Trabalho Prático I – Pesquisa Externa

Estrutura de Dados II

Ben hur Samuel, Otávio Mapa e Matheus Gonçalves

28 de janeiro de 2025

Sumário

1 Introdução

1.1 Considerações iniciais

Este trabalho prático tem como objetivo realizar um estudo da complexidade de desempenho de diferentes métodos de pesquisa externa. Os métodos que foram implementados, são os que foram ensinados em sala de aula. Sendo eles: Acesso sequencial indexado, árvore binária de pesquisa adequada à memória externa, Árvore B e Árvore B*.

1.2 Ferramentas utilizadas

Algumas ferramentas foram utilizadas durante a criação deste projeto:

- Ambiente de desenvolvimento do código fonte: Visual Studio Code.
- Linguagem utilizada: C.
- Ambiente de desenvolvimento da documentação: Overleaf LATEX.

1.3 Especificações da máquina onde foram realizado os testes

- **Processador:** i5-13450HX.
- **Placa gráfica:** RTX 3050ti (laptop)
- **Memória principal:** 24gb RAM

1.4 Instruções de Compilação e Execução

O código-fonte deste projeto encontra-se acessível em GitHub. Ademais, inclui-se um script automatizado destinado à compilação e execução do algoritmo.

Aviso Importante

É necessário ter o `MinGW-w64` (versão 64 bits) instalado para compilar o código corretamente.

Para rodar o algoritmo manualmente, siga os passos abaixo a partir da raiz do projeto:

Primeiramente, navegue até a pasta **arquivos**. Em seguida, compile o código para gerar o(s) arquivo(s) necessário(s), depois, execute o arquivo gerado:

```
cd arquivos
gcc *.c -o qualquernome.exe
arquivos.exe gerar <nome> <quantidade> <ordem>
```

Também é possível rodar este algoritmo utilizando a função `ler` com o comando:

```
arquivos.exe ler <nome>
```

Parâmetros:

- **nome**: Nome do arquivo a ser gerado.
- **quantidade**: Número de itens a serem gerados.
- **ordem**: 1 (crescente), 2 (decrescente), 3 (aleatório).

Após executar os passos acima, volte à raiz do projeto e compile o código para a aplicação principal:

```
cd ..
gcc -m64 *.c -o app.exe -Wall
```

Agora, para realizar a pesquisa, utilize o seguinte comando:

```
app.exe <metodo> <quantidade> <ordem> <chave> [-P]
```

Parâmetros:

- **metodo**: Número inteiro de 1 a 4 representando o algoritmo de pesquisa a ser utilizado:
  - (1) Acesso sequencial indexado.
  - (2) Árvore binária de pesquisa adequada à memória externa.
  - (3) Árvore B.
  - (4) Árvore B*.
- **quantidade**: Número de itens a serem gerados.
- **ordem**: 1 (crescente), 2 (decrescente), 3 (aleatório).
- **chave**: Chave a ser pesquisada no arquivo considerado.
- **-P**: Argumento opcional que, quando incluído, faz com que as chaves de pesquisa dos registros sejam exibidas na tela.

Os scripts Python disponíveis no diretório podem ser utilizados para automatizar o processo de compilação. Esses scripts não interferem no funcionamento dos métodos de pesquisa, sendo encarregados exclusivamente de compilar, executar o programa e gerar arquivos de índices aleatórios. A seguir, apresenta-se um exemplo de um desses scripts acompanhado das instruções para sua execução.

```
import subprocess
import os

comando_base = 'teste.exe'
metodo = str(4)
quantidade = str(2000000)
ordem = str(2)
chave = str(int(quantidade) - 10)
imprimir = '-P'

subprocess.run(['gcc', '-m64', '*.c', '-o', comando_base, '-Wall'])
subprocess.run([comando_base, metodo, quantidade, ordem, chave, imprimir])
os.remove(comando_base)
```

Para executar o algoritmo mostrado na use:

```
python rodarprograma.py
```

2 Introdução ao código

O presente código foi elaborado com o objetivo de desenvolver uma aplicação eficaz para a manipulação e pesquisa de dados em arquivos binários (memória secundária), empregando diversos métodos distintos: Sequencial Indexado, Árvore Binária, Árvore B e Árvore B*.

2.1 Tipos

O arquivo *tipos.h* estabelece as estruturas de dados empregadas dentro do âmbito do projeto.

```
typedef struct {
    int chave;              // Chave de pesquisa
    long dado1;             // Dado 1 (inteiro longo)
    char dado2[5000];       // Dado 2 (cadeia de 5000 caracteres)
} Registro;

typedef struct {
    int transferencias;     // Quantidade de transferencias na pesquisa
    int comparacoes;        // Quantidade de comparacoes na pesquisa
    clock_t inicio;         // Tempo inicial da pesquisa
    double tempoExecucao;   // Tempo final de execucao (ms)
    clock_t inicioPP;       // Tempo inicial do pre-processamento
    int transferenciasPP;   // Quantidade de transferencias no pre-processamento
    int comparacoesPP;      // Quantidade de comparacoes no pre-processamento
    double tempoExecucaoPP; // Tempo final de pre-processamento (ms)
} Estatisticas;

typedef struct {
    int posicao;            // Posicao no arquivo
    int chave;              // Chave do registro
} Indice;

typedef struct {
    Registro registro;      // Registro armazenado no no
    int esquerda;           // Indice do filho esquerdo
    int direita;            // Indice do filho direito
} NoBinario;

typedef struct Pagina {
    short nFilhos;                      // Numero de filhos na pagina
    Registro registros[MM];             // Array de registros armazenados na pagina
    Apontador ponteiros[MM + 1];         // Ponteiros para outras paginas filhas
} Pagina;

#define MB 3                // Ordem da arvore B* (para paginas internas)
#define MMB (2 * MB)        // Capacidade maxima de chaves nas paginas internas
#define MB2 3               // Ordem para paginas externas da arvore B*
#define MMB2 (2 * MB2)      // Capacidade maxima de registros nas paginas externas

typedef int TipoChave;

typedef enum { Interna, Externa } TipoPagina;

typedef struct PaginaEstrela {
    TipoPagina Pt;              // Tipo de pagina (Interna ou Externa)
    union {
        struct {
            int ni;             // Numero de chaves na pagina interna
            TipoChave ri[MMB]; // Array de chaves na pagina interna
            struct PaginaEstrela *pi[MMB + 1]; // Ponteiros para paginas filhas
        } U0;
        struct {
            int ne;             // Numero de registros na pagina externa
            Registro re[MMB2]; // Array de registros na pagina externa
        } U1;
    } UU;
} PaginaEstrela;

// Funcao para exibir um registro
void lerRegistro(Registro* registro);
```

