Anacleto
        return treeData.inOrderAscending();
    }
    public String[] mediumCase() {  1 usage  Lucas Anacleto
        return data;
    }
    public String[] worstCase() {  1 usage  Lucas Anacleto
        return treeData.inOrderDescending();
    }

}
```

A implementação da Árvore Binária de Busca (BST) em substituição ao uso de arrays trouxe vantagens significativas ao projeto. Primeiramente, a BST proporcionou uma gestão mais eficiente de memória, eliminando a necessidade de ordenar arrays repetidamente.

Além disso, sua capacidade de alocação dinâmica de elementos permitiu que se ajustasse automaticamente ao tamanho do dataset, garantindo flexibilidade e otimização do uso de recursos. Outro benefício crucial foi a capacidade da BST de manter a ordem dos dados de forma eficiente, possibilitando a obtenção dos dados em ordem crescente ou decrescente por meio de travessias in-order.

Em termos de complexidade, a nova implementação reduziu significativamente o custo computacional das operações, passando de uma complexidade de O(n log n) para O(n)

para a obtenção dos arrays do pior e melhor caso. Essa mudança não apenas melhorou a eficiência do projeto, mas também o tornou mais escalável para grandes volumes de dados, representando uma melhoria substancial em sua estrutura e desempenho.

2.2.3 Terceira estrutura (Tabela Hash)

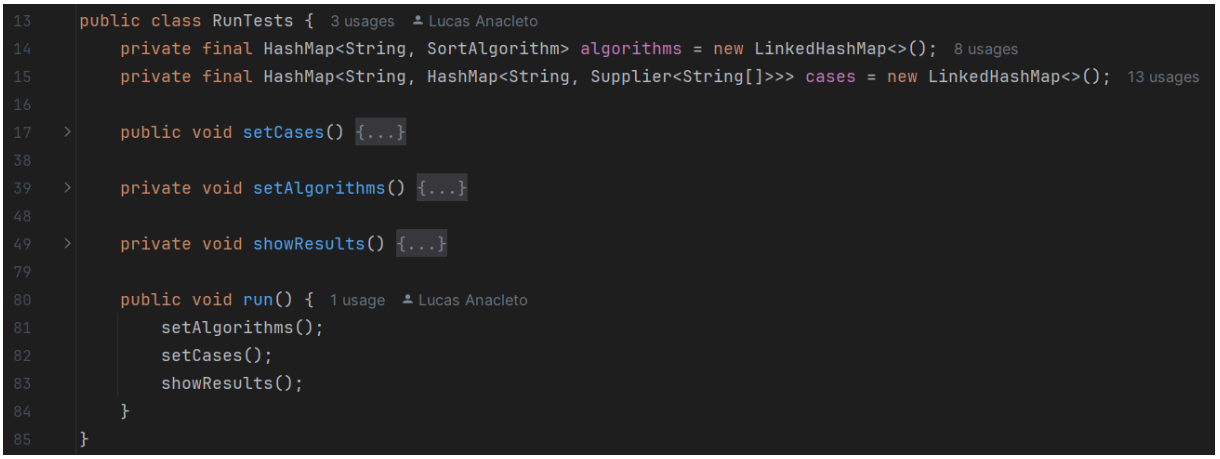
A adição de tabelas hash (dicionários) no contexto do serviço de execução de testes trouxe melhorias significativas à implementação anteriormente manual. Antes, o código realizava as operações de execução dos algoritmos e geração de logs manualmente, o que tornava o código menos legível e menos escalável. Com a introdução das tabelas hash, o código foi simplificado e tornou-se mais claro e modular.

Essa reestruturação resultou em ganhos significativos de eficiência e desempenho. A utilização das tabelas hash proporcionou um método mais eficiente para armazenar e acessar os algoritmos e casos de teste. Em vez de lidar com múltiplas estruturas de dados e repetir código para cada algoritmo e tipo de caso, agora o código utiliza um único mecanismo de indexação, tornando as operações mais rápidas e otimizadas.

