UNIVERSIDADE ESTADUAL DA PARAÍBA
CENTRO DE CIÊNCIA E TECNOLOGIA - CCT
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

GUILHERME SILVA HENRIQUES DE SOUSA
LUIZ GUSTAVO SOUZA LOURENÇO
WERLYS KAYAN SOARES DOS SANTOS

DOCENTE: FÁBIO LUIZ LEITE JÚNIOR

Relatório submetido no curso de Ciência
da Computação da Universidade Estadual
da Paraíba, na disciplina de LEDA,
correspondente ao período de 2024.2.

CAMPINA GRANDE
2024

1. RESUMO

Este relatório dá sequência ao estudo iniciado no Projeto 1 da disciplina de LEDA, onde realizamos uma comparação entre diversos algoritmos de ordenação aplicados a um conjunto de dados que contém mais de 600 mil senhas. Na etapa anterior, exploramos algoritmos como Selection Sort, Insertion Sort, Quick Sort, Merge Sort, Counting Sort, Heapsort e Quick Sort com Mediana 3, todos implementados utilizando apenas a estrutura de dados array.

Na fase atual do projeto, ampliamos nossa análise ao incorporar três novas estruturas de dados, além dos arrays, com o intuito de aprimorar a eficiência e a funcionalidade dos algoritmos de ordenação. O objetivo é investigar como a introdução de diferentes estruturas de dados pode impactar o desempenho dos algoritmos de ordenação em um grande volume de dados, proporcionando uma compreensão mais profunda de suas aplicações tanto práticas quanto teóricas. No contexto do projeto, utilizamos a **Árvore AVL** como uma das estruturas de dados para aprimorar a eficiência dos algoritmos de ordenação. A escolha de uma árvore **AVL (Adelson-Velsky and Landis)** foi motivada pela necessidade de manter a árvore balanceada automaticamente, garantindo que todas as operações de inserção, remoção e busca fossem realizadas em tempo **logarítmico**. utilizamos a estrutura de dados **Sequential List** para substituir o uso de arrays em parte da implementação dos algoritmos de ordenação. A **Sequential List** é uma estrutura de dados que armazena elementos de maneira sequencial, mas ao contrário dos arrays tradicionais, ela oferece uma flexibilidade maior, especialmente no que se refere à inserção e remoção de elementos. utilizamos a estrutura de dados **Fila (Queue)** como uma alternativa para otimizar e organizar o fluxo de dados em alguns algoritmos de ordenação e processos de manipulação dos dados de entrada. A **Fila** é uma estrutura de dados fundamental em ciência da computação, caracterizada pelo princípio de **FIFO (First In, First Out)**, ou seja, o primeiro elemento a ser inserido é o primeiro a ser removido. Essa característica a torna ideal para cenários onde o processamento de dados deve ocorrer de forma sequencial e em ordem.

2. DESENVOLVIMENTO

2.1. Estrutura de dados

Neste projeto, implementamos três novas estruturas de dados: Árvore AVL, Fila e Lista Sequencial. Cada uma delas possui características específicas que contribuem para a eficiência e a usabilidade dos algoritmos de ordenação.

- **Árvore AVL**: A Árvore AVL é uma estrutura de dados de árvore binária de busca que se auto equilibra. Isso significa que, após cada inserção ou remoção de um nó, a árvore ajusta sua altura para garantir que a diferença entre as alturas das subárvores esquerda e direita de qualquer nó seja no máximo um. Essa propriedade permite que as operações de busca, inserção e remoção sejam realizadas em tempo O(log n), tornando-a uma escolha ideal para cenários onde a eficiência na manipulação de dados é crucial.

- **Fila**: A Fila é uma estrutura de dados que segue a política FIFO (First In, First Out), onde o primeiro elemento a ser inserido é o primeiro a ser removido. Essa estrutura é especialmente útil para o processamento sequencial de dados, como em situações de gerenciamento de tarefas ou na leitura de arquivos, onde a ordem de chegada dos dados deve ser mantida. A fila otimiza o uso de recursos e garante um fluxo contínuo de informações.

