Universidade Federal de Ouro Preto - UFOP

Instituto de Ciências Exatas e Biológicas - ICEB
Departamento de Computação - DECOM
Curso de Ciência da Computação

Trabalho Prático 1 (TP1)
BCC203 - Estrutura de Dados II

David Cristian de Paula Rodrigues 24.2.4094

Matheus Cardoso de Araújo 24.1.4033

Pedro Souza Goularte 24.2.4084

Pedro Henrique Boseja Carolino Barbosa 24.2.4092

Professor: Guilerme Tavares Assis

Ouro Preto - MG
2025

Sumário

1 Introdução

Este trabalho apresenta a análise e implementação de diferentes métodos de pesquisa externa, incluindo o Acesso Sequencial Indexado, Árvore Binária, Árvore B e Árvore B*. Os algoritmos foram implementados em linguagem C, seguindo as especificações da disciplina de Estrutura de Dados II.

O objetivo principal é comparar o desempenho dos métodos considerando:

- número de transferências entre memória interna e externa;
- número de comparações de chaves;
- tempo de execução.

Os testes foram realizados com arquivos de diferentes tamanhos e diferentes formas de ordenação.

2 Método Sequencial Indexado

O método de Acesso Sequencial Indexado divide o arquivo em páginas contendo uma quantidade fixa de registros, armazenando em um índice auxiliar a menor chave de cada página. A busca ocorre em duas etapas: (1) localizar a página-alvo no índice e (2) procurar a chave dentro da página carregada. A implementação segue o modelo tradicional estudado em Estrutura de Dados II, utilizando um vetor de índice e leitura paginada do arquivo.

2.1 Criação do Índice de Páginas

O índice de páginas é construído utilizando a função:

```
1 int criarIndicePaginas(const char *nomeArquivo, TipoIndice tabela[],
2                        int numRegistros, long *transferencias);
```

Cada entrada do índice contém:

- a posição da página no arquivo;
- a menor chave presente na página;
- o custo de 1 transferência para leitura do primeiro elemento de cada página.

2.2 Busca no Índice

Para determinar a página onde a chave pode estar, utiliza-se:

```
1 int buscarPaginaNoIndice(int chave, TipoIndice tabela[],
2                          int numPaginas, long *comp);
```

A busca é sequencial no vetor do índice. Em cada comparação, verifica-se se a chave está entre os limites da página atual ou da próxima.

2.3 Carregamento da Página-Alvo

Após determinar a página-alvo, o programa carrega seu conteúdo:

```
int carregarPagina(const char *nomeArquivo, int numPagina,
                   PaginaAS *paginaAlvo, int numRegistros,
                   long *transferencias);
```

Cada registro lido soma uma transferência.

2.4 Busca Dentro da Página

A busca interna é linear:

```
int buscarNaPagina(int chave, PaginaAS *pag,
                   long *comp, TipoItem *resultado);
```

Se a chave estiver na página, o item é retornado via referência.

2.5 Resultados Experimentais

Nesta seção são apresentados os resultados experimentais obtidos conforme o arquivo de entrada: ordenado ascendente, ordenado descendente e misto. As tabelas completas aparecem apenas para o arquivo ascendente, pois nos demais casos a busca não localizou chaves suficientes para formar tabelas comparáveis.

2.5.1 Arquivo Ordenado Ascendente

O arquivo ascendente é o caso ideal do método, pois contém chaves de 1 até N em ordem crescente, garantindo que todas as pesquisas resultem em sucesso.

Tabela 1: Criação do índice — Arquivo Ascendente

Registros	Transferências	Tempo (s)
100	25	0.005
200	50	0.014
2000	500	0.125
20000	5000	1.242
200000	50000	13.829
2000000	500000	125.713

Criação do índice

Tabela 2: Pesquisa — Arquivo Ascendente

Registros	Comparações	Transferências	Tempo (s)
100	377	80	0.032
200	602	80	0.032
2000	4252	80	0.034
20000	43536	80	0.034
200000	546071	80	0.035
2000000	4272829	80	0.047

Pesquisa

Análise do Caso Ascendente

- Todas as chaves pesquisadas foram encontradas.
- A pesquisa sempre acessa o índice e a página do arquivo → custo fixo de 4 transferências.
- O número de comparações cresce conforme o tamanho da página e o volume de registros.
- A criação do índice é linear no número de páginas.

2.5.2 Arquivo Ordenado Descendente

- As chaves pesquisadas não pertenciam ao arquivo.
- O índice detectava imediatamente que a chave estava fora do intervalo.
- Zero transferências; apenas 1 comparação para rejeição.
- Tempo de execução desprezível.

