UNIVERSIDADE FEDERAL DE OURO PRETO
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO - DECOM
Professor: GUILHERME TAVARES DE ASSIS

RELATÓRIO PRÁTICO

Disciplina: BCC 203 Estrutura de Dados II Turma: 11

Crescêncio Gusmão Castro 22.2.4094
João Pedro dos Santos Ferraz 23.1.4030
João Vitor Coelho Oliveira 23.1.4133
Kanan Hollerbach Lopes 23.1.4017
Luiza Oliveira Magalhães de Souza 23.1.4009

OURO PRETO
2025

Sumário

Introdução

No conteúdo abordado na disciplina de Estrutura de Dados I I , foram introduzidas a relevância e a efetividade de técnicas de pesquisa e ordenação de dados em memória secundária, as quais se fazem necessárias quando não é possível armazenar um grande volume de dados na memória interna.

Realizar transferências de dados armazenados em disco para memória principal tem um custo significativo, em números de comparações e leituras. Dessa forma, a fim de otimizar o desempenho dos algoritmos de pesquisa externa, associam-se a eles sistemas eficientes de transferência de dados, como o sistema de paginação.

Para avaliar a performance das técnicas utilizadas, consideramos como principais métricas: o número de transferências (leitura) entre as memórias interna e externa - especialmente importante em algoritmos de pesquisa externa; o número de comparações entre as chaves de pesquisa - especialmente importante em algoritmos de pesquisa interna; e o tempo de execução.

Os custos são refletidos pela forma em que acessamos os dados: enquanto a memória secundária pode ser acessada apenas por um dado momento, a memória principal pode acessar qualquer dado a um custo único, de maneira aleatória. A fim de encontrar eficientemente um registro buscado pelo usuário em um arquivo, nosso objetivo foi implementar e avaliar os resultados da análise experimental da complexidade de desempenho dos seguintes métodos de pesquisa externa:

1. Acesso Sequencial Indexado;
2. Árvore binária de pesquisa adequada à memória externa;
3. Árvore B;
4. Árvore B*;

Diretivas de compilação

Para compilar o programa são necessárias duas etapas:

A primeira consiste em compilar o código da geração do arquivo escolhido para realizar os testes de pesquisa. Nesse algoritmo, há uma interação com o usuário para atender às demandas das características do arquivo, determinando seu tamanho. No caso, a formulação do arquivo .txt é uma prática para validação dos dados inseridos, e ele é criado junto com o arquivo out.bin, que é o arquivo binário lido durante a execução e contém os dados.

Além disso, o usuário também é questionado em relação ao formato da apresentação dos dados, isto é, se os registros vão ser ordenados em ordem crescente, decrescente ou aleatória. Após essa fase, já é possível compilar o programa principal para os métodos de pesquisa.

O comando para a execução do programa é da forma:

```
./pesquisa <método> <quantidade> <situação> <chave> [-P]
```

Sendo:

- **pesquisa**: o nome do arquivo executável do programa (padronizado por "pesquisa", conforme o enunciado);
- **método**: o método de pesquisa externa a ser executado, classificado de 1 a 4, tal que 1 é o acesso sequencial indexado, 2 é a árvore binária de pesquisa adequada à memória externa, 3 é a árvore B e 4 é a árvore B*;
- **quantidade**: a quantidade de registros do arquivo considerado, podendo ser 100, 1.000, 10.000, 100.000 ou 1.000.000;
- **situação**: situação de ordem do arquivo, classificada de 1 a 3, tal que 1 é o arquivo ordenado ascendentemente, 2 é o arquivo ordenado descendentemente e 3 é o arquivo ordenado aleatoriamente;
- **chave**: a chave a ser pesquisada dentro do arquivo;
- **[-P]** : é um argumento opcional, que deve ser colocado quando se deseja que as chaves de pesquisa dos registros do arquivo considerado sejam apresentadas na tela (deve ser escrito como: -p).

Dessa maneira, executa-se o método de pesquisa desejado com os parâmetros escolhidos. Após essas duas etapas, o programa compila, constrói a estrutura de dados escolhida e faz a pesquisa da chave pretendida.

Descrição dos testes

Apresentamos o experimento e os resultados dos testes dos quatro métodos de pesquisa, implementados na linguagem C.

Temos um arquivo gerador (generator.c), que gera chaves em ordem crescente, decrescente e aleatória. Os arquivos podem ter 100, 1.000, 10.000, 100.000 e 1.000.000 de registros quaisquer.

Em cada um dos testes, foi realizada a medida de:

- Número de transferências (leitura) de registros da memória externa para a memória interna;
- Número de transferências (escrita) de registros da memória interna para a memória externa;
- Número de comparações entre valores do campo de ordenação dos registros;
- Tempo médio de execução do algoritmo.