- **Lista Sequencial**: A Lista Sequencial é uma estrutura de dados que armazena elementos em uma sequência linear, permitindo acesso rápido a qualquer elemento por meio de seu índice. Essa estrutura é simples de implementar e é ideal para cenários onde a quantidade de dados é conhecida e não muda frequentemente. A lista sequencial é útil para armazenar coleções de dados que requerem acesso rápido e fácil, como em algoritmos de ordenação que necessitam de acesso sequencial aos elementos.

Essas estruturas de dados foram escolhidas com base em suas características e na necessidade de otimizar a execução de diversas operações no projeto.

3. VANTAGENS

- **Árvore AVL**

A utilização da **Árvore AVL** no contexto do projeto oferece diversas vantagens, principalmente em termos de eficiência e desempenho. Como uma árvore binária de busca balanceada, ela garante que as operações de inserção e busca sejam realizadas em tempo O(log n), evitando que o tempo de execução se degrade para O(n), como ocorre em árvores desbalanceadas ou buscas sequenciais. Isso é particularmente importante ao lidar com grandes volumes de dados, como as senhas do projeto.

A árvore AVL também mantém seu balanceamento automaticamente após cada inserção ou remoção, impedindo que a árvore se torne degenerada em uma lista ligada, o que ocorreria se os dados fossem inseridos de forma ordenada sem balanceamento. Isso assegura que as operações permaneçam eficientes mesmo com dados inseridos em ordem ou quase ordenada.

Além disso, a altura balanceada da árvore AVL garante desempenho consistente para operações como busca e inserção, independentemente da ordem dos dados. Essa característica é crucial em algoritmos de ordenação, onde a ordem dos dados pode impactar negativamente a performance se não houver balanceamento.

Quando utilizada em algoritmos de ordenação, como o TreeSort, a árvore AVL permite uma ordenação eficiente, onde os elementos são inseridos na árvore e, em seguida, uma travessia em ordem gera os dados ordenados. Essa abordagem pode ser mais eficiente em termos de tempo e espaço quando comparada a algoritmos como SelectionSort ou InsertionSort, que possuem complexidade **$O(n^2)$** no pior caso.

Onde foi Aplicada a Árvore AVL no Código:

1. **Inserção e Balanceamento de Senhas**:
    - A Árvore AVL foi aplicada para armazenar as senhas de maneira ordenada. Ao invés de simplesmente usar uma lista ou array, onde a busca e a ordenação poderiam ser lentas, a AVL permite que as senhas sejam inseridas de maneira eficiente enquanto mantém a estrutura da árvore balanceada.
2. **Algoritmo de Ordenação (TreeSort)**:
    - A Árvore AVL foi utilizada como parte do processo de ordenação. A árvore AVL permite uma inserção eficiente das senhas, mantendo a ordem à medida que

novas senhas são inseridas. Após a inserção de todas as senhas na árvore, uma travessia in-order (em ordem crescente) é realizada para obter as senhas ordenadas. Essa técnica é equivalente ao algoritmo de ordenação TreeSort, que é eficiente quando a árvore está balanceada.

3. **Código Aplicado**: A implementação da **Árvore AVL** foi aplicada em métodos de inserção, onde as senhas são **inseridas** na árvore, e em métodos de travessia in-order para gerar os dados ordenados. A travessia em ordem é realizada de maneira eficiente devido ao balanceamento automático da árvore.

- **Fila**

A estrutura de dados **Fila** foi crucial no projeto para garantir que as senhas fossem processadas na ordem em que foram recebidas, sem a necessidade de reordenação. Isso foi especialmente útil durante a ordenação das senhas por atributos como tipo, mês ou data de criação, permitindo que os dados fossem processados de forma sequencial e eficiente. A Fila Também se destacou pela eficiência nas inserções e remoções, com complexidade constante (O(1)), permitindo que elementos fossem removidos do início e inseridos ao final sem a necessidade de reorganizar os dados.

Além disso, a Fila foi aplicada em algoritmos de ordenação estáveis, como o Counting Sort e o Bucket Sort, ajudando a manter a ordem relativa dos elementos iguais, preservando a estabilidade da ordenação. Ela também facilitou o controle de processamento por etapas, permitindo que o processamento fosse feito de forma incremental e organizada, sem sobrecarregar a memória ou o processamento, ao manipular um elemento por vez. Dessa forma, a Fila contribuiu para otimizar o fluxo de dados e melhorar a eficiência geral do processo de ordenação.