2.5.3 Arquivo Misto

- Comportamento irregular devido à ausência de ordenação global.
- Em muitos casos, o índice retornou página inexistente (transferências = 0).
- Em outros, a pesquisa chegou à página, mas a chave não estava presente.
- Em arquivos pequenos (100–200 registros), algumas chaves foram encontradas por coincidência.

2.6 Conclusões sobre o Método Sequencial Indexado

O método se mostrou eficiente somente quando o arquivo está ordenado em ordem crescente. Para arquivos mistos ou descendentes, o desempenho é extremamente dependente da distribuição das chaves pesquisadas.

As principais conclusões são:

- Excelente desempenho para arquivos ordenados ascendentemente.
- Desempenho mínimo (quase zero custo) para arquivos descendentes, pois a rejeição é imediata.
- Alta variabilidade em arquivos mistos.
- Crescimento linear na criação do índice, mas custo de busca depende da página alvo.

3 Árvore Binária de Pesquisa

A Árvore Binária de Pesquisa é uma estrutura de dados hierárquica onde cada nó possui no máximo dois filhos (esquerdo e direito), e mantém a propriedade de que todos os valores à esquerda são menores que o nó atual, e todos à direita são maiores. A implementação para memória externa armazena os nós em arquivo binário, utilizando ponteiros de posição (offsets) para navegar entre os nós.

3.1 Estrutura do Nó em Arquivo

Cada nó é armazenado no arquivo com a seguinte estrutura:

```
typedef struct {
    Registro registro;
    long esquerda;  // -1 se nao existe
    long direita;   // -1 se nao existe
} NoArquivo;
```

Os campos `esquerda` e `direita` armazenam a posição (em número de nós) de cada filho no arquivo. O valor -1 indica ausência de filho.

3.2 Criação da Árvore

A função `criarArvoreBinaria` inicializa o arquivo da árvore:

```
void criarArvoreBinaria(const char *nomeArquivoArvore);
```

Esta função cria um nó raiz vazio (chave = -1) no início do arquivo, servindo como ponto de partida para todas as operações.

3.3 Operações de Leitura e Escrita

Duas funções auxiliares gerenciam a transferência de nós entre disco e memória:

```
NoArquivo lerNo(FILE *arquivo, long posicao);
void escreveNo(FILE *arquivo, long posicao, NoArquivo no);
```

A função `lerNo` posiciona o ponteiro do arquivo na posição especificada e lê o nó, contabilizando uma transferência. A função `escreveNo` atualiza o nó na posição indicada.

3.4 Inserção de Registros

A inserção segue o algoritmo recursivo tradicional de árvores binárias, adaptado para arquivo:

```
long inserirRecursivo(FILE *arquivo, long posicao,
                      Registro reg, long *comp);
```

O processo de inserção funciona da seguinte forma:

- Se a posição é -1 (inexistente), cria um novo nó no final do arquivo
- Caso contrário, lê o nó atual e compara as chaves
- Se a nova chave for menor, insere recursivamente à esquerda
- Se a nova chave for maior, insere recursivamente à direita
- Se as chaves forem iguais, não insere (evita duplicatas)
- Atualiza o nó pai com o ponteiro para o novo filho

Cada comparação de chaves incrementa o contador `*comp`.

3.5 Busca de Registros

A busca percorre a árvore de forma iterativa:

```
Registro* buscarEmArquivo(const char *nomeArquivoArvore,
                          int chave, long *comp,
                          long *transferencias);
```

O algoritmo de busca:

- Inicia na raiz (posição 0)
- Enquanto a posição não for -1:
  - Lê o nó atual (uma transferência)
  - Compara a chave procurada com a chave do nó
  - Se encontrou, retorna o registro
  - Se a chave procurada for menor, vai para a esquerda
  - Se a chave procurada for maior, vai para a direita
- Se chegou ao final sem encontrar, retorna NULL

Cada leitura de nó conta como uma transferência, e cada comparação incrementa o contador.

3.6 Construção da Árvore a partir de Arquivo

A função `lerArquivoBinario` constrói a árvore lendo registros de um arquivo:

```
void lerArquivoBinario(const char *nomeArquivoDados,
                       const char *nomeArquivoArvore,
                       int numRegistros, long *transferencias,
                       long *comp, double *tempo);
```

Esta função:

- Cria uma árvore binária vazia
- Lê cada registro do arquivo de dados (uma transferência por registro)
- Insere cada registro na árvore, acumulando as comparações
- Mede o tempo total de construção

3.7 Teste com 20 Chaves Aleatórias

Para o modo de teste (-T), implementou-se a função `pesquisar20AleatoriasAB`:

```
void pesquisar20AleatoriasAB(const char *nomeArquivoDados,
                             const char *nomeArquivoArvore,
                             int numRegistros);
```

Esta função:

- Seleciona 20 posições aleatórias no arquivo de dados
- Para cada posição, lê a chave correspondente
- Busca a chave na árvore binária construída
- Acumula as transferências e comparações totais
- Reporta os resultados individuais e totais

3.8 Resultados Experimentais

Nesta seção são apresentados os resultados experimentais obtidos com o uso da Árvore Binária de Pesquisa. Os testes foram realizados com arquivos contendo 100, 200, 2000, 20000, 200000 e 2000000 registros nas três situações: ordenado ascendente, ordenado descendente e misto. Todas as pesquisas envolveram 20 chaves aleatórias.