2.2 Estatísticas

O arquivo *estatisticas.c* implementa funções que são responsáveis pela coleta, análise e apresentação das estatísticas de execução durante as fases de pesquisa e pré-processamento do sistema

```
void inicializarEstatisticas(Estatisticas *estatisticas) {
    estatisticas->transferencias = 0;
```

```
    estatisticas->comparacoes = 0;
    estatisticas->tempoExecucao = 0.0;
    estatisticas->transferenciasPP = 0;
    estatisticas->comparacoesPP = 0;
    estatisticas->tempoExecucaoPP = 0;
    estatisticas->inicioPP = clock();
}

void inicializarTimerPesquisa(Estatisticas *estatisticas){
    estatisticas->inicio = clock();
}

void finalizarPreProcessamento(Estatisticas *estatisticas) {
    estatisticas->tempoExecucaoPP = (double)(clock() - estatisticas->inicioPP) * 1000 /
        CLOCKS_PER_SEC;
}

void finalizarEstatisticasPequisa(Estatisticas *estatisticas){
    estatisticas->tempoExecucao = (double)(clock() - estatisticas->inicio) * 1000 /
        CLOCKS_PER_SEC;
}

void printarEstatisticas(const Estatisticas *estatisticas) {
    printf("=== Estatisticas de Pre-Processamento ===\n");
    printf("Transferencias (Pre-Processamento): %d\n", estatisticas->transferenciasPP);
    printf("Comparacoes (Pre-Processamento): %d\n", estatisticas->comparacoesPP);
    printf("Tempo de Execucao (Pre-Processamento): %.2f ms (%.2f s)\n", estatisticas->
        tempoExecucaoPP, estatisticas->tempoExecucaoPP / 1000.0);

    printf("=== Estatisticas ===\n");
    printf("Transferencias: %d\n", estatisticas->transferencias);
    printf("Comparacoes: %d\n", estatisticas->comparacoes);
    printf("Tempo de Execucao: %.2f ms\n", estatisticas->tempoExecucao);
}
```

O arquivo contém as seguintes funções principais:

- **inicializarEstatisticas**: Inicializa as variáveis de estatísticas, como transferências e comparações, e começa a contagem do tempo de pré-processamento.
- **inicializarTimerPesquisa**: Inicia o cronômetro para medir o tempo de execução da pesquisa.
- **finalizarPreProcessamento**: Calcula o tempo total do pré-processamento após o término.
- **finalizarEstatisticasPesquisa**: Calcula o tempo total da pesquisa.
- **printarEstatisticas**: Exibe os valores das estatísticas de transferências, comparações e tempos de execução (para pré-processamento e pesquisa).

Essas funções são essenciais para monitorar o desempenho dos métodos de pesquisa

2.3 Arquivos

Funcionalidades para manipulação de arquivos binários (criação e leitura). A seguir, as funções implementadas no arquivo `Arquivos.c`:

- **lerIndicesDoArquivo**: Esta função recebe como parâmetros o nome de um arquivo e um vetor de índices. Ela abre o arquivo de índices e lê os índices de forma sequencial, armazenando-os no vetor fornecido.
- **gerarArquivoBinario**: Esta função é responsável por gerar um arquivo binário a partir dos parâmetros: caminho do arquivo, a quantidade de registros, a situação (que determina a ordem dos registros) e o nome do arquivo de índices (caso a situação seja aleatória). O arquivo gerado contém registros com a seguinte estrutura definida 2.1.

O formato dos dados no arquivo binário pode ser controlado pela situação, que pode ser ordenada de forma ascendente, descendente ou aleatória, dependendo do parâmetro `situacao`.

```
if (situacao == 3) {
    indicesAleatorios = (int *)malloc(quantidadeRegistros * sizeof(int));
    lerIndicesDoArquivo(arquivoIndices, indicesAleatorios, quantidadeRegistros);
}

for (int i = 0; i < quantidadeRegistros; i++) {
    int chave;

    if (situacao == 1) {
        chave = i;
    } else if (situacao == 2) {
        chave = quantidadeRegistros - i - 1;
    } else if (situacao == 3) {
        chave = indicesAleatorios[i];
    } else {
        fprintf(stderr, "Situacao invalida!\n");
        exit(1);
    }

    registro.chave = chave;
    registro.dado1 = chave + 5;
    snprintf(registro.dado2, sizeof(registro.dado2), "Dado 2 do registro %d", chave);

    fwrite(&registro, sizeof(Registro), 1, arquivo);
}
```

Em caso de ordem aleatório é usado o arquivo de indexes, para agilizar o processo. Que por sua vez é gerado pelo seguinte script python:

```
quantidades = [100, 200, 2000, 20000, 200000, 2000000]
situacoes = [1, 2, 3]
comando_base = 'arquivos.exe'


def gerar_indices_embaralhados(tamanho):
    indices = list(range(tamanho))
    random.shuffle(indices)
    return indices


def salvar_indices(tamanho, indices):
    nome_arquivo = f'./indices-aleatorios-{tamanho}.bin'
    with open(nome_arquivo, 'wb') as f:
        for indice in indices:
            f.write(struct.pack('i', indice))


for tamanho in quantidades:
    indices_embaralhados = gerar_indices_embaralhados(tamanho)
    salvar_indices(tamanho, indices_embaralhados)
```

- **lerArquivoBinario**: Esta função lê um arquivo binário já gerado.
- **main**: A função `main` gerencia a interação com o usuário, permitindo que ele escolha entre os comandos `gerar` ou `ler`.

2.4 Main

O programa aceita argumentos de linha de comando para escolher o método de pesquisa, a quantidade de registros, a ordem dos dados (ascendente, descendente ou aleatória), a chave a ser pesquisada e uma opção de depuração (-P) para informações detalhadas.

Os métodos de pesquisa são implementados de forma modular:

1. Sequencial Indexado

2. Árvore Binária
3. Árvore B
4. Árvore B*

A coleta de métricas de desempenho é feita por meio das funções **inicializarEstatisticas** e **finalizarEstatisticas** definidas em estatisticas.c 2.2, que registram comparações e acessos ao disco, permitindo avaliar a eficiência de cada método.

Para os métodos 1, 2, e 3 é utilizado um array de registros afim de diminuir a quantidade de acessos a memoria secundaria:

```
    Registro *registros = NULL;

    registros = (Registro *) malloc(quantidade * sizeof(Registro));
    estatisticas.transferenciasPP++;
```

3 Métodos de pesquisa

3.1 Acesso Sequencial Indexado

3.1.1 Pré-processamento

É um tipo de pesquisa que lê os registros em memória secundária e cria uma tabela de índices na memória principal. Com isso, realiza pesquisas por paginação utilizando essa tabela, reduzindo o número de acessos à memória secundária.

```
int pesquisaIndexada(
    Registro* registro,
    int tamanho,
    int ordem,
    FILE* arquivo,
    Estatisticas* estatisticas,
    int debug);
```

O **pesquisaIndexada** tem como parâmetros: o registro buscado, o tamanho do arquivo, a ordem de pesquisa, o arquivo a ser pesquisado, um ponteiro para a estrutura de estatísticas e a opção de debug. Se o registro for encontrado, a função retorna 1; caso contrário, retorna 0.

A quantidade de itens por página é definida por **ITENSPAGINA** no início do programa.