Para cada teste, era criado um arquivo de cada tamanho e localizado as chaves escolhidas pelos integrantes do grupo. Dessa forma, cada um via como funcionava o algoritmo e testava em sua própria máquina, percebendo também como o processador de cada computador pode apresentar um diferencial na hora de executar os métodos.

Alguns testes foram bem sucedidos, tiveram tempos de execução e comparações entre chaves excelentes, porém outros testes não foram tão bons e alguns apresentaram erros, que serão descritos no tópico Dificuldades Encontradas.

Desenvolvimento

Acesso Sequencial Indexado

Associado ao sistema de paginação, esse método usa um arquivo de índice de páginas para localizar rapidamente registros e permitir a leitura sequencial dentro delas. Os índices se dão em pares de valores, que armazenam, respectivamente, a chave do primeiro item e o endereço de uma determinada página.

No código foi criado a função Sequencial, que inicia-se na abertura do arquivo gerado (caso não foi possível criar, aparecerá uma mensagem), e calcula a quantidade de páginas que serão geradas dividindo - se o número total de itens pela quantidade de itens por página, que nesse caso será 100 itens por página. Dessa forma, ao ler o arquivo, o código faz essas transferências dos dados para formular o sistema de paginação.

Então, é gerado um switch para saber se o arquivo está ordenado crescente ou decrescentemente. Dessa forma, o arquivo é lido e começa a comparação das chaves para procurar a que deseja encontrar. Essa comparação é feita em duas etapas: primeiro é comparado a chave que se procura pelo primeiro item da página. Caso esteja na primeira página, o algoritmo acessa ela para fazer a busca, caso contrário, lê - se o primeiro item da próxima página até encontrar uma página em que esse item esteja localizado.

Caso encontre, o Item é retornado e printado na tela, caso contrário, aparecerá uma mensagem de erro alertando que não existe a chave. No final do processo, o arquivo é fechado, as alocações de memória que foram feitas para criar a paginação são liberadas e o programa se encerra.

```
void sequencial(int qntdItens, int situacao, long int chave, long *nTransferenciaPre, long
*nTransferencia, long *nComparacoes);
```

```
//PROCURA DO ITEM
for (unsigned int k = 0; k < count; k++) {
    (*nComparacoes)++;
    if(l.paginas[i].itens[k].Chave == chave){
        printf("-----------------------------------------------\nItem
        encontrado\n\nDetalhes ['%d', '%ld', '%s', '%s']
        \n-----------------------------------------------", l.paginas[i].itens[k].
        Chave, l.paginas[i].itens[k].dado1, l.paginas[i].itens[k].dado2, l.paginas[i].
        itens[k].dado3);

        //LIBERAÇÃO DE MEMÓRIA ALOCADA
        free(l.paginas);

        //FECHAMENTO DOS ARQUIVOS
        fclose(arq);
        return;
    }
}
```

Árvore binária de Pesquisa adequada à memória externa

Na implementação desse método, a organização dos registros não acontece em memória principal. Em vez disso, usamos uma estrutura que os nós são armazenados no disco, simulando ponteiros através das posições dos nós no arquivo.

Cada nó possui uma chave e ponteiros simulados para os filhos esquerdo e direito, representados por inteiros que indicam a posição no arquivo onde esses filhos estão gravados. A construção da árvore é feita sequencialmente a partir das informações do arquivo `out.bin`, gerando o arquivo `arvore_binaria.bin`, que armazena os nós da árvore. Em relação a inserção, cada novo nó é comparado com o nó atual e direcionado para a esquerda ou direita de acordo com a chave, até encontrar uma posição vazia, que é então atualizada no disco.

```c
FILE *arv = fopen("arvore_binaria.bin", "wb+");
if (!arv) {
    printf("Erro ao criar arvore_binaria.bin\n");
    fclose(arquivo);
    return 1;
}

TipoRegistro temp;
int pos = 0;
while (fread(&temp, sizeof(TipoRegistro), 1, arquivo) == 1 && pos < quantidade) {
    NoArvore no;
    no.chave = temp.Chave;
    no.dado1 = temp.dado1;
    strcpy(no.dado2, temp.dado2);
    no.esquerda = no.direita = -1;
    no.indice = pos;
    fseek(arv, pos * sizeof(NoArvore), SEEK_SET);
    fwrite(&no, sizeof(NoArvore), 1, arv);
    pos++;
    nTransferenciaPre++;
}

rewind(arv);
for (int i = 1; i < pos; i++) {
    inserirNaArvore(arv, 0, i, &nTransferencia, &nComparacoes);
}
```

```c
void inserirNaArvore(FILE *arquivo, int posRaiz, int posNovo, int *acessos, int *comparacoes) {
    NoArvore novo = lerNo(arquivo, posNovo, acessos);
    int posAtual = posRaiz;

    while (1) {
        NoArvore atual = lerNo(arquivo, posAtual, acessos);
        (*comparacoes)++;

        if (novo.chave < atual.chave) {
            if (atual.esquerda == -1) {
                atual.esquerda = posNovo;
                escreverNo(arquivo, posAtual, &atual, acessos);
                break;
            } else {
                posAtual = atual.esquerda;
            }
        } else {
            if (atual.direita == -1) {
                atual.direita = posNovo;
                escreverNo(arquivo, posAtual, &atual, acessos);
                break;
            } else {
                posAtual = atual.direita;
            }
        }
    }
}
```

