UNIVERSIDADE ESTADUAL DA PARAÍBA

CENTRO DE CIÊNCIA E TECNOLOGIA- CCT

CIÊNCIA DA COMPUTAÇÃO

ALISSOM DA SILVA RODRIGUES

DJHONATAH WESLEY C. ALVES

MATEUS RUFINO FERREIRA

PROJETO 2: TEMA 2 - PASSWORDS

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

CAMPINA GRANDE

2024

SUMÁRIO

1. INTRODUÇÃO

Demos continuidade ao estudo iniciado na primeira etapa do projeto, onde foi realizada uma análise dos resultados da ordenação das mais de 600mil senhas por meio dos algoritmos pedidos. Os algoritmos utilizados foram: 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 conforme solicitado.

As três estruturas de dados que utilizamos foram a Fila a Árvore Binária de Busca e Tabelas Hash. esses três foram utilizados no código substituindo algumas situações que vamos mostrar mais adiante, de maneira geral eles nos ajudaram na execução mais rápida em comparação e o código ficou com menos linhas em alguns locais, apesar de agora ter algumas classes a mais.

2 ESTRUTURAS DE DADOS

Neste projeto, implementamos três novas estruturas de dados: Fila, Árvore Binária de Busca e Tabela Hash.

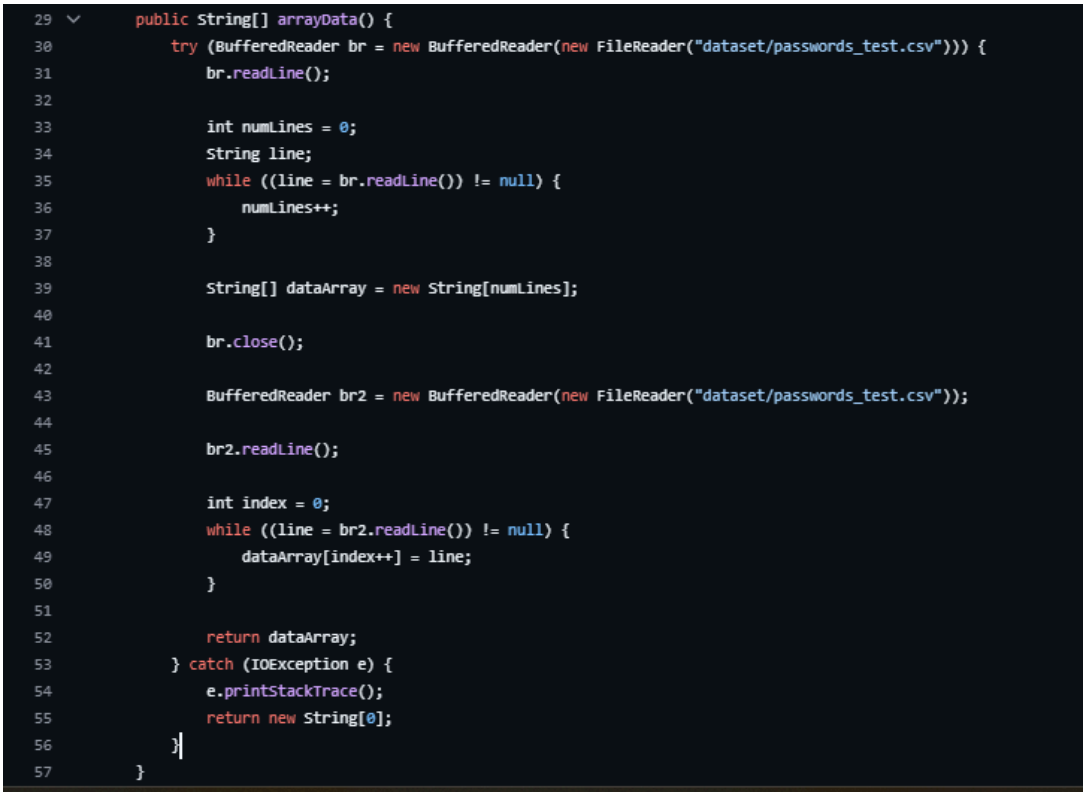
FILA é uma aplicação da logica FIFO (First In, First Out), é ideal para o processamento sequencial de lotes de dados, otimizando recursos e mantendo o fluxo contínuo.