No início da pesquisa, o número de páginas é calculado por:

```
nPaginas = (tamanho + ITENSPAGINA - 1) / ITENSPAGINA;
```

Em seguida, a tabela de índices é alocada:

```
Indice* tabelaIndices = malloc(*nPaginas * sizeof(Indice));
```

Se a alocação falhar, a função retorna um erro:

```
if (tabelaIndices == NULL) {
    perror("Erro ao alocar memoria para tabela de indices");
    return NULL;
}
```

Se a alocação for bem-sucedida, o programa cria a variável temporária para armazenar registros:

```
Registro tempoRegistro;
```

Então, percorremos cada página, lendo o registro inicial de cada uma, atualizando as estatísticas de comparações e transferências:

```
for (int i = 0; i < *nPaginas; i++) {
    estatisticas->comparacoesPP++;
    _fseeki64(arquivo, i * ITENSPAGINA * sizeof(Registro), SEEK_SET);
    estatisticas->transferenciasPP++;
    fread(&tempoRegistro, sizeof(Registro), 1, arquivo);
    tabelaIndices[i].posicao = i * ITENSPAGINA;
    tabelaIndices[i].chave = tempoRegistro.chave;
}
```

Após a construção da tabela de índices, finalizamos as estatísticas de pré-processamento:

```
finalizarPreProcessamento(estatisticas);
```

Se o modo debug estiver ativado, imprimimos a tabela de índices:

```
if (debug) {
    printf("Numero de paginas: %d\n", nPaginas);
    printf("Tabela de Indices:\n");
    printf("Pagina\tPosicao\tChave\n");
    for (int i = 0; i < nPaginas; i++) {
        printf("%d\t%d\t%d\n", i + 1, tabelaIndices[i].posicao, tabelaIndices[i].chave);
    }
}
```

3.1.2 Pesquisa

Na pesquisa, verificamos em qual página o registro pode estar, atualizando as estatísticas a cada comparação:

```
for (int i = 0; i < nPaginas; i++) {
    estatisticas->comparacoes++;
    if ((ordem == 1 && tabelaIndices[i].chave > registro->chave) ||
        (ordem == 2 && tabelaIndices[i].chave < registro->chave)) {
        break;
    }
    pagina = i + 1;
}
```

Se nenhuma página for encontrada (`pagina == -1`), encerramos a busca:

```
if (pagina == -1) {
    free(tabelaIndices);
    return 0;
}
```

Caso contrário, realizamos uma busca sequencial na página identificada, atualizando as estatísticas:

```
_fseeki64(arquivo, tabelaIndices[pagina - 1].posicao * sizeof(Registro), SEEK_SET);
for (int i = 0; i < ITENSPAGINA; i++) {
    estatisticas->comparacoes++;
    estatisticas->transferencias++;
    if (fread(&tempoRegistro, sizeof(Registro), 1, arquivo) != 1) {
        break;
    }
    estatisticas->comparacoes++;
    if (tempoRegistro.chave == registro->chave) {
        *registro = tempoRegistro;
        free(tabelaIndices);
        return 1;
    }
}
```

Finalmente, liberamos a memória alocada para a tabela de índices:

```
    free(tabelaIndices);
```

Esse processo garante que a pesquisa seja otimizada em termos de acessos à memória secundária e que as estatísticas de desempenho sejam devidamente registradas.

3.2 Árvore Binária de Pesquisa

3.2.1 Pré-processamento

A criação de uma árvore binária utiliza a função `criarArvore`, que insere registros na estrutura, garantindo que cada nó tenha referências para os filhos esquerdo e direito. O número de comparações e transferências realizadas durante o processo é contabilizado nas estatísticas.

Inicialmente, o arquivo da árvore é aberto para escrita, e a inserção de cada nó segue os seguintes passos: - O registro é transferido para um nó temporário. - O nó é adicionado ao final do arquivo da árvore:

```
    _fseeki64(arquivoArvore, 0, SEEK_END);
    fwrite(&no, sizeof(NoBinario), 1, arquivoArvore);
```

- Os ponteiros da árvore são atualizados para conectar o novo nó:

```
    atualizaPonteiros(arquivoArvore, &no, estatisticas);
```

Durante o processo, a função `atualizaPonteiros` percorre a árvore para localizar o ponto correto de inserção, atualizando o ponteiro do nó pai para referenciar o novo nó.

As estatísticas de comparações e transferências no pré-processamento (`comparacoesPP` e `transferenciasPP`) são atualizadas em cada etapa.

3.2.2 Pesquisa

A função `buscarArvore` realiza a busca por um registro na árvore binária. A pesquisa começa pelo nó raiz (posição inicial no arquivo) e segue iterativamente até localizar o registro ou atingir uma folha. O caminho da busca é determinado comparando a chave do registro com a do nó atual.

O processo inclui: - Leitura do nó atual:

```
fread(&lido, sizeof(NoBinario), 1, arquivoArvore);
estatisticas->transferencias++;
```

- Comparação para verificar se o registro foi encontrado:

```
estatisticas->comparacoes++;
if (registro->chave == lido.registro.chave) {
    *registro = lido.registro;
    return 1; // Registro encontrado
}
```

- Caso contrário, o ponteiro é atualizado para seguir para o próximo nó (esquerda ou direita), com base na comparação.

Se o registro for encontrado, é retornado `1`. Caso contrário, `0` é retornado.

Leitura da Árvore Binária

A função `lerArvore` percorre todo o arquivo da árvore binária, imprimindo cada nó, sua chave e os ponteiros para os filhos esquerdo e direito. Este processo é útil para visualizar a estrutura gerada.

Os dados são apresentados no formato:

```
Posicao | Chave | Esquerda | Direita
---------------------------------
   0    |  15   |    -1    |   1
```

Este módulo garante a criação e manipulação de uma árvore binária eficiente, acompanhando métricas detalhadas de desempenho para análise.

3.3 Árvore B

3.3.1 Pré-processamento

A inicialização define a raiz como `NULL`, enquanto a inserção de registros organiza os elementos nas páginas. Quando necessário, ocorre a divisão (*split*) de páginas para garantir o balanceamento.

Inicialização A função abaixo inicializa a árvore, atribuindo `NULL` ao ponteiro raiz:

```
// Inicializa a arvore definindo a raiz como NULL
void inicializa(Apontador *arvore) {
    *arvore = NULL;
}
```

Inserção O processo de inserção é feito de forma recursiva para localizar a posição correta. A função `InsereNaPagina` insere o registro na página atual. Caso a página esteja cheia, ocorre a divisão.

```
/*-----------------------------------------------------------*/
/* Funcao: InsereNaPagina                                    */
/* Descricao: Insere um registro em uma pagina nao cheia,    */
/*            reorganizando elementos e ponteiros.           */
/* Parametros:                                               */
/*   apontador - Pagina onde o registro sera inserido.       */
/*   Reg - Registro a ser inserido.                          */
/*   ApDir - Ponteiro para o filho direito do registro.      */
/*   estatisticas - Armazena metricas de desempenho.         */
/*-----------------------------------------------------------*/
void InsereNaPagina(Apontador apontador, Registro Reg, Apontador ApDir, Estatisticas *
    estatisticas) {
    if (apontador == NULL) return;  // Verificacao de seguranca

    int k = apontador->nFilhos;     // Numero atual de filhos
    int NaoAchouPosicao = (k > 0);  // Flag para busca da posicao correta

    // Encontra a posicao de insercao (ordem decrescente)
    while (NaoAchouPosicao) {
        if (Reg.chave >= apontador->registros[k-1].chave) {
            NaoAchouPosicao = 0;  // Posicao encontrada
            break;
        }
        // Desloca registros e ponteiros para a direita
        apontador->registros[k] = apontador->registros[k-1];
        apontador->ponteiros[k+1] = apontador->ponteiros[k];
        k--;
        if (k < 1) NaoAchouPosicao = 0;  // Limite minimo
    }
    // Insere o registro e atualiza a contagem
    apontador->registros[k] = Reg;
    apontador->ponteiros[k+1] = ApDir;
    apontador->nFilhos++;
}
```

Divisão de Páginas (*Split*)

