O relatório abaixo foi elaborado de acordo com a atividade, seguindo a orientação dos padrões *generics* em java e utilizando os fundamentos apresentados pelo CORMEN. Neste relatório, faremos uma breve discussão, apresentando principais detalhes da logica utilizado nos códigos, incluindo comentários das principais funções a análise da complexidade de tempo (Big O).

**PROGRAMA 1 – INDICE REMISSIVO**

1. **DESCRIÇÃO GERAL DO CÓDIGO**

O projeto Índice Remissivo em Java implementa uma estrutura de dados que armazena palavras e suas respectivas ocorrências em arquivos de texto. O índice remissivo utiliza três estruturas de dados diferentes: Tabela Hash, Árvore AVL e Árvore Rubro-Negra. Cada uma dessas estruturas possui suas próprias características e vantagens, conforme detalhado abaixo.

1. **ESTRUTURA DO PROJETO**

Codigos\_Trabalhos CP/

└── Projetos\_Java/

└── Trabalho\_2\_ED2/

├── .vscode/

│ ├── launch.json

│ ├── settings.json

├── arquivos/

│ ├── texto1.txt

│ ├── texto2.txt

├── src/

│ ├── EstruturaDeDados.java

│ ├── IndiceRemissivo.java

│ ├── TabelaHash.java

│ ├── ArvoreAVL.java

│ ├── ArvoreRB.java

1. **FUNÇÕES PRINCIPAIS E COMPLEXIDADE DE TEMPO**

**Classe Principal: IndiceRemissivo.java**

**- processarArquivos(int x, EstruturaDeDados estrutura)**

- Complexidade de Tempo: O(n), onde n é o número de linhas nos arquivos de texto.

- Lê os arquivos de texto e processa cada linha para identificar as palavras que devem ser armazenadas no índice remissivo.

**- processarLinha(String linha, int numeroLinha, int x, EstruturaDeDados estrutura)**

- Complexidade de Tempo: O(m), onde m é o número de palavras em cada linha.

- Divide a linha em palavras, verifica o comprimento de cada palavra e insere as palavras válidas na estrutura de dados escolhida.

**Estrutura de Dados: TabelaHash.java**

**- inserir(String palavra, int linha)**

- Complexidade de Tempo: O(1) no melhor caso, O(n) no pior caso devido a colisões.

- Insere uma palavra e a linha correspondente na tabela hash, utilizando endereçamento aberto para resolver colisões.

**- buscar(String palavra)**

- Complexidade de Tempo: O(1) no melhor caso, O(n) no pior caso.

- Busca uma palavra na tabela hash e retorna as linhas onde ela ocorre.

**- remover(String palavra)**

- Complexidade de Tempo: O(1) no melhor caso, O(n) no pior caso.

- Remove uma palavra da tabela hash, marcando a entrada como livre.

**- imprimir()**

- Complexidade de Tempo: O(n), onde n é o tamanho da tabela hash.

- Imprime todas as palavras e suas ocorrências armazenadas na tabela hash.

Texto

Descrição gerada automaticamente

**Estrutura de Dados: ArvoreAVL.java**

**- inserir(String palavra, int linha)**

- Complexidade de Tempo: O(log n), onde n é o número de nós na árvore.

- Insere uma palavra na árvore AVL, garantindo que a árvore permaneça balanceada após a inserção.

**- buscar(String palavra)**

- Complexidade de Tempo: O(log n).

- Busca uma palavra na árvore AVL e retorna as linhas onde ela ocorre.

**- remover(String palavra)**

- Complexidade de Tempo: O(log n).

- Remove uma palavra da árvore AVL, realizando as rotações necessárias para manter o balanceamento.

**- imprimir()**

- Complexidade de Tempo: O(n), onde n é o número de nós na árvore.

- Imprime todas as palavras e suas ocorrências armazenadas na árvore AVL.

Texto

Descrição gerada automaticamente

Estrutura da classe ArvoreAVL

**Estrutura de Dados: ArvoreRB.java**

**- inserir(String palavra, int linha)**

- Complexidade de Tempo: O(log n), onde n é o número de nós na árvore.