Depois dessa etapa, chamamos o algoritmo de busca para percorrer a árvore e ler os nós direto do arquivo. A cada passo, comparamos a chave que está sendo procurada e o caminho segue para o filho à esquerda ou direita conforme necessário, até que item desejado seja encontrado.

```c
int buscarNaArvore(int chave, FILE *arquivo, int *acessos, int *comparacoes) {
    int pos = 0;
    NoArvore atual;

    while (1) {
        fseek(arquivo, pos * sizeof(NoArvore), SEEK_SET);
        fread(&atual, sizeof(NoArvore), 1, arquivo);
        (*acessos)++;
        (*comparacoes)++;

        if (chave == atual.chave)
            return chave;

        if (chave < atual.chave) {
            if (atual.esquerda == -1) return -1;
            pos = atual.esquerda;
        } else {
            if (atual.direita == -1) return -1;
            pos = atual.direita;
        }
    }
}
```

Árvore B

O código implementa uma **Árvore B em memória principal**, com o propósito de armazenar e pesquisar registros com chaves do tipo `long`. A árvore é construída dinamicamente a partir da leitura de um arquivo binário (`out.bin`) contendo registros.A Árvore B é uma estrutura de dados balanceada, ideal para sistemas que lidam com **grandes volumes de dados**, pois **minimiza o número de acessos a disco**. Ela organiza os dados em páginas (nós) que podem conter múltiplos registros e ponteiros, mantendo a árvore sempre balanceada após inserções e buscas. **Importante:** nesta implementação, a Árvore B opera apenas na memória principal. Ou seja, não há uso de memória secundária (disco**)** para manter a estrutura da árvore. Temos como base:

1) A estrutura:

```
typedef struct TipoPagina {
    short n;
    TipoRegistro r[MM];
    TipoApontador p[MM + 1];
} TipoPagina;
```

Representa um nó da Árvore B, armazenando até `MM` registros e `MM+1` ponteiros para filhos.

2) Função responsável por inserir um novo registro na posição correta dentro de uma página, mantendo a ordenação.

```
void InsereNaPagina(TipoApontador Ap, TipoRegistro Reg, TipoApontador ApDir, int *comparacoes) {
    short NaoAchouPosicao = (Ap->n > 0);
    int k = Ap->n;

    while (NaoAchouPosicao) {
        if (comparacoes) (*comparacoes)++;
        if (Reg.Chave >= Ap->r[k - 1].Chave) {
            NaoAchouPosicao = 0;
        } else {
            Ap->r[k] = Ap->r[k - 1];
            Ap->p[k + 1] = Ap->p[k];
            k--;
            if (k < 1) NaoAchouPosicao = 0;
        }
    }

    Ap->r[k] = Reg;
    Ap->p[k + 1] = ApDir;
    Ap->n++;
}
```

3) A lógica recursiva que realiza inserções, detecta excesso de registros e divide páginas quando necessário.

```c
void Ins(TipoRegistro Reg, TipoApontador Ap, short *Cresceu,
         TipoRegistro *RegRetorno, TipoApontador *ApRetorno, int *comparacoes) {

    long i = 1, j;
    TipoApontador ApTemp;

    if (Ap == NULL) {
        *Cresceu = 1;
        *RegRetorno = Reg;
        *ApRetorno = NULL;
        return;
    }

    while (i < Ap->n && Reg.Chave > Ap->r[i - 1].Chave) {
        if (comparacoes) (*comparacoes)++;
        i++;
    }
    if (comparacoes) (*comparacoes)++;

    if (Reg.Chave == Ap->r[i - 1].Chave) {
        printf("Erro: registro %ld já está presente\n", Reg.Chave);
        *Cresceu = 0;
        return;
    }

    if (Reg.Chave < Ap->r[i - 1].Chave) i--;
```