Quando uma página atinge sua capacidade máxima, ela é dividida. O registro central é promovido ao nó pai, enquanto as partes restantes são redistribuídas.

```
/*-------------------------------------------------------------------*/
/* Funcao: Ins                                                       */
/* Descricao: Funcao auxiliar recursiva para insercao. Realiza       */
/*            splits de paginas quando necessario.                   */
/* Parametros:                                                       */
/*   registro - Registro a ser inserido.                             */
/*   apontador - Pagina atual na recursao.                           */
/*   cresceu - Flag que indica se a arvore aumentou de altura.       */
/*   regRetorno - Registro promovido apos split.                     */
/*   apRetorno - Nova pagina criada apos split.                      */
/*   estatisticas - Contador de operacoes.                           */
/*   debug - Ativa logs para depuracao.                              */
/*-------------------------------------------------------------------*/
void Ins(Registro registro, Apontador apontador, short *cresceu, Registro *regRetorno,
    Apontador *apRetorno, Estatisticas *estatisticas, int debug) {
```

```
    long i = 0;
    Apontador apTemp;

    // Caso base: chegou a uma folha NULL (insercao requer nova pagina)
    if (apontador == NULL) {
        *cresceu = 1;
        *regRetorno = registro;
        *apRetorno = NULL;
        return;
    }

    // Encontra a posicao de insercao na pagina atual
    while (i < apontador->nFilhos && registro.chave > apontador->registros[i].chave) {
        estatisticas->comparacoesPP++;  // Atualiza estatisticas
        i++;
    }

    // Verifica se a chave ja existe (evita duplicatas)
    if (i < apontador->nFilhos && registro.chave == apontador->registros[i].chave) {
        estatisticas->comparacoesPP++;
        *cresceu = 0;  // Nao cresce a arvore
        return;
    }

    // Chamada recursiva para o filho adequado
    Ins(registro, apontador->ponteiros[i], cresceu, regRetorno, apRetorno, estatisticas, debug
        );

    if (!*cresceu) return;  // Nao houve split no nivel inferior

    // Se a pagina nao esta cheia, insere diretamente
    if (apontador->nFilhos < MM) {
        InsereNaPagina(apontador, *regRetorno, *apRetorno, estatisticas);
        *cresceu = 0;
        return;
    }

    // Split da pagina (overflow)
    apTemp = (Apontador) malloc(sizeof(Pagina));
    if (apTemp == NULL) {  // Verifica alocacao
        printf("Erro: Falha na alocacao de memoria em 'Ins'!\n");
        exit(1);
    }
    apTemp->nFilhos = 0;
    apTemp->ponteiros[0] = NULL;

    int meio = MM / 2;  // Ponto de divisao

    // Decide se o novo registro vai para a pagina esquerda ou direita
    if (i <= meio) {
        // Move o ultimo registro para a nova pagina e insere o novo registro na atual
        InsereNaPagina(apTemp, apontador->registros[MM - 1], apontador->ponteiros[MM],
            estatisticas);
        apontador->nFilhos--;
        InsereNaPagina(apontador, *regRetorno, *apRetorno, estatisticas);
    } else {
        InsereNaPagina(apTemp, *regRetorno, *apRetorno, estatisticas);
    }

    // Move registros da pagina cheia para a nova pagina
    for (int j = meio + 1; j < MM; j++) {
        InsereNaPagina(apTemp, apontador->registros[j], apontador->ponteiros[j + 1],
            estatisticas);
    }

    // Atualiza contagem de filhos e ponteiros
    apontador->nFilhos = meio;
    apTemp->ponteiros[0] = apontador->ponteiros[meio + 1];

    // Prepara o registro e a pagina para promocao
    *regRetorno = apontador->registros[meio];
    *apRetorno = apTemp;
}
```

3.3.2 Pesquisa

A busca é realizada comparando a chave desejada com os registros de cada nó. A busca recursiva segue para a subárvore correspondente até localizar a chave ou atingir uma folha.

```
/*--------------------------------------------------------*/
/* Funcao: pesquisaArvoreB                                */
/* Descricao: Busca um registro pela chave na arvore B.   */
/* Retorno: 1 se encontrado, 0 caso contrario.            */
/* Parametros:                                            */
/*   registro - Registro com a chave buscada (resultado). */
/*   apontador - Pagina atual na recursao.                */
/*   estatisticas - Contador de comparacoes.              */
/*--------------------------------------------------------*/
int pesquisaArvoreB(Registro *registro, Apontador apontador, Estatisticas *estatisticas) {
    if (apontador == NULL) {  // Arvore vazia
        printf("Erro: Arvore vazia!\n");
        exit(1);
    }

    long i = 1;
    estatisticas->comparacoes++;
    // Encontra a posicao onde a chave deveria estar
    while (i < apontador->nFilhos && registro->chave > apontador->registros[i-1].chave) {
        i++;
        estatisticas->comparacoes++;
    }

    estatisticas->comparacoes++;
    if (registro->chave == apontador->registros[i-1].chave) {  // Chave encontrada
        *registro = apontador->registros[i-1];
        estatisticas->comparacoes++;
        return 1;
    }

    estatisticas->comparacoes++;
    // Decide se busca no filho esquerdo ou direito
    if (registro->chave < apontador->registros[i-1].chave) {
        return pesquisaArvoreB(registro, apontador->ponteiros[i-1], estatisticas);
    } else {
        return pesquisaArvoreB(registro, apontador->ponteiros[i], estatisticas);
    }
}
```

Aqui está a documentação para a Árvore B*, seguindo o mesmo estilo da documentação da Árvore B fornecida:

3.4 Árvore B*

3.4.1 Pré-processamento

A inicialização da Árvore B* define a raiz como `NULL`, enquanto a inserção de registros organiza os elementos nas páginas. Quando necessário, ocorre a divisão (*split*) de páginas para garantir o balanceamento, com otimizações específicas para a Árvore B*.

Inicialização A função abaixo inicializa a árvore, atribuindo `NULL` ao ponteiro raiz:

```
// Inicializa a arvore B* definindo a raiz como NULL
void inicializa_b_estrela(ApontadorEstrela *Arvore) {
    *Arvore = NULL;
}
```