3.8.1 Arquivo Ordenado Ascendente

O arquivo ascendente contém registros em ordem crescente. Como esperado, inserções sequenciais tendem a produzir uma árvore degenerada (pior caso), próxima a uma lista encadeada, o que aumenta drasticamente a profundidade e o custo das operações.

Tabela 3: Criação da árvore — Arquivo Ascendente

Registros	**Tempo (s)**
100	3.188
200	10.451
2000	*Estouro de pilha*
20000	*Estouro de pilha*
200000	*Estouro de pilha*
2000000	*Estouro de pilha*

Criação da Árvore

Tabela 4: Pesquisa — Arquivo Ascendente

Registros	**Comparações**	**Transferências**	**Tempo (s)**
100	1966	993	0.248
200	4250	2135	0.469
2000	*Estouro de pilha*		
20000	*Estouro de pilha*		
200000	*Estouro de pilha*		
2000000	*Estouro de pilha*		

Pesquisa

Análise do Caso Ascendente

- A inserção de registros em ordem crescente gera uma árvore altamente desbalanceada, degenerando em uma estrutura próxima a uma lista encadeada.
- Observou-se **estouro de pilha** ao tentar construir a árvore a partir de 2000 registros, indicando que a implementação recursiva atinge o limite de profundidade da pilha.
- Para os tamanhos construídos com sucesso (100 e 200), o número de comparações e transferências cresce rapidamente, confirmando o comportamento degenerado com complexidade $O(n^2)$.
- O número de transferências é aproximadamente metade das comparações, pois cada nó visitado requer duas comparações (uma para verificar se é maior, outra para decidir o lado).

3.8.2 Arquivo Ordenado Descendente

O arquivo descendente produz comportamento simétrico ao caso ascendente: inserções estritamente decrescentes também conduzem a uma árvore degenerada, com consequências semelhantes para profundidade e desempenho.

Tabela 5: Criação da árvore — Arquivo Descendente

Registros	**Tempo (s)**
100	3.250
200	11.016
2000	*Estouro de pilha*
20000	*Estouro de pilha*
200000	*Estouro de pilha*
2000000	*Estouro de pilha*

Criação da Árvore

Tabela 6: Pesquisa — Arquivo Descendente

Registros	**Comparações**	**Transferências**	**Tempo (s)**
100	1724	872	0.216
200	2986	1503	0.365
2000	*Estouro de pilha*		
20000	*Estouro de pilha*		
200000	*Estouro de pilha*		
2000000	*Estouro de pilha*		

Pesquisa

Análise do Caso Descendente

- Comportamento análogo ao caso ascendente, com degeneração estrutural e estouro de pilha a partir de 2000 registros.
- Os valores de comparações e transferências são ligeiramente diferentes dos observados no caso ascendente devido à aleatoriedade das chaves pesquisadas, mas mantêm a mesma ordem de magnitude.
- Conclusão: sem mecanismos de balanceamento (AVL, Red-Black) ou implementação iterativa, a árvore binária simples é inviável para inserções ordenadas em grandes volumes.

3.8.3 Arquivo Misto

O arquivo misto contém uma distribuição mais aleatória de chaves, o que tende a produzir árvores mais equilibradas de forma natural e reduzir o custo médio das operações.

Criação da Árvore

Pesquisa

Tabela 7: Criação da árvore — Arquivo Misto

Registros	**Tempo (s)**
100	1.341
200	2.922
2000	39.487
20000	432.157
200000	*Estouro de pilha*
2000000	*Estouro de pilha*

Tabela 8: Pesquisa — Arquivo Misto

Registros	**Comparações**	**Transferências**	**Tempo (s)**
100	262	141	0.086
200	316	168	0.091
2000	522	271	0.128
20000	712	366	0.276
200000	*Estouro de pilha*		
2000000	*Estouro de pilha*		

Análise do Caso Misto

- Com dados mistos, a árvore apresenta balanceamento natural, resultando em desempenho significativamente superior aos casos ordenados.
- O número de comparações e transferências cresce de forma muito mais suave, aproximando-se do comportamento logarítmico esperado.
- Os tempos de construção aumentam com o tamanho do arquivo, mas sem o estouro de pilha observado nos casos totalmente ordenados (pelo menos até 20.000 registros).
- A diferença entre 100 e 20.000 registros nas pesquisas mostra crescimento moderado, confirmando que a árvore mantém profundidade razoável.
- Para 200.000 e 2.000.000 de registros, observou-se **estouro de pilha** durante a construção, indicando que mesmo com distribuição aleatória, a profundidade da recursão excede os limites da pilha do sistema para volumes muito grandes.