4) Busca Recursiva com Contagem de Comparações, executa a busca de uma chave na árvore, percorrendo as páginas com comparação ordenada.

```c
int Pesquisa(TipoRegistro *x, TipoApontador Ap, int *comparacoes) {
    long i = 1;

    if (Ap == NULL) {
        printf("TipoRegistro não está presente\n");
        return 0;
    }

    while (i < Ap->n && x->Chave > Ap->r[i - 1].Chave) {
        i++;
        if (comparacoes) (*comparacoes)++;
    }
    if (comparacoes) (*comparacoes)++;

    if (x->Chave == Ap->r[i - 1].Chave) {
        *x = Ap->r[i - 1];
        return 1;
    }

    if (x->Chave < Ap->r[i - 1].Chave) {
        return Pesquisa(x, Ap->p[i - 1], comparacoes);
    } else {
        return Pesquisa(x, Ap->p[i], comparacoes);
    }
}
```

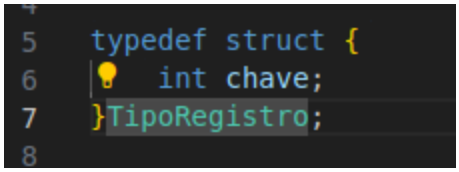
Árvore B*

A **Árvore B*** é uma estrutura de dados otimizada para armazenamento em memória externa, como discos, funcionando como uma variação mais eficiente da Árvore B. Cada nó da Árvore B* armazena múltiplas chaves e ponteiros, permitindo que cada leitura de disco traga um grande volume de dados. Seu principal diferencial é a exigência de que os nós internos estejam preenchidos em pelo menos 66% de sua capacidade, resultando em uma árvore mais "rasa" e menos acessos ao disco, crucial para o desempenho em sistemas de gerenciamento de banco de dados e arquivos que lidam com grandes volumes de dados.

A implementação da Árvore B* neste trabalho é uma estrutura de dados de busca auto-balanceada otimizada para armazenamento em disco, caracterizada por manter todas as chaves em ordem em nós folha (páginas externas) e utilizar nós internos para indexar essas folhas, minimizando acessos a disco ao agrupar múltiplos registros ou ponteiros em cada página. As inserções são realizadas de forma recursiva, propagando divisões de páginas (internas ou externas) para os níveis superiores quando uma página excede sua capacidade, garantindo que a árvore permaneça balanceada e que as chaves sejam promovidas adequadamente para os nós internos. A pesquisa percorre a árvore, comparando chaves nos nós internos até atingir a página externa correta, onde então busca linearmente pelo registro desejado. O código também inclui um sistema de paginação para simular o carregamento de dados de um arquivo binário em blocos, e um módulo de estatísticas para medir o tempo de execução, o número de comparações e os acessos a disco durante as operações. **Importante:** o tamanho das páginas foram 100 fixo para todos os teste, assim como M = 10 para todos os testes.

Estruturas:

1) TipoRegistro: Define a estrutura básica de um registro armazenado na árvore, contendo um campo `chave` inteiro para identificação e ordenação.



2) TipoPagina: Representa um nó da Árvore B*, que pode ser de dois tipos:
   - **Página Interna (**U0**):** Contém um array de chaves (`chave`) e um array de ponteiros para páginas filhas (`pi`), além de um contador (`ni`) para o número de chaves.

- **Página Externa (**U1**):** Contém um array de registros (`re`) e um contador (`ne`) para o número de registros.

```
16  typedef struct TipoPagina {
17      TipoIntExt pt;
18      union {
19          struct { // Pagina interna
20              int ni;
21              int chave[2 * M];
22              TipoApontador pi[2 * M + 1];
23          }U0;
24          struct { // Pagina externa
25              int ne;
26              TipoRegistro re[2 * MM];
27          }U1;
28      }UU;
29  } TipoPagina;
```

3) TipoApontador: É um ponteiro para uma TipoPagina, utilizado para referenciar os nós da árvore.

```
13  typedef TipoPagina* TipoApontador;
```

4) Estatísticas: Usada para coletar métricas de desempenho, incluindo o tempo de execução (tempo), o número de comparações de chaves (comparações) e o número de acessos a páginas (acessos).

```
38  typedef struct {
39      unsigned long long tempo;
40      int comparacoes;
41      int acessos;
42      Time time;
43  } Estatisticas;
```

5) PagReg: Estrutura auxiliar para o processo de paginação, representando um bloco de registros lidos do arquivo.

```c
typedef struct {
    TipoRegistro registros[TAM_PAG];
}PagReg;
```

Funções:

1) insere(TipoRegistro reg, TipoApontador ap, Estatisticas *estatisticas): Função principal de inserção, responsável por adicionar um TipoRegistro na árvore, gerenciando a criação da raiz e as divisões que podem ocorrer.

```c
TipoApontador insere(TipoRegistro reg, TipoApontador ap, Estatisticas *estatisticas) {
    //Startando temporizador
    struct timespec t_inicio, t_fim;
    clock_gettime(CLOCK_MONOTONIC, &t_inicio);

    //Primeira chamada, cria raiz
    if (ap == NULL) {
    TipoApontador novaFolha = malloc(sizeof(TipoPagina));
    if (!novaFolha) exit(EXIT_FAILURE);
        novaFolha->pt        = EXTERNO;
        novaFolha->UU.U1.ne  = 1;
        novaFolha->UU.U1.re[0] = reg;
        return novaFolha;
    }

    bool cresceu;
    TipoRegistro regRetorno;
    TipoPagina *apRetorno, *apTemp;

    ins(reg, ap, &cresceu, &regRetorno, &apRetorno, estatisticas);
    //crescimento na raiz
    if(cresceu) {
        apTemp = malloc(sizeof(TipoPagina));
        if(apTemp == NULL) {
            printf("Erro ao alocar Raiz");
            exit(EXIT_FAILURE);
        }
        apTemp->pt = INTERNO;
        apTemp->UU.U0.ni = 1;
        apTemp->UU.U0.chave[0] = regRetorno.chave; //Nova chave (Raiz)
        apTemp->UU.U0.pi[0] = ap;       // filho esquerdo = raiz antiga
        apTemp->UU.U0.pi[1] = apRetorno; // filho direito = nova página
        return apTemp; // RETORNA NOVA RAIZ
    }
    clock_gettime(CLOCK_MONOTONIC, &t_fim);
    estatisticas->time.inicio = t_inicio.tv_sec * 1000000000ULL + t_inicio.tv_nsec;
    estatisticas->time.fim    = t_fim.tv_sec    * 1000000000ULL + t_fim.tv_nsec;
    estatisticas->tempo += estatisticas->time.fim - estatisticas->time.inicio;
    return ap; // retorna raiz original
}
```

2) ins(TipoRegistro registro, TipoApontador paginaAtual, bool *cresceu, TipoRegistro *registroRetorno, TipoApontador *paginaRetorno, Estatisticas *estatisticas): Função recursiva auxiliar de inserção que percorre a árvore, localiza a página correta para inserção e lida com a lógica de divisão e promoção de chaves.

```c
// Insere um registro recursivamente em uma árvore B*
void ins(TipoRegistro registro, TipoApontador paginaAtual, bool *cresceu, TipoRegistro *registroRetorno, TipoApontador *paginaRetorno, Estatisticas *estatisticas) {
    unsigned int i;

    // Tratamento de pagina interna
    if (paginaAtual->pt == INTERNO) { i = 0;
        // Caminhamento pela arvore
        while (i < paginaAtual->UU.U0.ni && paginaAtual->UU.U0.chave[i] < registro.chave) i++;
        ins(registro, paginaAtual->UU.U0.pi[i], cresceu, registroRetorno, paginaRetorno, estatisticas);

        if (!*cresceu) {
            return;
        }

        // Pagina comporta novo registro
        if (paginaAtual->UU.U0.ni < 2 * M) {
            inserePaginaInterna(paginaAtual, registroRetorno->chave, *paginaRetorno, estatisticas);
            *cresceu = false;
            return;
        }

       // Criação da nova página
        TipoApontador apTemp = malloc(sizeof(TipoPagina));
        if (apTemp == NULL) {
            printf("Nao foi possivel alocar memoria (pagina Interna)\n");
            exit(EXIT_FAILURE);
        }
        apTemp->pt = INTERNO;
        apTemp->UU.U0.ni = 0;

        // Inserção do novo elemento na página correta
        if (i < M + 1) {
            // Inserir última chave antiga na nova página
            inserePaginaInterna(apTemp, paginaAtual->UU.U0.chave[2 * M - 1], paginaAtual->UU.U0.pi[2 * M], estatisticas);
            paginaAtual->UU.U0.ni--; // Decrementa a quantidade de chaves
            // Inserir nova chave na página original
            inserePaginaInterna(paginaAtual, registroRetorno->chave, *paginaRetorno, estatisticas);
        } else {
            inserePaginaInterna(apTemp, registroRetorno->chave, *paginaRetorno, estatisticas);
        }
```

```c
        // Transferência das chaves maiores (M+2 a 2M)
        for (int j = M + 2; j <= 2 * M; j++) {
            inserePaginaInterna(apTemp,  paginaAtual->UU.U0.chave[j - 1], paginaAtual->UU.U0.pi[j], estatisticas);
        }
        // Reorganiza ponteiros e promove chave
        paginaAtual->UU.U0.ni = M; //Tamnho pagina atual = M
        apTemp->UU.U0.pi[0] = paginaAtual->UU.U0.pi[M + 1]; //Pagina temp recebe ponteiro
        TipoRegistro regPromovido; //Registro temporario para subir chave
        regPromovido.chave = paginaAtual->UU.U0.chave[M]; //Item do meio promovido
        *registroRetorno = regPromovido; //Promove para proxima chamada da funcao
        *paginaRetorno = apTemp; //Retorna nova pagina
        *cresceu = true; //Arvore cresceu
    }
    // Tratamento da pagina externa (folha)
    else { i = 0;
        // Posicao correta para o item ficar
        while (i < paginaAtual->UU.U1.ne && paginaAtual->UU.U1.re[i].chave < registro.chave) i++;

        // Itens repetidos
        if (i < paginaAtual->UU.U1.ne && registro.chave == paginaAtual->UU.U1.re[i].chave) {
            *cresceu = false;
            return;
        }

        // Pagina atual "Comporta" novo item
        if (paginaAtual->UU.U1.ne < 2 * M) {
            inserePaginaExterna(paginaAtual, registro, estatisticas);
            *cresceu = false;
            return;
        }

        // insere e ordena
        inserePaginaExterna(paginaAtual, registro, estatisticas);
        // promove termo do meio
        *registroRetorno = paginaAtual->UU.U1.re[M];
        // cria nova folha e copia
        TipoApontador apTemp = malloc(sizeof(TipoPagina));
        if (!apTemp) exit(EXIT_FAILURE);
        apTemp->pt = EXTERNO;
        apTemp->UU.U1.ne = 0;
        for (int j = M+1; j <= 2 * M; j++) {
            inserePaginaExterna(apTemp, paginaAtual->UU.U1.re[j], estatisticas);
        }
```

```
        // ajusta ne da folha original para 0..M
        paginaAtual->UU.U1.ne = M + 1;
        *cresceu = true;
        *paginaRetorno = apTemp;
    }
}
```