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.

ABB (árvore binária de busca), uma estrutura de busca, inserção e remoção com no máximo dois filhos por nó, garantindo listas ordenadas dinamicamente.

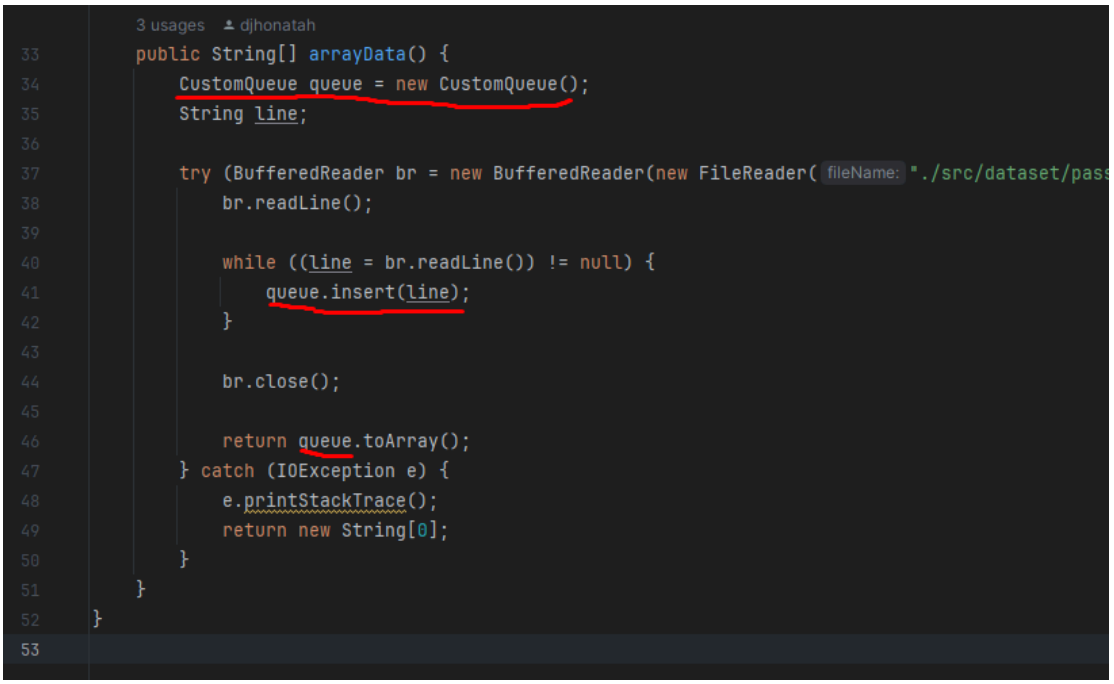
2.1 FILA

Uma das principais mudanças implementadas no projeto foi a substituição do uso de arrays pela estrutura de dados Fila (ou Queue) na classe *InputVar*, especificamente no método *arrayData* . Este método é responsável por ler o arquivo .csv orignial e armazenar esses dados em uma estrutura para processamento posterior. Substituímos os arrays que poderiam ser utilizados aqui, mas a realização de ações com os dados eram melhor do nosso ponto de vista quando utilizamos a fila. na Figura 1 mostramos o método arrayData que fizemos na parte 1 do projeto. Na figura 2 a última versão e a que foi feita no projeto 2.



```java
public String[] arrayData() {
    try (BufferedReader br = new BufferedReader(new FileReader("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("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];
    }
}
```