Inserção O processo de inserção é feito de forma recursiva para localizar a posição correta. A função `InsereNaPaginaExterna` insere o registro na página folha, enquanto a função `InsereNaPaginaInterna` insere chaves em páginas internas. Caso a página esteja cheia, ocorre a divisão.

```
/*--------------------------------------------------------*/
/* Funcao: InsereNaPaginaExterna                          */
/* Descricao: Insere registro em folha nao cheia.         */
```

```
/* Parametros:                                                */
/*   Ap - Pagina externa.                                     */
/*   Reg - Registro a inserir.                                */
/*   estatisticas - Contador de operacoes.                    */
/*-----------------------------------------------------------*/
void InsereNaPaginaExterna(ApontadorEstrela Ap, Registro Reg, Estatisticas *estatisticas) {
    int NaoAchouPosicao;
    int k = Ap->UU.U1.ne;  // Numero atual de registros
    NaoAchouPosicao = (k > 0);

    // Encontra posicao de insercao (ordem decrescente)
    while (NaoAchouPosicao) {
        estatisticas->comparacoesPP++;
        if (Reg.chave > Ap->UU.U1.re[k - 1].chave) {
            NaoAchouPosicao = 0;
            break;
        }
        // Desloca registros para a direita
        Ap->UU.U1.re[k] = Ap->UU.U1.re[k - 1];
        k--;
        if (k < 1) NaoAchouPosicao = 0;
    }
    // Insere o registro e atualiza contagem
    Ap->UU.U1.re[k] = Reg;
    Ap->UU.U1.ne++;
}

/*-----------------------------------------------------------*/
/* Funcao: InsereNaPaginaInterna                             */
/* Descricao: Insere chave em pagina interna nao cheia.      */
/* Parametros:                                               */
/*   Ap - Pagina interna.                                    */
/*   Reg - Chave a inserir.                                  */
/*   ApDir - Ponteiro para o filho direito.                  */
/*   estatisticas - Contador de operacoes.                   */
/*-----------------------------------------------------------*/
void InsereNaPaginaInterna(ApontadorEstrela Ap, TipoChave Reg, ApontadorEstrela ApDir,
    Estatisticas *estatisticas) {
    int NaoAchouPosicao;
    int k = Ap->UU.U0.ni;  // Numero atual de chaves
    NaoAchouPosicao = (k > 0);

    // Encontra posicao de insercao
    while (NaoAchouPosicao) {
        estatisticas->comparacoesPP++;
        if (Reg >= Ap->UU.U0.ri[k - 1]) {
            NaoAchouPosicao = 0;
            break;
        }
        // Desloca chaves e ponteiros para a direita
        Ap->UU.U0.ri[k] = Ap->UU.U0.ri[k - 1];
        Ap->UU.U0.pi[k + 1] = Ap->UU.U0.pi[k];
        k--;
        if (k < 1) NaoAchouPosicao = 0;
    }
    // Insere a chave e atualiza contagem
    Ap->UU.U0.ri[k] = Reg;
    Ap->UU.U0.pi[k + 1] = ApDir;
    Ap->UU.U0.ni++;
}
```

Divisão de Páginas (*Split*)

Quando uma página atinge sua capacidade máxima, ela é dividida. O registro central é promovido ao nó pai, enquanto as partes restantes são redistribuídas. A Árvore B* otimiza o processo de divisão, redistribuindo chaves entre irmãos antes de realizar o split.

```
/*-----------------------------------------------------------*/
/* Funcao: Ins_b_estrela                                     */
/* Descricao: Logica recursiva de insercao com splits        */
/*            otimizados para B*.                            */
/* Parametros:                                               */
/*   Reg - Registro a inserir.                               */
/*   Ap - Pagina atual.                                      */
/*   cresceu - Flag de crescimento.                          */
/*   RegRetorno - Chave promovida apos split.                */
```

```
/*   ApRetorno - Nova pagina apos split.                    */
/*   estatisticas - Contador de operacoes.                  */
/*----------------------------------------------------------*/
void Ins_b_estrela(Registro Reg, ApontadorEstrela Ap, short *cresceu, TipoChave *RegRetorno,
    ApontadorEstrela *ApRetorno, Estatisticas *estatisticas) {
    long i = 1;
    long j;
    ApontadorEstrela ApTemp;

    estatisticas->comparacoesPP++;
    if (Ap->Pt == Externa) {  // Insercao em folha
        *cresceu = 1;
        *RegRetorno = Reg.chave;
        *ApRetorno = NULL;

        // Encontra posicao na folha
        while (i < Ap->UU.U1.ne && Reg.chave > Ap->UU.U1.re[i - 1].chave) {
            estatisticas->comparacoesPP++;
            i++;
        }

        estatisticas->comparacoesPP++;
        if (Reg.chave == Ap->UU.U1.re[i - 1].chave) {  // Chave duplicada
            *cresceu = 0;
            return;
        }

        if (Reg.chave < Ap->UU.U1.re[i - 1].chave) i--;

        // Se a folha nao esta cheia, insere diretamente
        if (Ap->UU.U1.ne < MMB2) {
            InsereNaPaginaExterna(Ap, Reg, estatisticas);
            *cresceu = 0;
            return;
        }

        // Split da folha (overflow)
        ApTemp = (ApontadorEstrela)malloc(sizeof(PaginaEstrela));
        ApTemp->UU.U1.ne = 0;
        ApTemp->Pt = Externa;

        // Divide os registros entre a pagina atual e a nova
        if (i < MB2 + 1) {
            InsereNaPaginaExterna(ApTemp, Ap->UU.U1.re[MMB2 - 1], estatisticas);
            Ap->UU.U1.ne--;
            InsereNaPaginaExterna(Ap, Reg, estatisticas);
        } else {
            InsereNaPaginaExterna(ApTemp, Reg, estatisticas);
        }

        for (j = MB2 + 1; j <= MMB2; j++)
            InsereNaPaginaExterna(ApTemp, Ap->UU.U1.re[j - 1], estatisticas);

        // Prepara chave e pagina para promocao
        *RegRetorno = Ap->UU.U1.re[MB2].chave;
        Ap->UU.U1.ne = MB2;
        *ApRetorno = ApTemp;
        return;
    } else {  // Insercao em pagina interna
        estatisticas->comparacoesPP++;
        while (i < Ap->UU.U0.ni && Reg.chave > Ap->UU.U0.ri[i - 1]) {
            i++;
            estatisticas->comparacoesPP++;
        }

        estatisticas->comparacoesPP++;
        if (Reg.chave < Ap->UU.U0.ri[i - 1]) i--;

        // Chamada recursiva para o filho adequado
        Ins_b_estrela(Reg, Ap->UU.U0.pi[i], cresceu, RegRetorno, ApRetorno, estatisticas);

        if (!*cresceu) return;

        // Se a pagina interna nao esta cheia, insere a chave
        if (Ap->UU.U0.ni < MMB) {
            InsereNaPaginaInterna(Ap, *RegRetorno, *ApRetorno, estatisticas);
```

```
            *cresceu = 0;
            return;
        }

        // Split da pagina interna (overflow)
        ApTemp = (ApontadorEstrela)malloc(sizeof(PaginaEstrela));
        ApTemp->Pt = Interna;
        ApTemp->UU.U0.ni = 0;
        ApTemp->UU.U0.pi[0] = NULL;

        // Divide as chaves e ponteiros
        if (i < MB + 1) {
            InsereNaPaginaInterna(ApTemp, Ap->UU.U0.ri[MMB - 1], Ap->UU.U0.pi[MMB],
                estatisticas);
            Ap->UU.U0.ni--;
            InsereNaPaginaInterna(Ap, *RegRetorno, *ApRetorno, estatisticas);
        } else {
            InsereNaPaginaInterna(ApTemp, *RegRetorno, *ApRetorno, estatisticas);
        }

        for (j = MB + 2; j <= MMB; j++)
            InsereNaPaginaInterna(ApTemp, Ap->UU.U0.ri[j - 1], Ap->UU.U0.pi[j], estatisticas);

        // Atualiza contagem e ponteiros
        Ap->UU.U0.ni = MB;
        ApTemp->UU.U0.pi[0] = Ap->UU.U0.pi[MB + 1];
        *RegRetorno = Ap->UU.U0.ri[MB];
        *ApRetorno = ApTemp;
    }
}
```