- Insere uma palavra na árvore Rubro-Negra, garantindo que a árvore permaneça balanceada após a inserção.

**- buscar(String palavra)**

- Complexidade de Tempo: O(log n).

- Busca uma palavra na árvore Rubro-Negra e retorna as linhas onde ela ocorre.

**- remover(String palavra)**

- Complexidade de Tempo: O(log n).

- Remove uma palavra da árvore Rubro-Negra, realizando as rotações e ajustes de cores necessários para manter o balanceamento.

**- imprimir()**

- Complexidade de Tempo: O(n), onde n é o número de nós na árvore.

- Imprime todas as palavras e suas ocorrências armazenadas na árvore Rubro-Negra.

Texto

Descrição gerada automaticamente

Estrutura da classe ArvoreRB

**CONCLUSÃO**

O projeto de Índice Remissivo implementado em Java apresenta uma solução eficiente para armazenar e buscar palavras em arquivos de texto, utilizando três estruturas de dados diferentes: Tabela Hash, Árvore AVL e Árvore Rubro-Negra. Cada estrutura oferece suas próprias vantagens em termos de complexidade de tempo e adequação para diferentes tipos de operações e tamanhos de dados.

- Tabela Hash: Ideal para operações rápidas de inserção e busca em grandes conjuntos de dados, desde que as colisões sejam minimamente controladas.

- Árvore AVL: Garante operações balanceadas de busca, inserção e remoção, mantendo a altura da árvore em O(log n).

- Árvore Rubro-Negra: Similar à árvore AVL, mas com uma implementação de balanceamento mais flexível, utilizando propriedades de coloração.

**Texto

Descrição gerada automaticamente**

Execução do código e tela de seleção para criação do índice.

**PROGRAMA 2 – ARVORE B FUNCIONAL**

DESCRIÇÃO GERAL DO CÓDIGO

O projeto Árvore B em Java implementa uma estrutura de dados que suporta operações de inserção, exclusão e busca. A árvore B utiliza a memória secundária simulada (arquivos de texto) para armazenar seus nós. Além disso, foi desenvolvido um sistema de gerenciamento de usuários que permite inserir, buscar e excluir usuários, bem como visualizar a estrutura da árvore.

1. **ESTRUTURA DO PROJETO**

FUNÇÕES PRINCIPAIS E COMPLEXIDADE DE TEMPO

**Classe Principal: Main.java**

**- processarArquivos(int x, EstruturaDeDados estrutura)**

- Inicializa a árvore B e executa o menu de operações.

Complexidade de Tempo: O(1)

**- inserirUsuario(OperacoesArvoreB<Usuario> operacoesArvoreB)**

- Insere um novo usuário na árvore B. - Complexidade de Tempo: O(t log\_t n)

**- buscarUsuario(OperacoesArvoreB<Usuario> operacoesArvoreB)**

- Busca um usuário na árvore B base no ID fornecido pelo usuário..

- Complexidade de Tempo: O(t log\_t n)

**- excluirUsuario(OperacoesArvoreB<Usuario> operacoesArvoreB)**

- Exclui um usuário da árvore B base no ID fornecido pelo usuário..

- Complexidade de Tempo: O(t log\_t n)

**- visualizarEstrutura(ArvoreB<Usuario> arvoreB)**

- Visualiza a estrutura da árvore B, mostrando todos os nós e suas chaves..

- Complexidade de Tempo: O(n)

**Estrutura de Dados: ArvoreB.java**

**- ArvoreB(Class<T> clazz)**

- Construtor da árvore B, inicializa uma árvore vazia.

Complexidade de Tempo: O(1)

Parâmetros:

clazz: Classe dos objetos armazenados na árvore.

**- ArvoreB(NoArvoreB<T> raiz, Class<T> clazz)**

- Construtor da árvore B com nó raiz já existente.