Exemplo de código antes da alteração:

```
1   public class RunTests {  2 usages  Lucas Anacleto
2
3       CreateCasesByLength byLength = new CreateCasesByLength();  3 usages
4       CreateCasesByDate byDate= new CreateCasesByDate();  3 usages
5       CreateCasesByMonth byMonth = new CreateCasesByMonth();  3 usages
6
7       SelectionSort selectionSort = new SelectionSort();  9 usages
8       InsertionSort insertionSort = new InsertionSort();  9 usages
9       QuickSort quickSort = new QuickSort();  9 usages
10      MergeSort mergeSort = new MergeSort();  9 usages
11      CountingSort countingSort = new CountingSort();  9 usages
12      HeapSort heapSort = new HeapSort();  9 usages
13      QuickSortMedianaTres quickSortMedianaTres = new QuickSortMedianaTres();  9 usa
14
15
16  >   public void toLength() {...}
58
59  >   public void toMonth() {...}
101
102 >   public void toDate() {...}
144
145 }
```

Exemplo de código depois da alteração:



A organização lógica e eficiente das estruturas de dados facilita a adição de novos algoritmos e casos de teste, tornando o código mais flexível e fácil de manter. A modularidade proporcionada pelas tabelas hash permite que novas funcionalidades sejam facilmente integradas ao sistema, sem a necessidade de alterar o código existente de forma extensiva.

Essa mudança resultou em um código mais limpo, legível e escalável, melhorando significativamente a manutenibilidade e a eficiência do projeto como um todo. Ao simplificar e otimizar o processo de execução de testes, as tabelas hash contribuíram para uma implementação mais robusta e eficiente do sistema.

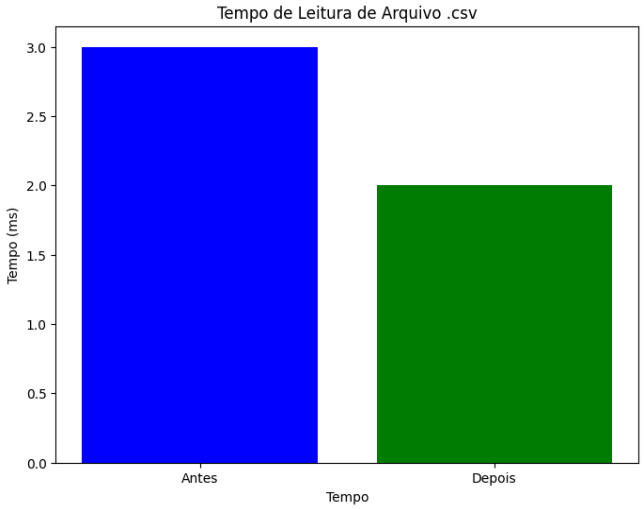
2.3 Análise de resultados

Nesta seção, abordaremos as análises de desempenho das alterações realizadas no código, com ênfase nas melhorias introduzidas pelas novas estruturas de dados implementadas. Com a implementação das novas estruturas de dados, observou-se uma melhoria substancial no tempo de execução em comparação com a versão anterior do código. Portanto, examinaremos em detalhes os resultados dessas mudanças, destacando os ganhos de desempenho e eficiência proporcionados pelas novas implementações.

A primeira mudança ocasionada pela implementação da fila no contexto de execução refere-se a uma significativa melhoria no tempo de execução do código. Anteriormente, o método para leitura de um arquivo .csv demandava cerca de 3ms para ser executado em uma chamada. No entanto, com a nova implementação utilizando a fila, esse tempo foi reduzido para 2ms. Vale ressaltar que este método é chamado várias vezes ao longo do código, o que resulta em um ganho ainda mais expressivo do que apenas 1ms.

Essa diminuição no tempo de execução é crucial, pois contribui para otimizar o desempenho global do programa, permitindo uma execução mais rápida e eficiente das operações envolvidas. Assim, a utilização da fila demonstrou ser uma escolha acertada, proporcionando benefícios tangíveis em termos de tempo de execução e eficiência do código.

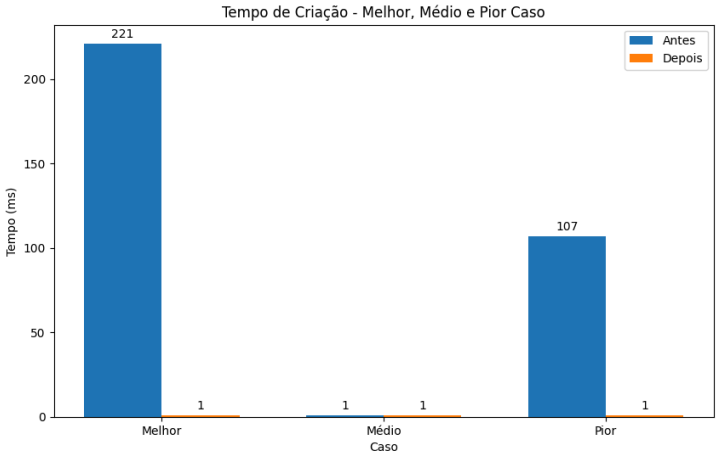
Abaixo temos uma representação gráfica de tempo referente às mudanças citadas.