3.4.2 Pesquisa

A busca é realizada comparando a chave desejada com os registros de cada nó. A busca recursiva segue para a subárvore correspondente até localizar a chave ou atingir uma folha.

```
/*--------------------------------------------------------------*/
/* Funcao: pesquisaBEstrela                                     */
/* Descricao: Busca um registro, diferenciando paginas          */
/*            internas (chaves) e externas (registros).         */
/* Retorno: 1 se encontrado, 0 caso contrario.                  */
/* Parametros:                                                  */
/*   x - Registro com a chave buscada.                          */
/*   Ap - Pagina atual na recursao.                             */
/*   estatisticas - Contador de operacoes.                      */
/*   debug - Ativa logs.                                        */
/*--------------------------------------------------------------*/
int pesquisaBEstrela(Registro *x, ApontadorEstrela Ap, Estatisticas *estatisticas, int debug)
    {
    int i;
    ApontadorEstrela Pag = Ap;

    if (Ap == NULL) return 0;  // Arvore vazia

    if (Pag->Pt == Interna) {  // Pagina interna (nao folha)
        i = 1;
        estatisticas->comparacoes++;
        // Busca binaria na pagina interna
        while (i < Pag->UU.U0.ni && x->chave > Pag->UU.U0.ri[i - 1]) {
            i++;
            estatisticas->comparacoes++;
        }
        estatisticas->comparacoes++;
        // Decide qual filho seguir
        if (x->chave < Pag->UU.U0.ri[i - 1])
            return pesquisaBEstrela(x, Pag->UU.U0.pi[i - 1], estatisticas, debug);
        else
            return pesquisaBEstrela(x, Pag->UU.U0.pi[i], estatisticas, debug);
    } else {  // Pagina externa (folha)
        i = 1;
        estatisticas->comparacoes++;
        // Busca binaria na folha
        while (i < Pag->UU.U1.ne && x->chave > Pag->UU.U1.re[i - 1].chave) {
            i++;
```

```
            estatisticas->comparacoes++;
        }
        estatisticas->comparacoes++;
        if (x->chave == Pag->UU.U1.re[i - 1].chave) {  // Chave encontrada
            *x = Pag->UU.U1.re[i - 1];
            return 1;
        } else {
            return 0;
        }
    }
}
```

Liberação de Memória

A liberação da memória utilizada pela árvore é feita pela função `liberaArvoreBEstrela`, que percorre recursivamente a árvore e libera a memória de cada nó. Isso é feito tanto para páginas internas quanto externas.

```
void liberaArvoreBEstrela(ApontadorEstrela Arvore) {
    if (Arvore == NULL) return;

    if (Arvore->Pt == Externa) {
        free(Arvore);
    } else {
        for (int i = 0; i <= Arvore->UU.U0.ni; i++) {
            liberaArvoreBEstrela(Arvore->UU.U0.pi[i]);
        }
        free(Arvore);
    }
}
```

Essa função ajuda a evitar vazamentos de memória durante a execução de programas que utilizam a árvore B*.

4 Desafios

O principal desafio enfrentado durante o desenvolvimento do sistema foi a recorrência de erros de *segmentation fault*, um problema principalmente associado à manipulação de ponteiros. Ademais, à medida que o volume de registros aumentava, o desempenho de todos os métodos era comprometido devido às restrições impostas pelo uso do compilador MinGW32. Este ambiente de 32 bits revelava-se inadequado para o manuseio de arquivos de grande porte, sendo que arquivos de 2000000 pesando aproximadamente **9,7GB** frequentemente resultavam em travamentos. A solução adotada foi a atualização para a versão de 64 bits do MinGW, juntamente com o uso da flag **-m64** durante a compilação. Essa alteração permitiu a alocação de mais memória e, por conseguinte, o manuseio de arquivos significativamente maiores, eliminando os problemas de desempenho e estabilidade. No entanto, com a alocação adicional de memória, começamos a observar, em alguns métodos, limitações de memória durante a inserção de páginas nas árvores, mesmo quando uma das máquinas possuía 24GB de memória RAM (memória principal). A solução encontrada consistiu em maximizar a quantidade de memória RAM disponível, permitindo que o algoritmo pudesse alocar espaço de forma eficiente.

5 Metricas

Todas as métricas foram calculadas com base na execução de algoritmo de testes, onde pode ser visualizado o valor de cada interação nesta tabela excel (Nessário e-mail institucional). As unidades de tempo estão todas em milissegundos (ms)

Tabela 1: Método 1 - Tamanho 100

Ordem	Média ComparaçõesPP	Média TransferênciasPP	Média TempoPP	Média Comparações	Média Transferências	Média Tempo
1	17.0	17.0	0.0	19.0	3.0	0.0
2	17.0	17.0	0.0	20.67	4.0	0.0

Tabela 2: Método 2 - Tamanho 100

Ordem	Média ComparaçõesPP	Média TransferênciasPP	Média TempoPP	Média Comparações	Média Transferências	Média Tempo
1	5149.0	5051.0	21.0	101.0	51.0	8.33
2	5149.0	5051.0	20.33	99.0	50.0	7.67
3	883.0	785.0	6.33	14.33	7.67	8.67

5.1 Tamanho: 100

5.2 Tamanho: 200

Tabela 5: Método 1 - Tamanho 200

Ordem	Média ComparaçõesPP	Média TransferênciasPP	Média TempoPP	Média Comparações	Média Transferências	Média Tempo
1	34.0	34.0	0.0	31.0	5.0	0.0
2	34.0	34.0	0.0	29.0	4.0	0.0

Tabela 6: Método 2 - Tamanho 200

Ordem	Média ComparaçõesPP	Média TransferênciasPP	Média TempoPP	Média Comparações	Média Transferências	Média Tempo
1	20299.0	20101.0	76.0	201.0	101.0	6.67
2	20299.0	20101.0	81.67	199.0	100.0	8.0
3	1934.0	1736.0	10.0	18.33	9.67	7.33

Tabela 7: Método 3 - Tamanho 200

Ordem	Média ComparaçõesPP	Média TransferênciasPP	Média TempoPP	Média Comparações	Média Transferências	Média Tempo
1	2292.0	1.0	1.67	18.0	0.0	0.0
2	200.0	1.0	1.0	18.67	0.0	0.0
3	1117.0	1.0	1.0	16.33	0.0	0.0

Tabela 8: Método 4 - Tamanho 200

Ordem	Média ComparaçõesPP	Média TransferênciasPP	Média TempoPP	Média Comparações	Média Transferências	Média Tempo
1	4615.0	1.0	1.67	13.0	0.0	0.0
2	3260.0	1.0	1.0	12.67	0.0	0.0
3	3725.0	1.0	1.67	14.33	0.0	0.0

Tabela 3: Método 3 - Tamanho 100

Ordem	Média ComparaçõesPP	Média TransferênciasPP	Média TempoPP	Média Comparações	Média Transferências	Média Tempo
1	962.0	1.0	0.33	14.0	0.0	0.0
2	100.0	1.0	0.67	12.0	0.0	0.0
3	484.0	1.0	0.67	16.67	0.0	0.0

Tabela 4: Método 4 - Tamanho 100

Ordem	Média ComparaçõesPP	Média TransferênciasPP	Média TempoPP	Média Comparações	Média Transferências	Média Tempo
1	1966.0	1.0	1.0	10.67	0.0	0.0
2	1433.0	1.0	0.67	11.67	0.0	0.0
3	1650.0	1.0	0.33	12.67	0.0	0.0