3.9 Análise de Complexidade

3.9.1 Complexidade de Construção

A construção da árvore depende diretamente da ordem de inserção dos elementos:

- **Melhor caso:** Se os elementos são inseridos de forma balanceada, a altura da árvore é $O(\log n)$ e a construção total é $O(n \log n)$
- **Pior caso:** Se os elementos são inseridos em ordem crescente ou decrescente, a árvore degenera em uma lista encadeada com altura $O(n)$, resultando em $O(n^2)$
- **Caso médio:** Para inserções aleatórias, a altura esperada é $O(\log n)$

3.9.2 Complexidade de Busca

A busca percorre da raiz até uma folha na pior das hipóteses:

- **Melhor caso:** Elemento na raiz, $O(1)$
- **Pior caso:** Elemento em uma folha de árvore degenerada, $O(n)$
- **Caso médio:** Árvore balanceada, $O(\log n)$

O número de transferências é igual ao número de nós visitados no caminho da busca.

3.10 Conclusões sobre a Árvore Binária de Pesquisa

A partir dos resultados experimentais obtidos, pode-se concluir:

3.10.1 Desempenho por Tipo de Arquivo

Arquivos Ordenados (Ascendente e Descendente)

- A árvore degenera em uma lista encadeada, com altura $O(n)$
- Complexidade de construção: $O(n^2)$
- Complexidade de busca: $O(n)$ no pior caso
- **Inviável para volumes acima de 200 registros** devido ao estouro de pilha
- Número de comparações e transferências cresce linearmente com a profundidade

Arquivos Mistos

- A árvore mantém balanceamento natural, com altura aproximada de $O(\log n)$
- Desempenho aceitável até 20.000 registros nos testes realizados
- Número de comparações cresce de forma logarítmica (262 para 100 registros, 712 para 20.000 registros)
- Tempo de pesquisa se mantém razoável mesmo com o crescimento do arquivo

3.10.2 Limitações Identificadas

- **Ausência de balanceamento:** A estrutura não possui mecanismos automáticos de balanceamento (rotações), tornando-a vulnerável à degeneração em casos ordenados.
- **Implementação recursiva:** A inserção recursiva atinge rapidamente o limite da pilha em árvores profundas, causando estouro de pilha.
- **Dependência da ordem de inserção:** O desempenho varia drasticamente conforme a ordem dos dados, sem garantias de comportamento consistente.
- **Escalabilidade limitada:** Inadequada para grandes volumes de dados, especialmente se houver alguma ordenação nos dados de entrada.

Tabela 9: Comparação entre Situações de Ordenação (20.000 registros)

Situação	Construção	Comparações	Transferências
Ascendente	Falhou	—	—
Descendente	Falhou	—	—
Misto	432.157s	712	366

3.10.3 Comparação de Métricas

3.10.4 Recomendações

Para uso prático de árvores binárias em pesquisa externa, recomenda-se:

- Implementar árvores balanceadas (AVL, Red-Black) para garantir altura logarítmica independente da ordem de inserção
- Utilizar implementação iterativa da inserção para evitar limitações da pilha de recursão
- Considerar aumento do limite de pilha do sistema quando a recursão for necessária
- Para grandes volumes de dados ordenados, preferir estruturas como Árvore B que são projetadas especificamente para esse cenário

3.10.5 Principais Conclusões

- A Árvore Binária simples apresenta excelente desempenho para dados com distribuição aleatória
- É completamente inadequada para dados ordenados sem mecanismos de balanceamento
- O estouro de pilha observado evidencia a necessidade de implementações mais robustas para uso em produção
- A variabilidade de desempenho entre diferentes ordenações (3.188s vs 432.157s para construção com dados diferentes) demonstra a falta de previsibilidade da estrutura
- Embora seja didaticamente importante, a árvore binária não balanceada não é recomendada para aplicações reais de pesquisa externa

4 Árvore B

A Árvore B é uma estrutura de dados balanceada projetada especificamente para sistemas que realizam leitura e escrita de grandes blocos de dados, como sistemas de arquivos e bancos de dados. Diferentemente da árvore binária, cada nó pode conter múltiplas chaves e múltiplos filhos, reduzindo a altura da árvore e, consequentemente, o número de acessos ao disco.