3) inserePaginaExterna(TipoApontador ap, TipoRegistro reg, Estatisticas *estatisticas): Insere um registro em uma página externa (folha), mantendo a ordem das chaves.

```
void inserePaginaExterna(TipoApontador ap, TipoRegistro reg, Estatisticas *estatisticas) {
    int k = ap->UU.U1.ne;

    while(k > 0) {
        if(reg.chave >= ap->UU.U1.re[k - 1].chave)
            break;
        ap->UU.U1.re[k] = ap->UU.U1.re[k - 1];
        k--;
    }

    ap->UU.U1.re[k] = reg;
    ap->UU.U1.ne++;
}
```

4) inserePaginaInterna(TipoApontador ap, int chavePromovida, TipoApontador apDir, Estatisticas *estatisticas): Insere uma chave e um ponteiro em uma página interna, mantendo a ordem.

```
void inserePaginaInterna(TipoApontador ap, int chavePromovida, TipoApontador apDir, Estatisticas *estatisticas) {
    int k = ap->UU.U0.ni;

    // Loop para encontrar a posição correta
    while (k > 0 && chavePromovida < ap->UU.U0.chave[k - 1]) {
        ap->UU.U0.chave[k] = ap->UU.U0.chave[k - 1];
        ap->UU.U0.pi[k + 1] = ap->UU.U0.pi[k];
        k--;
    }

    ap->UU.U0.chave[k] = chavePromovida;
    ap->UU.U0.pi[k + 1] = apDir;
    ap->UU.U0.ni++;
}
```

5) pesquisaArvoreBE(TipoRegistro reg, TipoApontador ap, Estatisticas *estatisticas, TipoApontador *paginaAtual): Realiza a busca por um TipoRegistro na árvore com base em sua chave, retornando true se encontrado e false caso contrário, além de fornecer a

página onde a chave foi encontrada.

```
196  //---------- PESQUISA ----------//
197  bool pesquisaArvoreBE(TipoRegistro reg, TipoApontador ap, Estatisticas *estatisticas, TipoApontador *paginaAtual) {
198      if(ap == NULL) return false; //Se raiz nula, retorna
199
200      unsigned int i = 0;
201      //Caminhamento paginas internas
202      if(ap->pt == INTERNO) {
203          while(i < ap->UU.U0.ni && reg.chave >= ap->UU.U0.chave[i]){
204              i++;
205              estatisticas->comparacoes ++;
206          }
207          return pesquisaArvoreBE(reg, ap->UU.U0.pi[i], estatisticas, paginaAtual);
208      }
209      //Verificacao chave
210      else while(i < ap->UU.U1.ne && ap->UU.U1.re[i].chave < reg.chave){
211          i++;
212          estatisticas->comparacoes ++;
213      }
214      if(i < ap->UU.U1.ne && reg.chave == ap->UU.U1.re[i].chave) {
215          (*paginaAtual) = ap; //folha no qual a chave esta presente
216          return true;
217      }
218      else return false;
219  }
```

6) abrirArquivo(int argv, const char *nomeDoArquivo): Abre o arquivo binário de onde os registros serão lidos.

```
7   FILE *abrirArquivo(int argv, const char *nomeDoArquivo) {
8       if(argv != 2) {
9           perror("Quantidade de argumentos invalida!\n");
10          exit(EXIT_FAILURE);
11      }
12
13      FILE *file = fopen(nomeDoArquivo, "rb");
14      if(file == NULL)
15          exit(EXIT_FAILURE);
16      return file;
17  }
18
19  bool fecharArquivo(FILE *arq) {
20      if(arq != NULL) {
21          fclose(arq);
22          return true;
23      }
24      return false;
25  }
```