Figura 1 - método arrayData na classe InputVar na parte 1 do projeto



```java
3 usages  djhonatah
public String[] arrayData() {
    CustomQueue queue = new CustomQueue();
    String line;

    try (BufferedReader br = new BufferedReader(new FileReader( fileName: "./src/dataset/pass
        br.readLine();

        while ((line = br.readLine()) != null) {
            queue.insert(line);
        }

        br.close();

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

Figura 2 - método arrayData na classe InputVar na parte 2 do projeto

Podemos ver na Figura 1 que, 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, Figura 2, a estrutura de dados Fila, representada pela classe *CustomQueue*, foi implementada substituindo essa função. A fila permite a inserção dinâmica de elementos durante uma única leitura do arquivo fazendo com que otimizasse bastante na realização da leitura dos dados.

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. A lógica do FIFO que é utilizada pela Fila garante que a ordem de inserção dos elementos seja preservada, essencial para o correto processamento dos dados lidos.

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

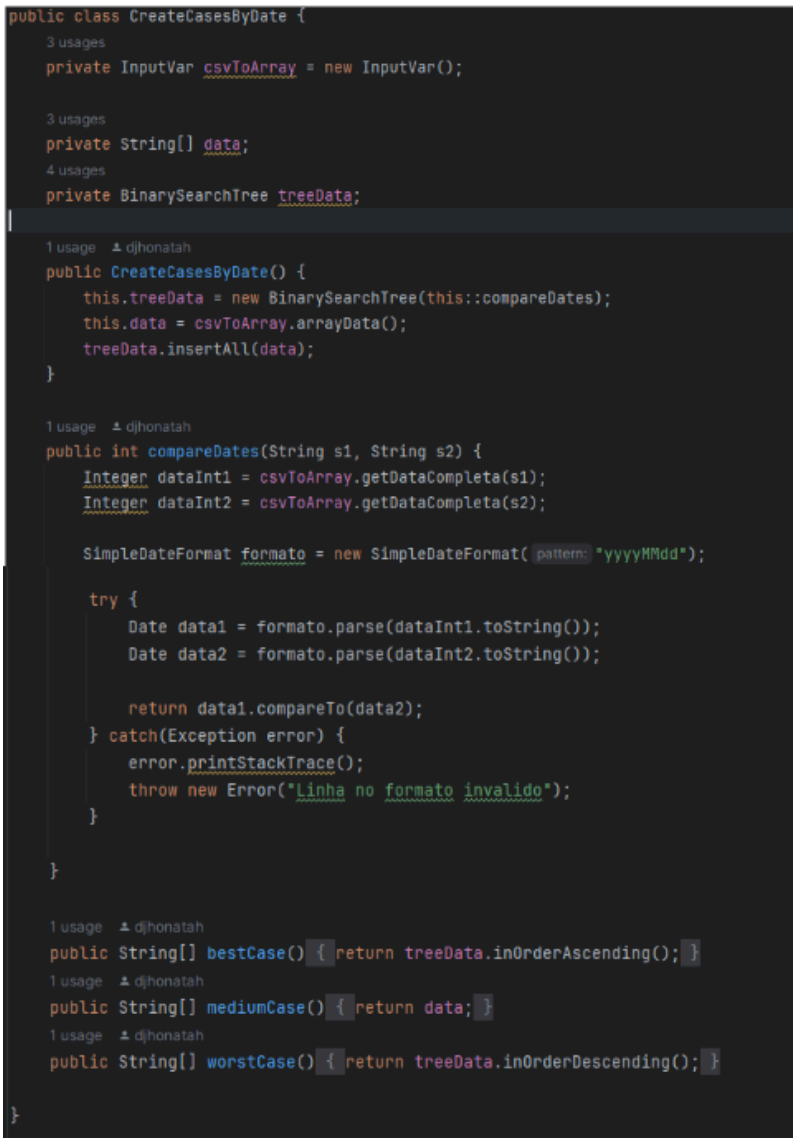
2.2 ABB (Árvore Binária de Busca)

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 no 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* próprio do Java. Para otimizar esse processo, substituímos a abordagem anterior pela implementação de uma Árvore Binária de Busca (ABB).

A ABB foi introduzida para inserir os dados uma única vez, permitindo a obtenção dos arrays em ordem crescente ou decrescente de maneira mais eficiente.