4.1 Estrutura da Página

Na implementação, cada nó é chamado de "página"e possui a seguinte estrutura:

```
#define ORDEM 50
#define MM 2 * ORDEM

typedef struct pagina {
    int n;                      // numero de chaves na pagina
    Registro registro[MM];      // vetor de registros
    struct pagina *filhos[MM+1]; // ponteiros para filhos
} Pagina;
```

Características da estrutura:

- `ORDEM`: Define a ordem da árvore (50 neste trabalho)
- `MM`: Número máximo de chaves por página ($2 \times ORDEM = 100$)
- `n`: Quantidade atual de chaves na página
- `registro[]`: Vetor contendo as chaves ordenadas
- `filhos[]`: Vetor de ponteiros para páginas filhas (MM+1 ponteiros)

4.2 Propriedades da Árvore B

A Árvore B mantém as seguintes propriedades:

- Todas as folhas estão no mesmo nível
- Cada nó interno (exceto a raiz) contém entre ORDEM e MM chaves
- A raiz tem no mínimo 1 chave (exceto se for a única página)
- Um nó com k chaves tem $k + 1$ filhos
- As chaves dentro de cada nó estão ordenadas

4.3 Pesquisa

A função de pesquisa percorre a árvore de forma iterativa:

```
Registro* pesquisa(Pagina *pagina, int chave, long *comp);
```

Algoritmo de pesquisa:

- Inicia na raiz
- Em cada página, percorre o vetor de chaves até encontrar a posição
- Se a chave está na página, retorna o registro
- Caso contrário, desce para o filho apropriado
- Continua até encontrar a chave ou chegar a uma folha

O número de comparações depende tanto do número de chaves em cada página quanto da altura da árvore.

4.4 Inserção

A inserção é mais complexa e envolve três funções principais:

4.4.1 Função InsereNaPagina

```
1 void InsereNaPagina(Pagina *ap, Registro Reg, Pagina *apDir);
```

Esta função insere um registro em uma página que possui espaço disponível:

- Move as chaves maiores para a direita
- Move os ponteiros dos filhos correspondentes
- Insere o novo registro na posição correta
- Incrementa o contador de chaves da página

4.4.2 Função Ins (Inserção Recursiva)

```
1 void Ins(Registro reg, Pagina *ap, short *cresceu,
2          Registro *regRetorno, Pagina **apRetorno, long *comp);
```

Esta é a função recursiva principal de inserção:
Caso base: Se `ap == NULL` (chegou na posição de inserção)

- Marca `*cresceu = TRUE`
- Retorna o registro a ser inserido
- Retorna `*apRetorno = NULL`

Caso recursivo:

- Procura a posição onde o registro deve ser inserido
- Verifica se a chave já existe (evita duplicatas)
- Chama recursivamente `Ins` para o filho apropriado
- Se nada "cresceu", retorna sem modificações

Tratamento do crescimento:

- Se a página tem espaço (`n < MM`):
  - Insere o registro que "subiu"
  - Marca `*cresceu = FALSE`
- Se a página está cheia (`n == MM`):
  - Cria uma nova página
  - Distribui as chaves entre a página original e a nova
  - Promove a chave do meio para o nível superior
  - Mantém `*cresceu = TRUE`

4.4.3 Divisão de Página

Quando uma página está cheia durante a inserção:

1. Cria uma nova página vazia (`apTemp`)
2. Decide se a nova chave vai na página original ou na nova
3. Move as chaves maiores que a posição ORDEM para a nova página
4. Ajusta a quantidade de chaves na página original (`ap->n = ORDEM`)
5. A chave na posição ORDEM é promovida para o pai
6. Retorna a chave promovida e o ponteiro para a nova página

4.4.4 Função Insere (Interface Principal)

```
1 void Insere(Registro reg, Pagina **ap, long *comp);
```

Esta função gerencia o caso especial de crescimento da raiz:

- Chama `Ins` para inserir o registro
- Se a chave "subiu"até o topo (`cresceu == TRUE`):
  - Cria uma nova página raiz
  - Coloca a chave promovida na nova raiz
  - A página antiga se torna filho esquerdo
  - A nova página criada se torna filho direito
  - Atualiza o ponteiro da raiz

4.5 Construção da Árvore

A função `lerArquivoArvoreB` constrói a árvore a partir do arquivo:

```
1 void lerArquivoArvoreB(const char *nomeArquivo,
2                        int numRegistros, Pagina **raiz,
3                        long *transferencias, long *comp,
4                        double *tempo);
```

Processo:

- Inicializa a raiz como NULL
- Lê cada registro do arquivo (uma transferência por registro)
- Insere cada registro na árvore usando `Insere`
- Acumula o número de comparações realizadas
- Mede o tempo total de construção

4.6 Teste com 20 Chaves Aleatórias

A função `pesquisar20Aleatorias` realiza o teste:

```
1 void pesquisar20Aleatorias(const char *nomeArquivo,
2                            int numRegistros, Pagina *raiz);
```

Funcionamento:

- Seleciona 20 posições aleatórias do arquivo
- Para cada posição, lê a chave correspondente
- Busca a chave na árvore B construída
- Acumula comparações e transferências
- Exibe resultados individuais e totais

4.7 Resultados Experimentais

Nesta seção são apresentados os resultados experimentais obtidos com a Árvore B. Os testes foram realizados com arquivos contendo 100, 200, 2000, 20000, 200000 e 2000000 registros. Devido às propriedades de balanceamento da Árvore B, os testes foram realizados apenas com arquivos mistos, uma vez que a estrutura mantém desempenho consistente independentemente da ordem de inserção.