7) paginacao(FILE *arq, PagReg *reg, Estatisticas *estatisticas): Lê um bloco de registros do arquivo, simulando um acesso a disco, e atualiza as estatísticas de acesso ao arquivo.

```c
bool paginacao(FILE *arq, PagReg *reg, Estatisticas *estatisticas) {
    struct timespec t_inicio, t_fim;
    clock_gettime(CLOCK_MONOTONIC, &t_inicio);

    estatisticas->acessos ++;

    size_t lidos = fread(reg, sizeof(PagReg), 1, arq);

    clock_gettime(CLOCK_MONOTONIC, &t_fim);
    estatisticas->time.inicio = t_inicio.tv_sec * 1000000000ULL + t_inicio.tv_nsec;
    estatisticas->time.fim    = t_fim.tv_sec    * 1000000000ULL + t_fim.tv_nsec;
    estatisticas->tempo += estatisticas->time.fim - estatisticas->time.inicio;

    return (lidos == 1);
}
```

8) iniciaEstatisticas(Estatisticas *estatisticas): Zera os contadores das estatísticas para uma nova medição de desempenho.

```c
void iniciaEstatisticas(Estatisticas *estatisticas) {
    estatisticas->acessos = 0;
    estatisticas->comparacoes = 0;
    estatisticas->tempo = 0;
    estatisticas->time.inicio = 0;
    estatisticas->time.fim = 0;
}
```

9) destruirArvoreBE(TipoApontador ap): Libera recursivamente toda a memória alocada para a árvore, prevenindo vazamentos.

```c
void destruirArvoreBE(TipoApontador ap) {
    if (ap == NULL)
        return;

    if (ap->pt == INTERNO) {
        for (int i = 0; i <= ap->UU.U0.ni; i++) {
            destruirArvoreBE(ap->UU.U0.pi[i]);
        }
    }

    free(ap);
}
```

10) gerarAleatorios(int *vetor): Gera um array de números inteiros aleatórios, utilizado para testar a função de pesquisa.

```
43  void gerarAleatorios(int *vetor) {
44      srand((unsigned)time(NULL));
45      for(int i = 0; i < 10; i++)
46          vetor[i] = rand() % 1000; // valores entre 0 e 999
47  }
```

Conclusão

Em suma, os quatros métodos de pesquisa externa explorados neste trabalho foram técnicas que tiveram um grande avanço na área de Estrutura de Dados, pois elas auxiliaram na busca de itens em arquivos muito grande que não conseguem ser transferidos totalmente para a memória principal.

A implementação dos métodos foi uma maneira de perceber e comparar como cada um funciona e o quanto de dados e tempo leva para cada um executar eficientemente. Nos tópicos abaixo são apresentados essas análises e as dificuldades que o grupo encontrou durante a construção do algoritmo.

Análise comparativa dos métodos

Para comparar a performance dos diferentes métodos aplicados a arquivos de diferentes tamanhos e modos de ordenação, construímos tabelas que avaliam as métricas principais.

Acesso Sequencial Indexado Arquivo em **ordem crescente**	100 Registros	1000 Registros	10.000 Registros	100.000 Registros	1.000.000 Registros
N° de transferências no pré-processamento	101	110	200	1100	
N° de transferências na pesquisa	1	1	1	1	
N° de comparações no pré-processamento	0	0	0	0	0
N° de comparações entre chaves	33	61	94	546	
Tempo de execução	0,006	0,008	0,120	0,020	

Acesso Sequencial Indexado Arquivo em **ordem decrescente**	100 Registros	1000 Registros	10.000 Registros	100.000 Registros	1.000.000 Registros
N° de transferências no pré-processamento	101	110	200	1100	
N° de transferências na pesquisa	1	1	1	1	
N° de comparações no pré-processamento	0	0	0	0	0
N° de comparações entre chaves	34	66	128	171	
Tempo de execução	0,005	0,006	0,008	0,026	

Árvore Binária Arquivo de **ordem crescente**	100 Registros	1000 Registros	10.000 Registros	100.000 Registros	1.000.000 Registros
N° de transferências no pré-processamento	100	1000	-	-	-
N° de transferências na pesquisa	5205	501555	-	-	-
N° de comparações no pré-processamento	4950	499500	-	-	-
N° de comparações entre chaves	57	57	-	-	-
Tempo de execução	0,053000	4,581000	-	-	-