```java
public class CreateCasesByDate {
    InputVar csvToArray = new InputVar();
    String[] data = csvToArray.arrayData();

    Comparator<String> comparatorDateCrescent = Comparator.comparing((String str) -> {
        String[] piecesDate = str.split(",(?=(?:[^\"]*\"[^\"]*\")*[^\"]*$)", -1)[3].split(" ")[0].split("/");
        return Integer.parseInt(piecesDate[2]) * 10000 + Integer.parseInt(piecesDate[1]) * 100 + Integer.parseInt(piecesDate[0]);
    });
    Comparator<String> comparatorDateDecreasing = Comparator.comparing((String str) -> {
        String[] piecesDate = str.split(",(?=(?:[^\"]*\"[^\"]*\")*[^\"]*$)", -1)[3].split(" ")[0].split("/");
        return Integer.parseInt(piecesDate[2]) * 10000 + Integer.parseInt(piecesDate[1]) * 100 + Integer.parseInt(piecesDate[0]);
    }).reversed();

    public String[] melhorCaso() {
        String[] dataTransforming = Arrays.copyOf(data, data.length);
        Arrays.sort(dataTransforming, comparatorDateCrescent);
        return dataTransforming;
    }
    public String[] medioCaso() {
        return data;
    }
    public String[] piorCaso() {
        String[] dataTransforming = Arrays.copyOf(data, data.length);
        Arrays.sort(dataTransforming, comparatorDateDecreasing);
        return dataTransforming;
    }
}
```

Figura 3 -classe createCasesByDate na parte 1 do projeto



```java
public class CreateCasesByDate {
    3 usages
    private InputVar csvToArray = new InputVar();

    3 usages
    private String[] data;
    4 usages
    private BinarySearchTree treeData;

    1 usage  djhonatah
    public CreateCasesByDate() {
        this.treeData = new BinarySearchTree(this::compareDates);
        this.data = csvToArray.arrayData();
        treeData.insertAll(data);
    }

    1 usage  djhonatah
    public int compareDates(String s1, String s2) {
        Integer dataInt1 = csvToArray.getDataCompleta(s1);
        Integer dataInt2 = csvToArray.getDataCompleta(s2);

        SimpleDateFormat formato = new SimpleDateFormat( pattern: "yyyyMMdd");

        try {
            Date data1 = formato.parse(dataInt1.toString());
            Date data2 = formato.parse(dataInt2.toString());

            return data1.compareTo(data2);
        } catch(Exception error) {
            error.printStackTrace();
            throw new Error("Linha no formato invalido");
        }

    }

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

}
```

Figura 4 -classe createCasesByDate na parte 2 do projeto

A implementação da Árvore Binária de Busca (ABB) em substituição ao uso de arrays trouxe vantagens significativas ao projeto. Primeiramente, a ABB 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.

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. Na figura X mostramos uma tabela que compara um teste com os mesmos dados comparando a eficiência do projeto anterior e depois de finalizado.

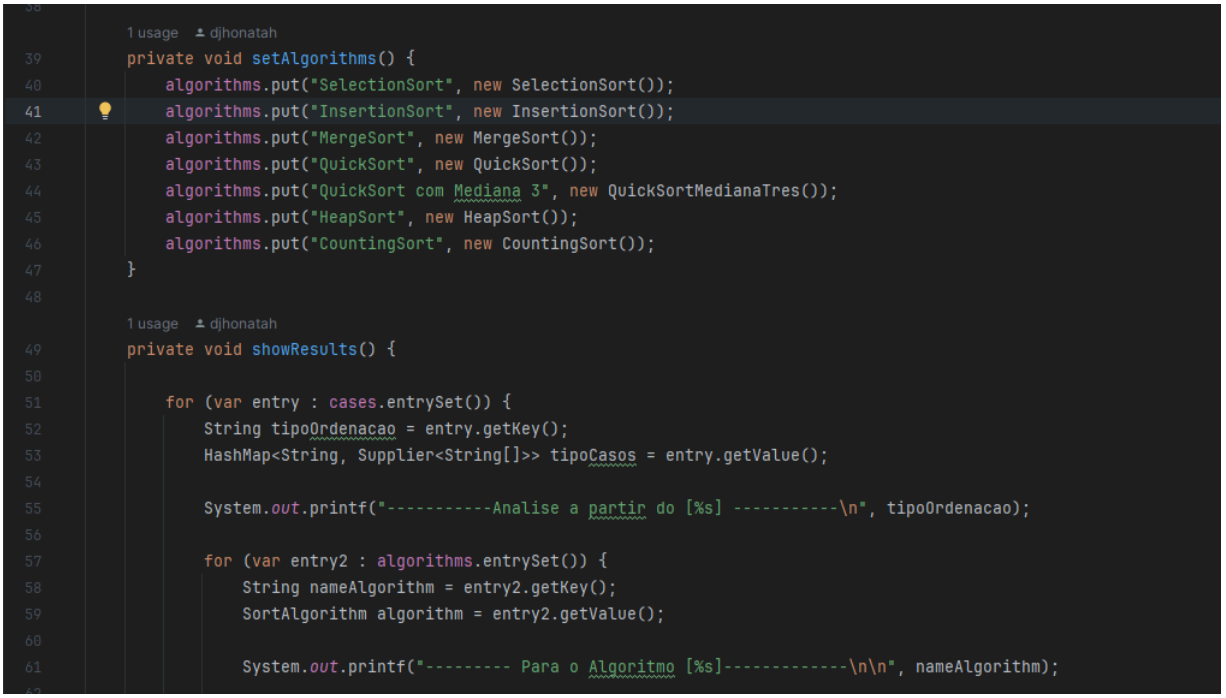
2.2 TABELA HASH

As Tabelas Hash (dicionários) foram utilizadas na execução dos testes, onde antes era feito um a um agora uma Tabelas Hash para executar. Antes, o código realizava as operações de execução dos algoritmos e geração de logs manualmente, o que tornava o código menos legível e com mais linhas. 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, desempenho e estética. 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.