Complexidade de Tempo: O(1)

Parâmetros:

raiz: Nó raiz da árvore.

clazz: Classe dos objetos armazenados na árvore.

- Getters e Setters

- Inclui métodos para obter e definir a raiz da árvore.

Complexidade de Tempo: O(1)

**Estrutura de Dados: NoArvoreB.java**

**- ArvoreB <T extends Comparable<T> & Serializable>**

Construtor do nó da árvore B. Inicializa um nó como folha ou interno e aloca espaço   para as chaves e filhos.

Complexidade de Tempo: O(1)

Parâmetros:

folha: booleano indicando se o nó é uma folha.

Inicializa os arrays de chaves e filhos com base no grau mínimo t.

- Getters e Setters

- Inclui métodos para obter e definir o número de chaves, a lista de chaves, a lista de

filhos e se o nó é folha.

- Complexidade de Tempo: O(1)

**Gerenciamento de Arquivos: GerenciadorArquivo.java**

salvarArvore(String nomeArquivo, NoArvoreB<T> raiz)

Salva a árvore B em um arquivo.

Complexidade de Tempo: O(n) onde n é o número de nós.

- Parâmetros:

- nomeArquivo: Nome do arquivo onde a árvore será salva.

- raiz: Nó raiz da árvore a ser salva.

- carregarArvore(String nomeArquivo, Class<T> clazz)

- Carrega a árvore B de um arquivo.

- Complexidade de Tempo: O(n)

- Parâmetros:

- nomeArquivo: Nome do arquivo de onde a árvore será carregada.

- clazz: Classe dos objetos armazenados na árvore.

**Classe Auxiliar: OperacoesArvoreB.java**

OperacoesArvoreB(ArvoreB<T> arvoreB)

Construtor das operações da árvore B, recebe uma instância de ArvoreB.

Complexidade de Tempo: O(1)

Parâmetros:

arvoreB: Instância da árvore B onde as operações serão realizadas.

inserir(T chave)

Insere uma chave na árvore B, lidando com a possível divisão de nós se necessário, e salva os dados.

Complexidade de Tempo: O(t log\_t n)

Parâmetros:

chave: Chave a ser inserida na árvore.

**inserirNaoCheio(NoArvoreB<T> x, T chave)**

Insere uma chave em um nó que não está cheio, localizando a posição correta para a chave.

Complexidade de Tempo: O(t log\_t n)

Parâmetros:

x: Nó onde a chave será inserida.

chave: Chave a ser inserida.

**dividirFilho(NoArvoreB<T> x, int i, NoArvoreB<T> y)**

Divide um nó filho quando ele está cheio, criando um novo nó e redistribuindo as chaves.

Complexidade de Tempo: O(t log\_t n)

Parâmetros:

x: Nó pai do nó cheio.

i: Índice do nó filho que será dividido.

y: Nó filho que será dividido.

**excluir(T chave)**

Exclui uma chave da árvore B, lidando com a possível fusão de nós se necessário, e salva os dados.

Complexidade de Tempo: O(t log\_t n)

Parâmetros:

chave: Chave a ser excluída.

**excluir(NoArvoreB<T> x, T chave)**

Exclui uma chave de um nó específico, garantindo que a árvore continue balanceada.

Complexidade de Tempo: O(t log\_t n)

Parâmetros:

x: Nó de onde a chave será excluída.

chave: Chave a ser excluída.

**buscarChave(NoArvoreB<T> x, T chave)**

Busca a posição de uma chave em um nó específico.

Complexidade de Tempo: O(t log\_t n)

Parâmetros:

x: Nó onde a busca será realizada.

chave: Chave a ser buscada.

**removerDeFolha(NoArvoreB<T> x, int idx)**

Remove uma chave de um nó folha, deslocando as chaves para manter a ordem.

Complexidade de Tempo: O(t log\_t n)

Parâmetros:

x: Nó folha de onde a chave será removida.

idx: Índice da chave a ser removida.

**removerDeNaoFolha(NoArvoreB<T> x, int idx)**

Remove uma chave de um nó que não é folha, lidando com o predecessor ou sucessor da chave.