4.7.1 Arquivo Misto

A Árvore B, por sua natureza balanceada, apresenta comportamento uniforme independente da ordem de inserção dos dados. Os testes com arquivo misto demonstram o desempenho real da estrutura.

Tabela 10: Criação da árvore — Arquivo Misto (Ordem = 50)

Registros	**Transferências**	**Comparações**	**Tempo (s)**
100	100	2452	0.001
200	200	6371	0.003
2000	2000	83743	0.045
20000	20000	1338457	0.518
200000	200000	15889753	6.312
2000000	*Killed (estouro de memória)*		

Criação da Árvore

Pesquisa (20 chaves aleatórias)

Análise do Caso Misto

- A Árvore B mantém desempenho consistente e previsível em todas as escalas testadas.

Tabela 11: Pesquisa — Arquivo Misto (Ordem = 50)

Registros	Comparações	Transferências	Tempo (s)
100	1264	20	0.000644
200	849	20	0.000403
2000	950	20	0.000238
20000	1282	20	0.000380
200000	1474	20	0.000421
2000000	*Não executado*		

- **Construção da árvore:** O número de transferências é igual ao número de registros (uma transferência por registro lido do arquivo). O número de comparações cresce conforme esperado, refletindo a necessidade de encontrar a posição correta em cada página durante a inserção.
- **Pesquisa:** O número de transferências permanece constante (20), correspondendo às 20 chaves pesquisadas. Isso demonstra a eficiência da estrutura em memória principal após a construção.
- **Número de comparações na pesquisa:** Cresce logaritmicamente, variando de 1264 (100 registros) a 1474 (200000 registros), um aumento de apenas 16% para um arquivo 2000 vezes maior.
- **Tempo de pesquisa:** Extremamente baixo e praticamente constante, variando entre 0.0002s e 0.0004s para todos os tamanhos testados.
- **Escalabilidade:** Os testes com 2.000.000 de registros foram terminados pelo sistema (killed) tanto na construção quanto na pesquisa, devido ao consumo excessivo de memória RAM. A implementação mantém toda a estrutura da árvore em memória principal, o que se torna inviável para volumes muito grandes.

4.7.2 Análise de Crescimento

Tabela 12: Taxa de crescimento das comparações (construção)

De → Para	Fator de Registros	Fator de Comparações
100 → 200	2×	2.60×
200 → 2000	10×	13.14×
2000 → 20000	10×	15.98×
20000 → 200000	10×	11.87×

Crescimento das Comparações na Construção Observa-se que o crescimento das comparações é superlinear mas subquadrático, aproximando-se do comportamento esperado $O(n \log n)$ para construção da árvore.

Crescimento das Comparações na Pesquisa O número médio de comparações por pesquisa cresce muito lentamente, demonstrando a natureza logarítmica da busca na Árvore B. A variação de 42.5 para 73.7 comparações (aumento de 73%) para um arquivo 1000 vezes maior confirma o comportamento $O(\log n)$.

Tabela 13: Comparações médias por pesquisa

Registros	Comparações/Pesquisa
100	63.2
200	42.5
2000	47.5
20000	64.1
200000	73.7

4.8 Análise de Complexidade

4.8.1 Altura da Árvore

A principal vantagem da Árvore B é sua altura logarítmica garantida:

- Para n chaves e ordem m, a altura é $O(\log_m n)$
- Com ORDEM = 50, cada página pode ter até 100 chaves
- Exemplo: 1.000.000 chaves → altura aproximada de 3 ou 4 níveis

4.8.2 Complexidade de Inserção

Tempo de execução:

- Busca da posição: $O(\log_m n)$ para descer na árvore
- Comparações em cada página: $O(m)$ para encontrar a posição
- Divisão de página: $O(m)$ no pior caso
- Total: $O(m \cdot \log_m n)$

Transferências:

- Proporcional à altura da árvore: $O(\log_m n)$
- Independente da ordem de inserção (árvore sempre balanceada)

4.8.3 Complexidade de Busca

Tempo de execução:

- Descida na árvore: $O(\log_m n)$ páginas visitadas
- Busca dentro de cada página: $O(m)$ comparações
- Total: $O(m \cdot \log_m n)$