5.3 Tamanho: 2000

Tabela 9: Método 1 - Tamanho 2000

Ordem	Média ComparaçõesPP	Média TransferênciasPP	Média TempoPP	Média Comparações	Média Transferências	Média Tempo
1	334.0	334.0	2.0	181.0	5.0	0.33
2	334.0	334.0	2.0	179.0	4.0	0.67

Tabela 10: Método 2 - Tamanho 2000

Ordem	Média ComparaçõesPP	Média TransferênciasPP	Média TempoPP	Média Comparações	Média Transferências	Média Tempo
1	2002999.0	2001001.0	9499.33	2001.0	1001.0	12.33
2	2002999.0	2001001.0	9463.33	1999.0	1000.0	12.33
3	29097.0	27099.0	179.67	23.0	12.0	7.0

Tabela 11: Método 3 - Tamanho 2000

Ordem	Média ComparaçõesPP	Média TransferênciasPP	Média TempoPP	Média Comparações	Média Transferências	Média Tempo
1	35376.0	1.0	18.0	25.67	0.0	0.0
2	2000.0	1.0	17.0	26.67	0.0	0.0
3	16360.0	1.0	21.0	20.67	0.0	0.0

Tabela 12: Método 4 - Tamanho 2000

Ordem	Média ComparaçõesPP	Média TransferênciasPP	Média TempoPP	Média Comparações	Média Transferências	Média Tempo
1	68007.0	1.0	15.67	19.67	0.0	0.0
2	42192.0	1.0	14.67	17.0	0.0	0.0
3	51792.0	1.0	15.33	17.67	0.0	0.0

5.4 Tamanho: 20000

Tabela 13: Método 1 - Tamanho 20000

Ordem	Média ComparaçõesPP	Média TransferênciasPP	Média TempoPP	Média Comparações	Média Transferências	Média Tempo
1	3334.0	3334.0	19.33	1681.0	5.0	1.67
2	3334.0	3334.0	18.67	1679.0	4.0	1.67

Tabela 14: Método 2 - Tamanho 20000

Ordem	Média ComparaçõesPP	Média TransferênciasPP	Média TempoPP	Média Comparações	Média Transferências	Média Tempo
3	373829.0	353831.0	2812.33	33.0	17.0	8.67

Tabela 15: Método 3 - Tamanho 20000

Ordem	Média ComparaçõesPP	Média TransferênciasPP	Média TempoPP	Média Comparações	Média Transferências	Média Tempo
1	480012.0	1.0	166.0	34.33	0.0	0.0
2	20000.0	1.0	158.67	33.67	0.0	0.0
3	211536.0	1.0	169.0	26.33	0.0	0.0

Tabela 16: Método 4 - Tamanho 20000

Ordem	Média ComparaçõesPP	Média TransferênciasPP	Média TempoPP	Média Comparações	Média Transferências	Média Tempo
1	895096.0	1.0	172.0	24.0	0.0	0.0
2	530443.0	1.0	154.67	22.0	0.0	0.0
3	656235.0	1.0	172.33	21.67	0.0	0.0

5.5 Tamanho: 200000

Tabela 17: Método 1 - Tamanho 200000

Ordem	Média ComparaçõesPP	Média TransferênciasPP	Média TempoPP	Média Comparações	Média Transferências	Média Tempo
1	33334.0	33334.0	189.0	16681.0	5.0	17.33
2	33334.0	33334.0	186.0	16679.0	4.0	16.67

Tabela 18: Método 2 - Tamanho 200000

Ordem	Média ComparaçõesPP	Média TransferênciasPP	Média TempoPP	Média Comparações	Média Transferências	Média Tempo
3	4639190.0	4439192.0	35556.67	39.0	20.0	141.0

Tabela 19: Método 3 - Tamanho 200000

Ordem	Média ComparaçõesPP	Média TransferênciasPP	Média TempoPP	Média Comparações	Média Transferências	Média Tempo
1	6042546.0	1.0	1841.0	41.0	0.0	0.0
2	200000.0	1.0	1923.0	40.33	0.0	0.0
3	2638937.0	1.0	2095.67	36.33	0.0	0.0

Tabela 20: Método 4 - Tamanho 200000

Ordem	Média ComparaçõesPP	Média TransferênciasPP	Média TempoPP	Média Comparações	Média Transferências	Média Tempo
1	11138838.0	1.0	1704.67	30.0	0.0	0.0
2	6308905.0	1.0	1422.33	27.67	0.0	0.0
3	8031854.0	1.0	1670.67	29.33	0.0	0.0

5.6 Tamanho: 2000000

Tabela 21: Método 1 - Tamanho 2000000

Ordem	Média ComparaçõesPP	Média TransferênciasPP	Média TempoPP	Média Comparações	Média Transferências	Média Tempo
1	333334.0	333334.0	50646.0	166681.0	5.0	194.0
2	333334.0	333334.0	58136.0	166679.0	4.0	259.67

Tabela 22: Método 3 - Tamanho 2000000

Ordem	Média ComparaçõesPP	Média TransferênciasPP	Média TempoPP	Média Comparações	Média Transferências	Média Tempo
1	73023510.0	1.0	263823.0	50.33	0.0	2.0
2	2000000.0	1.0	104455.67	47.0	0.0	3.0
3	30831131.0	1.0	208104.33	43.0	0.0	0.0

Tabela 23: Método 4 - Tamanho 2000000

Ordem	Média ComparaçõesPP	Média TransferênciasPP	Média TempoPP	Média Comparações	Média Transferências	Média Tempo
1	132848606.0	1.0	139991.67	36.33	0.0	3.67
2	72236007.0	1.0	97623.0	32.67	0.0	1.67
3	93878988.0	1.0	237480.33	31.0	0.0	0.33

6 Análise das Métricas

6.1 Acesso Sequencial Indexado

O método de *acesso sequencial indexado* combina as vantagens do acesso sequencial com o conceito de paginação, permitindo otimizar a recuperação de dados em arquivos ordenados. Contudo, é importante destacar que esse método não é adequado para acessos de ordem aleatória, uma vez que a tabela de índices depende da ordenação prévia do arquivo.

- **Desempenho e Complexidade**:

  O método demonstrou um comportamento consistente em termos de número de comparações, transferências e tempo de execução, apresentando uma relação linear entre o número de registros e o tempo necessário para realizar tanto o pré-processamento quanto a busca de dados. Esse desempenho linear é um dos principais atributos do método, conferindo-lhe eficiência em cenários de baixo volume de dados.

- **Pré-processamento**:

  O pré-processamento no acesso sequencial indexado se mostrou significativamente mais rápido em comparação com outros métodos, independentemente do número de registros. Essa característica torna o método vantajoso em situações em que o pré-processamento precisa ser realizado com frequência, uma vez que ele minimiza o tempo necessário para organizar os dados antes de sua utilização.

- **Desempenho em Busca**:

  Por outro lado, o desempenho do método em operações de busca é consideravelmente inferior ao de técnicas como Árvore B e Árvore B*, que apresentam algoritmos mais eficientes para busca em grandes volumes de dados. Embora o acesso sequencial indexado seja eficiente em arquivos pequenos e ordenados, ele não é a melhor escolha quando se busca otimização no tempo de resposta para grandes conjuntos de dados.

- **Aplicabilidade**:

  Em termos de aplicabilidade, o acesso sequencial indexado é particularmente adequado para arquivos de tamanho pequeno a médio, onde os dados são previamente ordenados e a simplicidade da implementação e tempo de pré-processamento são fatores críticos.