```
  1     public class RunTests {
100         public void pordata() {
116
117             System.out.println("\nQuickSort P/ Data:");
118             quickSort.date(bestCaseByDate, "melhor caso");
119             quickSort.date(mediumCaseByDate, "médio caso");
120             quickSort.date(worstCaseByDate, "pior caso");
121
122             System.out.println("\nMergeSort P/ Data:");
123             mergeSort.date(bestCaseByDate, "melhor caso");
124             mergeSort.date(mediumCaseByDate, "médio caso");
125             mergeSort.date(worstCaseByDate, "pior caso");
126
127             System.out.println("\nCountingSort P/ Data:");
128             countingSort.date(bestCaseByDate, "melhor caso");
129             countingSort.date(mediumCaseByDate, "médio caso");
130             countingSort.date(worstCaseByDate, "pior caso");
131
132             System.out.println("\nHeapSort P/ Data:");
133             heapSort.date(bestCaseByDate, "melhor caso");
134             heapSort.date(mediumCaseByDate, "médio caso");
135             heapSort.date(worstCaseByDate, "pior caso");
136
137             System.out.println("\nQuickSort com Mediana 3 P/ Data:");
138             quickSortMedianaTres.date(bestCaseByDate, "melhor caso");
139             quickSortMedianaTres.date(mediumCaseByDate, "médio caso");
140             quickSortMedianaTres.date(worstCaseByDate, "pior caso");
141         }
142
143     }
```

Figura 5 - Trecho do método porData na classe RunTest, com 143 linhas



```
    1 usage  djhonatah
    private void setAlgorithms() {
        algorithms.put("SelectionSort", new SelectionSort());
        algorithms.put("InsertionSort", new InsertionSort());
        algorithms.put("MergeSort", new MergeSort());
        algorithms.put("QuickSort", new QuickSort());
        algorithms.put("QuickSort com Mediana 3", new QuickSortMedianaTres());
        algorithms.put("HeapSort", new HeapSort());
        algorithms.put("CountingSort", new CountingSort());
    }

    1 usage  djhonatah
    private void showResults() {

        for (var entry : cases.entrySet()) {
            String tipoOrdenacao = entry.getKey();
            HashMap<String, Supplier<String[]>> tipoCasos = entry.getValue();

            System.out.printf("-----------Analise a partir do [%s] -----------\n", tipoOrdenacao);

            for (var entry2 : algorithms.entrySet()) {
                String nameAlgorithm = entry2.getKey();
                SortAlgorithm algorithm = entry2.getValue();