Complexidade de Tempo: O(t log\_t n)

Parâmetros:

x: Nó de onde a chave será removida.

idx: Índice da chave a ser removida.

**getPredecessor(NoArvoreB<T> x, int idx)**

Obtém o predecessor de uma chave, que é a maior chave no subárvore à esquerda.

Complexidade de Tempo: O(t log\_t n)

Parâmetros:

x: Nó de onde será obtido o predecessor.

idx: Índice da chave cujo predecessor será obtido.

**getSucessor(NoArvoreB<T> x, int idx)**

Obtém o sucessor de uma chave, que é a menor chave no subárvore à direita.

Complexidade de Tempo: O(t log\_t n)

Parâmetros:

x: Nó de onde será obtido o sucessor.

idx: Índice da chave cujo sucessor será obtido.

juntar(NoArvoreB<T> x, int idx)

Junta dois nós filhos em um único nó, reduzindo o número de filhos de um nó pai.

Complexidade de Tempo: O(t log\_t n)

Parâmetros:

x: Nó pai cujos filhos serão juntados.

idx: Índice do nó filho que será juntado com o próximo.

preencher(NoArvoreB<T> x, int idx)

Preenche um nó que tem menos do que t chaves, garantindo que ele tenha pelo menos t chaves.

Complexidade de Tempo: O(t log\_t n)

Parâmetros:

x: Nó que será preenchido.

idx: Índice do nó filho que será preenchido.

pegarDoAnterior(NoArvoreB<T> x, int idx)

Pega uma chave do nó anterior (à esquerda) para preencher o nó atual.

Complexidade de Tempo: O(t log\_t n)

Parâmetros:

x: Nó que será preenchido.

idx: Índice do nó filho que pegará a chave do anterior.

pegarDoProximo(NoArvoreB<T> x, int idx)

Pega uma chave do próximo nó (à direita) para preencher o nó atual.

Complexidade de Tempo: O(t log\_t n)

Parâmetros:

x: Nó que será preenchido.

idx: Índice do nó filho que pegará a chave do próximo.

buscar(T chave)

Busca uma chave na árvore B, começando pela raiz.

Complexidade de Tempo: O(t log\_t n)

Parâmetros:

chave: Chave a ser buscada.

buscar(NoArvoreB<T> no, T chave)

Busca uma chave em um nó específico da árvore.

Complexidade de Tempo: O(t log\_t n)

Parâmetros:

no: Nó onde a busca será realizada.

chave: Chave a ser buscada.

salvarDados()

Salva a estrutura da árvore B em um arquivo binário, persistindo os dados.

Complexidade de Tempo: O(n)

salvarDadosTexto()

Salva a estrutura da árvore B em um arquivo de texto, facilitando a visualização.

Complexidade de Tempo: O(n)

salvarNoTexto(BufferedWriter writer, NoArvoreB<T> no, int nivel, String posicao)

Salva um nó específico em um arquivo de texto, incluindo detalhes sobre a posição do nó.

Complexidade de Tempo: O(n)

Parâmetros:

writer: Escritor do arquivo de texto.

no: Nó a ser salvo.

nivel: Nível do nó na árvore.

posicao: Posição do nó (Raiz, Folha Esquerda, Folha Direita, Interno).

noParaString(NoArvoreB<T> no)

Converte um nó em string para exibição.

Complexidade de Tempo: O(n)

Parâmetros:

no: Nó a ser convertido em string.

Classe Usuário: Usuario.java

Usuario(int id\_usuario, String login, String nome, String email, Date data\_nascimento, String foto)

Construtor do usuário, inicializa todos os campos.

Complexidade de Tempo: O(1)

Parâmetros:

id\_usuario: ID do usuário.

login: Login do usuário.

nome: Nome do usuário.

email: Email do usuário.

data\_nascimento: Data de nascimento do usuário.

foto: Foto do usuário.

Getters e Setters

Inclui métodos para obter e definir os atributos do usuário.