Onde foi Aplicada a Fila no Código:

1. **Inserção e Controle de Fluxo das Senhas:**

A **Fila** foi aplicada para armazenar as senhas de maneira sequencial enquanto elas são recebidas ou geradas. Ao invés de usar uma estrutura de dados como uma lista ou array, onde a

manipulação dos dados pode ser feita de forma menos controlada, a fila assegura que as senhas sejam processadas **na ordem em que foram inseridas**. Essa característica do FIFO (First In, First Out) é útil para garantir que o processamento das senhas seja feito de maneira ordenada, sem pular ou perder nenhum dado.

2. **Algoritmo de Processamento ou Ordenação em Lote:**

A Fila pode ser usada em algoritmos de **processamento em lote** ou até mesmo como parte de um **algoritmo de ordenação**. Por exemplo, ao receber as senhas, elas são inseridas na fila, e depois, uma vez que o algoritmo de ordenação ou qualquer outro processamento esteja pronto para ser realizado, as senhas são retiradas da fila **uma a uma** e processadas de forma sequencial. Esse processo mantém a ordem de chegada das senhas e pode ser combinado com outros algoritmos de ordenação ou processamento.

- **Lista Sequencial**

Lista Sequencial oferece várias vantagens, principalmente pela sua **simplicidade e eficiência**. Ela permite **acesso rápido e direto** aos elementos, já que os dados são armazenados em memória contígua e podem ser acessados em **O(1)**, o que é muito eficiente para operações de leitura e manipulação. Sua estrutura simples facilita a implementação, tornando-a uma opção prática para armazenar dados temporários, como senhas, durante o processo de ordenação. Em algoritmos de ordenação simples, como **SelectionSort** e **InsertionSort**, a lista sequencial é eficiente, pois permite fácil acesso aos elementos por índice, beneficiando-se da organização sequencial dos dados. Além disso, apesar de não ser a melhor escolha para inserções e remoções frequentes, em **algoritmos de ordenação** com dados pré-carregados, ela oferece **desempenho estável**, já que as operações de leitura e escrita são rápidas e não exigem manipulação complexa de ponteiros ou reestruturação de dados, como ocorre em outras estruturas mais sofisticadas.

Onde foi Aplicada a Lista Sequencial no Código:

1. **Armazenamento e Acesso às Senhas:**

A **Lista Sequencial** foi aplicada para armazenar as senhas de maneira simples e eficiente. Em vez de usar estruturas de dados mais complexas, como árvores ou listas encadeadas, a lista sequencial permite armazenar as senhas em uma **memória contígua**, o que torna o acesso aos dados muito rápido. A lista sequencial permite o **acesso direto aos**

elementos através de seus índices, tornando a manipulação de senhas mais ágil quando se precisa acessar ou modificar elementos em posições específicas.

2. **Algoritmo de Ordenação (SelectionSort, InsertionSort):**

A Lista Sequencial foi usada como estrutura de dados temporária para armazenar as senhas durante os **algoritmos de ordenação**. Em algoritmos como **SelectionSort** e **InsertionSort**, a lista sequencial facilita a **troca de elementos** e o acesso rápido aos índices para a comparação e reordenação das senhas. Esses algoritmos, que operam de maneira eficiente em listas sequenciais, tiram proveito da **simples estrutura de array** para realizar as operações de ordenação com facilidade.

4. CONCLUSÃO

Este relatório apresenta as mudanças realizadas no código com o objetivo de aprimorar sua eficiência e desempenho geral. A introdução de novas estruturas de dados e técnicas proporcionou melhorias significativas, otimizando os processos internos do sistema e garantindo maior agilidade em sua execução. Além disso, os ajustes realizados permitiram uma automação mais eficiente, assegurando uma operação mais estável e eficaz.

Essas mudanças contribuíram de forma substancial para a evolução do sistema, refletindo uma abordagem focada em desempenho. Como resultado, o sistema tornou-se mais robusto, preparado para lidar com cenários com maior quantidade de dados. Em resumo, as melhorias implementadas consolidaram uma base sólida para um desempenho superior.