Árvore Binária Arquivo de **ordem decrescente**	100 Registros	1000 Registros	10.000 Registros	100.000 Registros	1.000.000 Registros
N° de transferências no pré-processamento	100	1000	-	-	-
N° de transferências na pesquisa	5242	501932	-	-	-
N° de comparações no pré-processamento	4950	499500	-	-	-
N° de comparações entre chaves	94	434	-	-	-
Tempo de execução	0,061000	4,630000	-	-	-

Árvore Binária Arquivo de **ordem aleatória**	100 Registros	1000 Registros	10.000 Registros	100.000 Registros	1.000.000 Registros
N° de transferências no pré-processamento	100	1000	10000	100.000	-
N° de transferências na pesquisa	882	12147	13318	2224003	-
N° de comparações no pré-processamento	672	10142	11303	2023985	-
N° de comparações entre chaves	12	7	17	20	-
Tempo de execução	0,012000	0,171000	0,16300	31,72000	-

Árvore B Arquivo de **ordem crescente**	100 Registros	1000 Registros	10.000 Registros	100.000 Registros	1.000.000 Registros
N° de transferências no pré-processamento	100	1000	10000	100000	-
N° de transferências na pesquisa	0	0	0	0	0
N° de comparações no pré-processamento	1002	16288	225566	2891697	-
N° de comparações entre chaves	9	13	15	19	-
Tempo de execução	0.001	0.011	0.105	1.07	-

Árvore B Arquivo de **ordem decrescente**	100 Registros	1000 Registros	10.000 Registros	100.000 Registros	1.000.000 Registros
N° de transferências no pré-processamento	100	1000	10.000	100.000	-
N° de transferências na pesquisa	0	0	0	0	0
N° de comparações no pré-processamento	734	9721	118366	1390861	-
N° de comparações entre chaves	4	12	11	13	-
Tempo de execução	0,001	0,012	0,094	0,95	-

Árvore Binária Arquivo de **ordem aleatória**	100 Registros	1000 Registros	10.000 Registros	100.000 Registros	1.000.000 Registros
N° de transferências no pré-processamento	100	1000	10.000	100.000	-
N° de transferências na pesquisa	0	0	0	0	0
N° de comparações no pré-processamento	821	12575	161891	1982694	-
N° de comparações entre chaves	3	9	15	20	-
Tempo de execução	0,00000	0,01	0,108	2.76	-

Árvore B* Arquivo de **ordem crescente**	100 Registros	1000 Registros	10.000 Registros	100.000 Registros	1.000.000 Registros
N° de transferências	2	11	101	1001	-
N° de comparações	8	16	19	19	-
Tempo de execução(índice)	0.000012(segundos)	0.000179(segundos)	0.001918(segundos)	0.014872(segundos)	-
Tempo de execução(Pesquisa)	0.000058(segundos)	0.000045(segundos)	0.000067(segundos)	0.000032(segundos)	-

Árvore B* Arquivo de **ordem decrescente**	100 Registros	1000 Registros	10.000 Registros	100.000 Registros	1.000.000 Registros
N° de transferências	2	11	101	1001	-
N° de comparações	12	15	20	26	-
Tempo de execução(índice)	0.000014(segundos)	0.000150(segundos)	0.001717(segundos)	0.015911(segundos)	-
Tempo de execução(Pesquisa)	0.000057(segundos)	0.000041(segundos)	0.000045(segundos)	0.000046(segundos)	-

Árvore B* Arquivo de **ordem aleatória**	100 Registros	1000 Registros	10.000 Registros	100.000 Registros	1.000.000 Registros
N° de transferências	2	11	101	1001	-
N° de comparações	1	4	6	21	-
Tempo de execução(Índice)	0.000012(segundos)	0.000227(segundos)	0.002193(segundos)	0.017631(segundos)	-
Tempo de execução(Pesquisa)	0.000042(segundos)	0.000052(segundos)	0.000061(segundos)	0.000026(segundos)	-

Dificuldades de Implementação

Durante a construção do código e a realização dos testes, algumas dificuldades foram encontradas. As principais foram as seguintes:

1) **Falta de unidade durante a implementação das estruturas**: A fim de agilizar a construção do trabalho, o grupo se separou para fazer os diferentes métodos. Entretanto, não foi decidido um modelo padrão de registros, fazendo com que cada um elaborasse o próprio sistema de `structs`. Felizmente, devido ao funcionamento dos `arquivos.h`, foi possível realizar a linkagem na `main.c`, exceto do método de B*, que gerava muito conflito. Sendo assim, não ter um padrão de registros demonstra má prática de programação e uma postura principiante enquanto programador, e prejudicou o funcionamento 100% do trabalho.
2) **Tamanho do arquivo impede a execução dos testes**: Em arquivos com maior quantidade de itens, os testes não conseguiram ser concluídos devido a demora na execução do método, não sendo possível conseguir os resultados. Isso se deve não só ao tamanho do arquivo, como também a forma como os métodos foram implementados.