Transferências:

- Melhor caso: 1 (chave na raiz)
- Pior caso: $O(\log_m n)$ (chave em uma folha)
- Número de transferências é sempre logarítmico

4.9 Vantagens sobre a Árvore Binária

- **Balanceamento garantido:** A árvore permanece balanceada independentemente da ordem de inserção, evitando a degeneração em lista
- **Altura reduzida:** Com múltiplas chaves por nó, a altura é muito menor que uma árvore binária equivalente
- **Menos transferências:** Número de acessos ao disco é proporcional à altura, que é logarítmica na base da ordem
- **Aproveitamento do cache:** Leitura de múltiplas chaves de uma vez aproveita melhor as operações de I/O em bloco

4.10 Conclusões sobre a Árvore B

A partir dos resultados experimentais obtidos, pode-se concluir:

4.10.1 Desempenho Geral

- A Árvore B demonstrou desempenho consistente e altamente escalável para todos os tamanhos testados com sucesso (até 200.000 registros).
- O balanceamento automático garante altura logarítmica independentemente da ordem de inserção dos dados.
- A busca apresenta eficiência excepcional, com tempo praticamente constante mesmo para arquivos grandes.

4.10.2 Análise de Complexidade Observada

Construção

- **Transferências:** Crescimento linear ($O(n)$), uma transferência por registro lido.
- **Comparações:** Crescimento próximo de $O(n \log n)$, conforme esperado teoricamente. As variações observadas (fatores entre 11.87× e 15.98× para aumentos de 10× nos dados) são consistentes com a inserção em árvore balanceada.
- **Tempo:** Cresce de forma aproximadamente linear com o número de registros, variando de 0.001s (100 registros) a 6.312s (200.000 registros).

Pesquisa

- **Transferências:** Constante (20), independente do tamanho do arquivo, pois a árvore já está em memória principal.
- **Comparações:** Crescimento logarítmico, com média por pesquisa aumentando de 63.2 para 73.7 (apenas 16.6% de aumento) quando o arquivo cresce de 100 para 200.000 registros (2000×).
- **Tempo:** Praticamente constante (0.0003s), demonstrando excelente desempenho para operações de consulta.

4.10.3 Vantagens Observadas

- **Previsibilidade:** Desempenho consistente independente da ordem dos dados.
- **Escalabilidade:** Crescimento controlado em todas as métricas, permitindo uso em arquivos grandes.
- **Eficiência em buscas:** Tempo de pesquisa extremamente baixo e quase independente do tamanho do arquivo.
- **Altura controlada:** Com ORDEM=50, cada página pode ter até 100 chaves, resultando em árvores muito baixas (poucos níveis).
- **Aproveitamento de memória:** Múltiplas chaves por página reduzem o número de acessos necessários.

4.10.4 Limitações Identificadas

- **Consumo de memória:** Para 2.000.000 de registros, todos os testes (tanto construção quanto pesquisa) foram terminados pelo sistema (killed), indicando consumo excessivo de RAM. A árvore B nesta implementação mantém toda a estrutura em memória principal, o que se torna proibitivo para volumes muito grandes. Cada página aloca espaço para até 100 registros (ORDEM=50), e com milhões de registros, o número de páginas alocadas excede a memória disponível.
- **Complexidade de implementação:** O código de inserção é mais complexo que estruturas simples, envolvendo divisão de páginas e promoção de chaves.
- **Overhead na construção:** O tempo de construção cresce mais rapidamente que em estruturas mais simples devido às operações de manutenção do balanceamento.

4.10.5 Comparação com Árvore Binária

Tabela 14: Árvore B vs Árvore Binária (arquivo misto)

Registros	**Tempo Construção (s)**		**Comp. Pesquisa**	
	Binária	**Árvore B**	**Binária**	**Árvore B**
100	1.341	0.001	262	1264
200	2.922	0.003	316	849
2000	39.487	0.045	522	950
20000	432.157	0.518	712	1282

Observações da comparação:

- **Tempo de construção:** Árvore B é significativamente mais rápida, sendo até 834× mais rápida para 20.000 registros (432.157s vs 0.518s).
- **Comparações na pesquisa:** Árvore Binária tem menos comparações para tamanhos pequenos devido à altura menor, mas cresce mais rapidamente. Para 20.000 registros, a diferença já é pequena (712 vs 1282).
- **Escalabilidade:** Árvore Binária falhou completamente para arquivos ordenados, enquanto a Árvore B funciona independentemente da ordem.

4.10.6 Adequação para Pesquisa Externa

A Árvore B demonstra ser uma estrutura ideal para pesquisa externa devido a:

- Altura logarítmica garantida, minimizando acessos ao disco
- Páginas com múltiplas chaves, aproveitando leituras em bloco
- Desempenho previsível e consistente
- Excelente relação entre tempo de construção e eficiência de busca

No entanto, para volumes muito grandes (milhões de registros), a implementação precisa ser adaptada para trabalhar com páginas em disco ao invés de manter toda a estrutura em memória.