6.2 Árvore Binária de Pesquisa

A Árvore Binária de Pesquisa é uma estrutura de dados tradicional, que organiza os registros em uma hierarquia, onde os elementos à esquerda de um nó possuem chaves menores e os elementos à direita possuem chaves maiores. Contudo, seu desempenho demonstrou ser inferior ao de outros métodos analisados, devido aos custos associados à atualização dos ponteiros durante a construção da árvore uma vez que é simulada memória secundária. Esse processo de atualização se mostra relativamente custoso, comprometendo a eficiência da método.

- **Desempenho e Complexidade**:

  O desempenho da Árvore Binária de Pesquisa foi consistentemente inferior ao de outras abordagens, uma vez que é construída e mantida em memória secundária. A operação de atualização aumenta a complexidade de construção e manutenção da estrutura. Porém sua vantagem aos demais métodos é que funciona para arquivos com registros desordenados, diferente da Acesso Indexado, e é consideravelmente mais simples de implementar que a Árvore B e B*.

- **Pré-processamento**:

  O pré-processamento da Árvore Binária de Pesquisa é consideravelmente mais lento em comparação com outras estruturas de dados, especialmente quando o conjunto de dados grandes e ordenados uma vez que a árvore não se balanceia sozinha, tendo para um lado (direito ou esquerdo), aumentando assim o numero de comparações e transferências ao atualizar os ponteiros.

- **Desempenho em Busca**:

  O desempenho de busca na Árvore Binária de Pesquisa também é afetado pelo seu desbalanceamento. Em tais situações, a árvore tende a degenerar para uma estrutura linear, resultando em um tempo de busca pior do que o esperado, aproximando-se da complexidade de uma lista encadeada. Isso torna a Árvore Binária de Pesquisa uma escolha quase que restrita a cenários onde os dados estão desordenados, porém neste cenário também é consideravelmente menor que métodos como Árvore B e Árvore B*, ganhando somente em simplicidade de implantação.

- **Aplicabilidade**:

  A Árvore Binária de Pesquisa é adequada para cenários onde a os dados se encontram desordenados, e onde a simplicidade da implementação é um fator desejado.

6.3 Árvore B

A Árvore B é uma estrutura de dados balanceada amplamente utilizada para gerenciar registros de forma eficiente. Embora o método tenha mostrado grande eficiência nas operações de busca, seu pré-processamento revelou-se consideravelmente mais lento quando comparado a abordagens como o Acesso Indexado.

- **Desempenho e Complexidade**:

  O método demonstrou um desempenho eficiente, com um número constante de transferências tanto no pré-processamento (1 transferência) quanto na busca (0 transferências). Isso ocorre porque os registros são lidos uma única vez a partir de um grande array denominado "registros", e a árvore é gerada diretamente a partir desse array. Como resultado, não há necessidade de acessar a memória secundária novamente durante a construção ou busca, o que contribui para sua alta eficiência.

- **Pré-processamento**:

  Embora o desempenho de busca tenha sido notável, o pré-processamento da Árvore B é mais demorado quando comparado ao de métodos como o Acesso Indexado. Esse tempo de pré-processamento é em grande parte influenciado pela necessidade de organizar os dados, o que pode ser mais custoso em termos de tempo dependendo da quantidade de dados.

- **Desempenho em Busca**:

  A Árvore B se destacou como o melhor método para pesquisa entre os avaliados, superando até mesmo a Árvore B* em alguns testes. O processo de busca se beneficia significativamente da estrutura balanceada da árvore, permitindo localizações rápidas de registros, com uma complexidade de busca logarítmica em relação ao número de registros.

- **Aplicabilidade**:

  A Árvore B é particularmente eficiente em cenários em que a busca é uma operação crítica e os dados não são pré-processados recorrentemente.

6.4 Árvore B*

A Árvore B* é uma variação da Árvore B, projetada para melhorar a eficiência da mesma, mantendo a estrutura balanceada. Em termos de desempenho, a Árvore B* demonstrou resultados bastante semelhantes aos da Árvore B, com uma performance constante nas transferências,

devido ao fato de que, assim como na Árvore B, os registros são lidos uma única vez a partir de um grande array "registros", e a árvore é gerada a partir desse array, sem necessidade de acessos repetidos à memória secundária.

- **Desempenho e Complexidade**:

  O desempenho da Árvore B* mostrou-se comparável ao da Árvore B, apresentando um número constante de transferências tanto no pré-processamento quanto na busca, pelo mesmo motivo: a leitura única dos registros e a construção da árvore diretamente a partir do array. No entanto, em termos de busca, a Árvore B* apresentou um desempenho ligeiramente inferior ao da Árvore B em alguns testes, embora as diferenças tenham sido mínimas.

- **Pré-processamento**:

  O pré-processamento na Árvore B* segue o mesmo princípio da Árvore B, com um número constante de transferências durante a organização dos dados. Porém tendo o pré-processamento mais demorado entre todos os métodos. A principal diferença entre os dois métodos está na estrutura interna da árvore, que, embora balanceada, pode introduzir ligeiras variações no tempo de construção da árvore em comparação à Árvore B por ser mais complexo.

- **Desempenho em Busca**:

  Em termos de desempenho de busca, a Árvore B* se mostrou ligeiramente menos eficiente que a Árvore B em alguns testes, embora as diferenças entre os dois métodos sejam pequenas. A principal vantagem da Árvore B* sobre a Árvore B reside em sua capacidade de manter um maior nível de balanceamento durante as operações de inserção.

- **Aplicabilidade**:

  A Árvore B* é indicada para cenários onde a alta taxa de inserção é um fator importante, pois ela melhora o balanceamento da árvore durante essas operações. No entanto, para cenários em que a busca é tem sua eficiência próxima a da Árvore B.

7 Conclusão

Os métodos analisados demonstraram diferentes desempenhos e complexidades, com cada abordagem oferecendo vantagens em cenários específicos. O *Acesso Sequencial Indexado* se destacou pela simplicidade e eficiência no pré-processamento, mas apresentou limitações em operações de busca, especialmente em grandes volumes de dados. A *Árvore Binária de Pesquisa* revelou-se adequada para cenários com dados desordenados, oferecendo facilidade de implementação, mas seu desempenho em busca e construção da árvore não se mostrou competitivo em comparação com as outras abordagens. A *Árvore B* e a *Árvore B**, por sua vez, se destacaram em operações de busca, com a *Árvore B* liderando nesse quesito devido à sua eficiência, enquanto a *Árvore B** ofereceu um desempenho ligeiramente inferior, mas com vantagens em termos de balanceamento durante inserções.

8 Reflexões Finais

O desenvolvimento deste projeto, apesar dos desafios encontrados, como o manejo de ponteiros e a necessidade de lidar com arquivos de grande porte, gerando varios *segmentation fault*, foi uma experiência desafiadora e produtiva. A manipulação de estruturas de dados como árvores

balanceadas e o acesso sequencial indexado proporcionaram uma compreensão mais profunda sobre a eficiência e a complexidade algorítmica em cenários de grande volume de dados. A gestão de memória, especialmente no que diz respeito ao uso de ponteiros em operações de inserção e deleção, exigiu uma atenção extra, mas também permitiu aprender sobre os detalhes técnicos que impactam diretamente o desempenho dos algoritmos. Apesar das dificuldades, a implementação e o teste dos diferentes métodos mostraram-se altamente enriquecedores.