                System.out.printf("--------- Para o Algoritmo [%s]------------\n\n", nameAlgorithm);
```

```
                SortAlgorithm algorithm = entry2.getValue();

                System.out.printf("--------- Para o Algoritmo [%s]------------\n\n", nameAlgorithm);

                for (var entry3 : tipoCasos.entrySet()) {
                    String caseName = entry3.getKey();
                    String[] caseArray = entry3.getValue().get();

                    algorithm.toggleSort(caseArray, caseName, tipoOrdenacao);
                }

                System.out.println();
            }

            System.out.println();
            System.out.println();
        }


    }

    1 usage  djhonatah
    public void run() {
        setAlgorithms();
        setCases();
        showResults();
    }
}
```

Figura 6 e 7 - Trechos da classe RunTestes agora utilizando Tabelas Hash e com agora 85 linhas

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 manutenção 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.

3 DISCUSSÃO E COMPARAÇÕES

As mudanças feitas no código foram analisadas para entender como elas melhoraram o desempenho, focando nas novas estruturas de dados usadas. Com essas estruturas, o tempo de execução melhorou bastante em relação à versão anterior do código. Por isso, vamos examinar os resultados dessas mudanças e destacar os ganhos em desempenho e eficiência trazidos pelas novas implementações.

A primeira melhoria com a nova fila foi uma redução significativa no tempo de execução. Antes, o método para ler um arquivo .csv levava cerca de 2260 ms para rodar. Com a fila, esse tempo caiu para 1295 ms. Essa leitura foi feita apenas para testes e comparação, e não faz parte do código final.