A implementação da árvore binária de busca também trouxe um ganho significativo em comparação com o código anterior. No contexto da criação dos casos para análises de tamanho de senhas, a geração dos casos de melhor, médio e pior cenário demandava considerável tempo de execução: aproximadamente 221ms para o melhor caso, 1ms para o caso médio e 107ms para o pior caso. No entanto, com a utilização da árvore binária de busca, esses tempos foram drasticamente reduzidos para quase 0, chegando a cerca de 1ms para cada criação de caso.

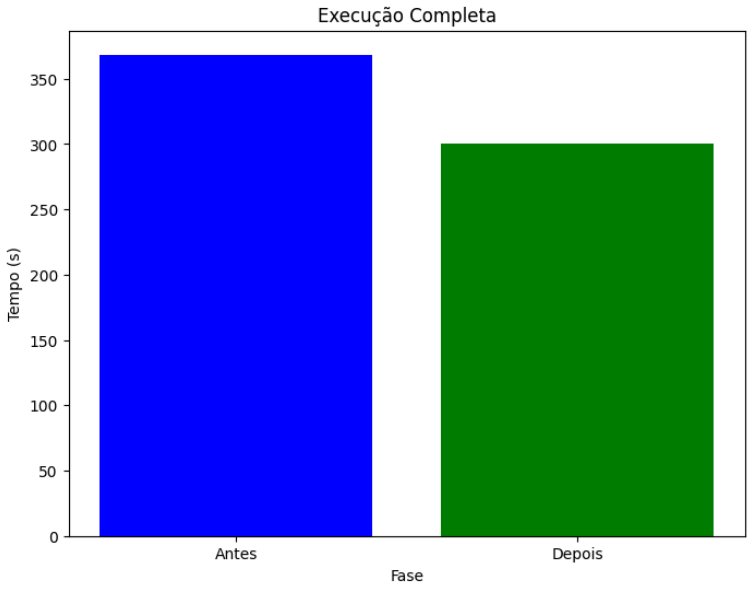
É importante ressaltar que estamos analisando apenas a geração dos casos para análises de tamanho de senha, e ainda há os contextos de data e mês. Portanto, é possível inferir que a melhoria proporcionada pela árvore binária de busca foi ainda mais efetiva, contribuindo para uma execução mais rápida e eficiente do código como um todo. Esses resultados evidenciam a importância e o impacto positivo das escolhas de implementação de estruturas de dados adequadas na otimização do desempenho do programa.

Abaixo temos uma representação gráfica de tempo referente às mudanças citadas.



Por fim, a implementação da tabela hash no código automatizou o processo de execução e resultou em tempos de execução ainda mais otimizados em comparação com as implementações anteriores. Anteriormente, o código levava cerca de 360 segundos para ser executado. No entanto, com as novas implementações, esse tempo foi reduzido em aproximadamente 60 segundos. Isso representa uma diferença significativa de aproximadamente 16.67% de melhoria no tempo de execução. Essa drástica redução ressalta o impacto positivo da introdução da tabela hash, evidenciando sua eficácia em agilizar e otimizar o processo de execução do código.

Abaixo temos uma representação gráfica de tempo referente às mudanças citadas.



3. CONCLUSÃO

Ao longo deste relatório, empreendemos uma análise detalhada das mudanças implementadas no código, visando melhorar sua eficiência e desempenho. Inicialmente, foram introduzidas novas estruturas de dados, como filas e tabelas hash, resultando em reduções significativas nos tempos de execução. A implementação de uma fila reduziu o tempo de execução em cada chamada, enquanto a introdução de uma tabela hash levou também a uma diminuição substancial do tempo total de execução.

Além disso, a adoção de uma árvore binária de busca para lidar com a criação de casos de melhor, médio e pior cenário demonstrou melhorias consideráveis, com os tempos de execução reduzidos para quase 0. Essas mudanças proporcionaram uma automação mais eficiente do processo, otimizando ainda mais o desempenho do sistema.

Não menos importante, as melhorias nos algoritmos de ordenação também desempenharam um papel crucial na otimização geral do código. Algoritmos como SelectionSort, InsertionSort, QuickSort, MergeSort, CountingSort, HeapSort e QuickSort com Mediana 3 tiveram seus tempos de execução consideravelmente reduzidos em diferentes cenários, refletindo uma abordagem abrangente para a otimização do código.

Portanto, podemos concluir que as melhorias cumulativas implementadas não apenas resultaram em um código mais eficiente e otimizado, mas também geraram um impacto positivo substancial no desempenho geral do sistema. A otimização contínua e a busca por melhorias são essenciais para manter a eficácia de um sistema em constante evolução.