Complexidade de Tempo: O(1)

compareTo(Usuario other)

Compara este usuário com outro usuário com base no ID.

Complexidade de Tempo: O(1)

Parâmetros:

other: Outro usuário para comparação.

toString()

Converte o usuário em string para exibição.

Complexidade de Tempo: O(1)

**CONCLUSÃO**

O projeto de Árvore B implementado em Java apresenta uma solução eficaz para armazenar e buscar dados de usuários utilizando uma estrutura de árvore B que suporta inserção, exclusão e busca. A implementação utiliza memória secundária para persistência dos dados, garantindo que a árvore seja mantida entre execuções do programa.

Árvore B: Garante operações balanceadas de busca, inserção e remoção, mantendo a altura da árvore em O(log\_t n).

Abaixo segue a execução do código em um exemplo prático com as operações básicas e a visualização da estrutura da árvore em diferentes estados.

Exemplo de Execução do Código:

Interface gráfica do usuário, Texto

Descrição gerada automaticamente

Esta imagem mostra a estrutura do projeto em IntelliJ IDEA. No painel de projeto, vemos a organização das pastas e arquivos do projeto. A pasta src contém subpastas para documentos, estrutura, questoes, e classes como Main.java, ArvoreB.java, NoArvoreB.java, OperacoesArvoreB.java, e Usuario.java.

Tela de computador com texto preto sobre fundo branco

Descrição gerada automaticamente

Esta imagem exibe a execução inicial do programa onde o arquivo da árvore B não é encontrado, resultando na criação de uma nova árvore. O menu de operações é mostrado com opções para inserir, buscar, excluir usuários, visualizar a estrutura dos nós, ou sair do programa.

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

  A terceira imagem mostra a tentativa de buscar e excluir usuários inexistentes. O sistema informa que o usuário não foi encontrado para as operações de busca e exclusão.

Interface gráfica do usuário, Texto

Descrição gerada automaticamente

Nesta imagem, o usuário foi inserido com sucesso na árvore B. A estrutura da árvore B é visualizada e indica que a árvore está vazia, pois nenhuma operação de inserção foi executada antes da visualização.

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

Apresenta a estrutura da árvore B após a inserção dos três primeiros usuários. A

árvore é visualizada como um nó raiz com três chaves.Mostra a inserção do quarto usuário com ID 4. Detalhes do usuário são fornecidos, e a árvore B é salva, confirmando a inserção bem-sucedida do usuário.Exibe a estrutura da árvore B após a inserção do quarto usuário. A árvore é visualizada com um nó raiz interno e dois nós folha contendo os usuários.

Mostra a estrutura do projeto com foco nos arquivos do pacote estruturas. O código da classe ArvoreB está visível com as importações e a declaração da classe.

Exibe a execução do programa com a inserção de mais um usuário, seguida pela busca de um usuário inexistente (ID 2), resultando em um usuário não encontrado.

Mostra a inserção de mais um usuário com ID 4, seguida pela visualização da estrutura da árvore B. A árvore é visualizada com um nó raiz interno e dois nós folha contendo os usuários.   Mostra a exclusão de um usuário (ID 2), resultando em um usuário não encontrado. Em seguida, a exclusão de outro usuário (ID 1) é bem-sucedida, e a estrutura da árvore B é visualizada como vazia.

Exibe a execução do programa com a inserção de mais um usuário, seguida pela exclusão de um usuário e a visualização da estrutura da árvore B. A árvore é visualizada como vazia após a exclusão  Mostra a execução do programa com a inserção de mais um usuário, seguida pela visualização da estrutura da árvore B. A árvore é visualizada com um nó raiz contendo duas chaves.  Exibe a execução do programa com a inserção de mais um usuário, seguida pela visualização da estrutura da árvore B. A árvore é visualizada com um nó raiz interno e dois nós folha contendo os usuários.  Mostra a execução do programa com a exclusão de um usuário, seguida pela visualização da estrutura da árvore B. A árvore é visualizada com um nó raiz interno e dois nós folha contendo os usuários.