Essa redução no tempo é importante porque ajuda o programa a rodar mais rápido e de forma mais eficiente. Assim, usar a fila se mostrou uma boa escolha, trazendo benefícios claros em termos de velocidade e eficiência do código.

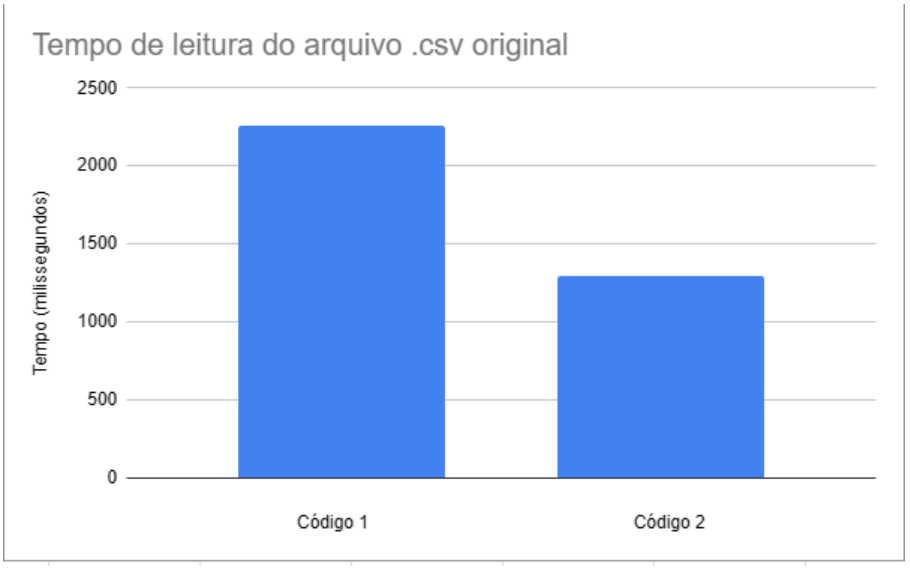


Tabela 1 - Comparação da leitura do arquivo passwords.csv antes e depois das estruturas de dados.

A implementação da árvore binária de busca também trouxe uma grande melhoria em relação ao código anterior. Na criação dos casos para analisar o tamanho de senhas, os tempos de execução eram significativos: cerca de 197 ms no melhor caso, 1 ms no caso médio e 110 ms no pior caso. Porém, com a árvore binária de busca, esses tempos caíram para quase zero,

ficando em torno de 1 ms para cada caso..

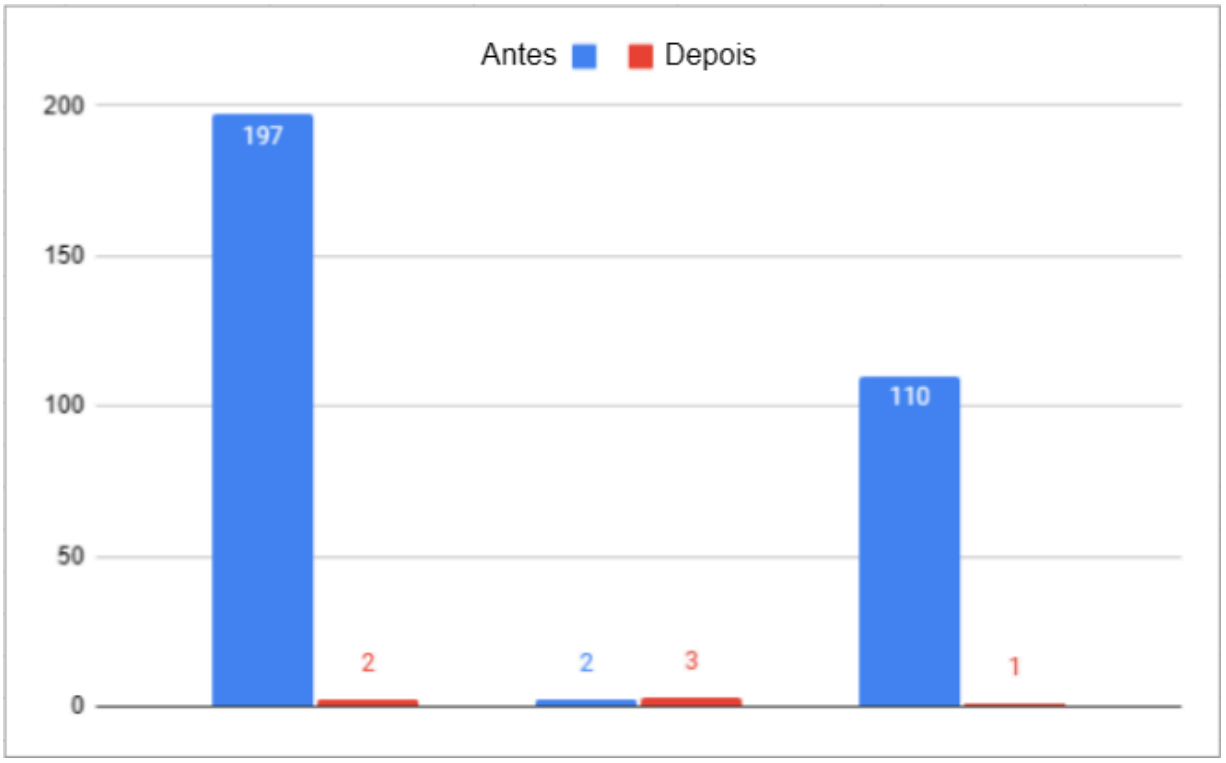


Tabela 2 - comparação entre os tempos de execução no melhor médio e pior caso.

Vale lembrar que estamos falando apenas da criação dos casos para o tamanho de senhas, ainda restam os contextos de data e mês. Isso mostra que a árvore binária de busca trouxe uma melhoria ainda maior, tornando o código mais rápido e eficiente. Esses resultados deixam claro como a escolha certa de estruturas de dados pode melhorar bastante o desempenho do programa.

A implementação da tabela hash no código automatizou o processo de execução e deixou os tempos de execução ainda mais rápidos em comparação com as versões anteriores. Antes, o código levava 484 segundos para ser executado. No entanto, com as novas mudanças, esse tempo foi reduzido em cerca de 88 segundos. Isso significa aproximadamente que 81,81% do tempo usado anteriormente, agora é salvo. Essa grande redução destaca o impacto positivo da tabela hash, mostrando como ela é eficiente para agilizar e melhorar o desempenho do código.

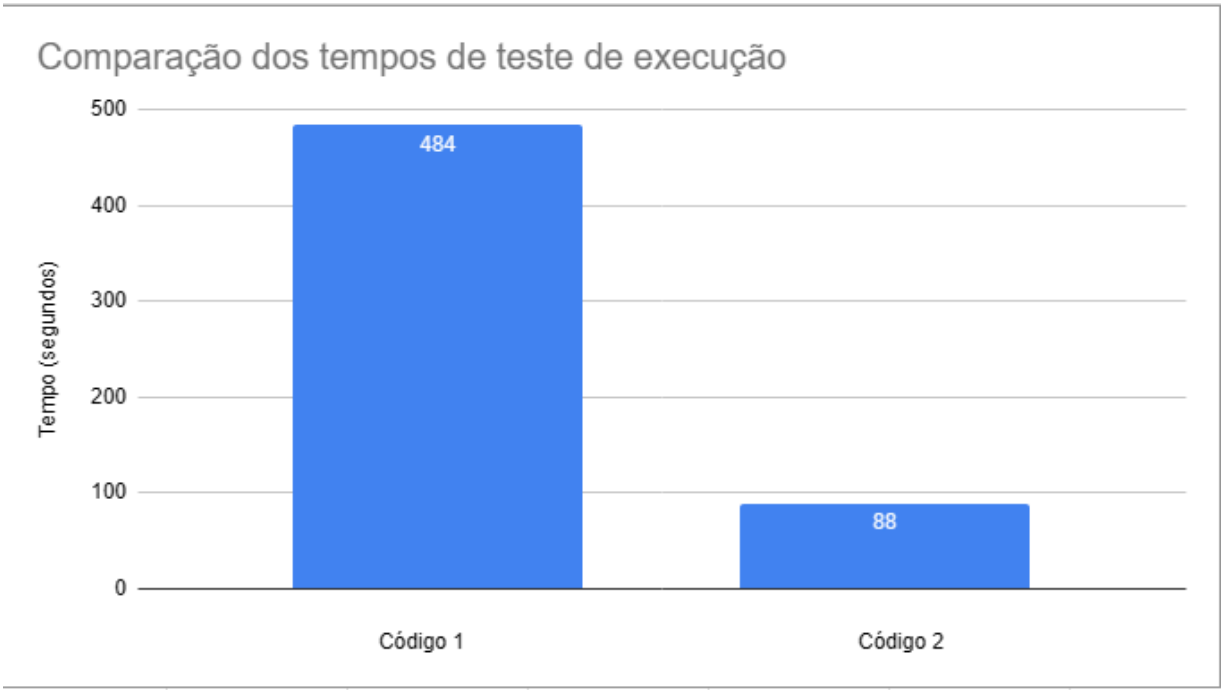


Tabela 3 - Comparação entre o código antigo (Código 1) e o atual (Código 2) do tempo de execução geral em um teste.

3. CONCLUSÃO

Neste relatório, fizemos uma análise detalhada das mudanças feitas no código, com o objetivo de melhorar sua eficiência e desempenho. Novas estruturas de dados, como filas e tabelas hash, foram introduzidas, o que reduziu bastante os tempos de execução. A fila diminuiu o tempo em cada chamada, enquanto a tabela hash ajudou a reduzir o tempo total de execução.

Além disso, a árvore binária de busca foi usada para criar casos de melhor, médio e pior cenário, trazendo melhorias significativas, com tempos de execução quase zerados. Essas mudanças tornaram o processo mais automático e ainda mais eficiente, otimizando o desempenho do sistema.

Também é importante destacar as melhorias nos algoritmos de ordenação, que tiveram um papel fundamental na otimização geral do código. Algoritmos como SelectionSort, InsertionSort, QuickSort, MergeSort, CountingSort, HeapSort e QuickSort com Mediana 3 tiveram seus tempos de execução reduzidos em vários cenários, mostrando um esforço abrangente para melhorar o código.

Assim, podemos concluir que as mudanças feitas não só deixaram o código mais rápido e eficiente, mas também tiveram um impacto muito positivo no desempenho geral do sistema. Continuar otimizando e buscando melhorias é essencial para manter um sistema eficiente e atualizado.