4.10.7 Recomendações

- Para arquivos até 200.000 registros, a implementação atual é adequada e oferece excelente desempenho.
- Para volumes maiores, considerar implementação com páginas em disco (Árvore B externa verdadeira).
- A ordem da árvore (ORDEM=50) mostrou-se apropriada para os testes, mas pode ser ajustada conforme o tamanho típico das páginas de disco.
- Para sistemas que exigem inserções frequentes, a Árvore B* pode oferecer melhor aproveitamento de espaço nas páginas.

5 Árvore B*

A Árvore B* é uma variação otimizada da Árvore B que visa melhorar o aproveitamento de espaço nas páginas. Enquanto a Árvore B tradicional permite que páginas internas tenham de 50% a 100% de ocupação, a Árvore B* garante que as páginas permaneçam pelo menos 2/3 cheias, reduzindo o desperdício de espaço e melhorando a eficiência do cache.

5.1 Diferenças em Relação à Árvore B

As principais diferenças entre Árvore B* e Árvore B são:

- **Taxa de ocupação mínima:** Árvore B* mantém páginas com pelo menos 66.7% de ocupação (2/3), enquanto Árvore B permite 50%.
- **Redistribuição de chaves:** Antes de dividir uma página cheia, a Árvore B* tenta redistribuir chaves com páginas irmãs adjacentes.
- **Divisão 2-para-3:** Quando uma página e sua irmã estão cheias, a Árvore B* divide essas duas páginas em três páginas, cada uma com aproximadamente 2/3 de ocupação.
- **Melhor aproveitamento de espaço:** Resultado em árvores mais compactas e com menos níveis para o mesmo número de chaves.

5.2 Estrutura e Implementação

A estrutura da página e as operações básicas são similares à Árvore B, mantendo a mesma definição de ORDEM e MM. A diferença principal está no algoritmo de inserção, que implementa a redistribuição e a divisão 2-para-3.

5.3 Resultados Experimentais

Os testes foram realizados com arquivos mistos contendo 100, 200, 2000, 20000 e 200000 registros, utilizando a mesma ordem (ORDEM = 50) da Árvore B para permitir comparação direta.

5.3.1 Arquivo Misto

Tabela 15: Criação da árvore B* — Arquivo Misto (Ordem = 50)

Registros	**Transferências**	**Comparações**	**Tempo (s)**
100	100	2452	0.003
200	200	6561	0.003
2000	2000	98288	0.050
20000	20000	2000405	0.648
200000	200000	25662583	7.050
2000000	*Killed (estouro de memória)*		

Criação da Árvore

Tabela 16: Pesquisa B* — Arquivo Misto (Ordem = 50)

Registros	**Comparações**	**Transferências**	**Tempo (s)**
100	1081	20	0.000657
200	562	20	0.000692
2000	1099	20	0.000242
20000	1148	20	0.000342
200000	1937	20	0.000524
2000000	*Killed (estouro de memória)*		

Pesquisa (20 chaves aleatórias)

Análise do Caso Misto

- A Árvore B* apresenta comportamento similar à Árvore B, com desempenho consistente e previsível.
- **Construção:** O número de transferências permanece igual ao número de registros. O número de comparações é ligeiramente superior ao da Árvore B, reflexo das operações adicionais de redistribuição de chaves.
- **Pesquisa:** O número de comparações na pesquisa é comparável ao da Árvore B, com algumas variações devido à diferente distribuição de chaves resultante do algoritmo de divisão 2-para-3.

- **Tempo de pesquisa:** Extremamente baixo, variando entre 0.0002s e 0.0006s, similar à Árvore B.
- **Escalabilidade:** Como esperado, o teste com 2.000.000 de registros também foi terminado pelo sistema (killed) devido ao consumo de memória, comportamento idêntico à Árvore B.

5.3.2 Comparação Árvore B vs Árvore B*

Tabela 17: Comparação de construção: Árvore B vs B*

2*Registros	Comparações		Tempo (s)	
	B	B*	B	B*
100	2452	2452	0.001	0.003
200	6371	6561	0.003	0.003
2000	83743	98288	0.045	0.050
20000	1338457	2000405	0.518	0.648
200000	15889753	25662583	6.312	7.050

Construção da Árvore

Tabela 18: Comparação de pesquisa: Árvore B vs B*

2*Registros	Comparações		Tempo (s)	
	B	B*	B	B*
100	1264	1081	0.000644	0.000657
200	849	562	0.000403	0.000692
2000	950	1099	0.000238	0.000242
20000	1282	1148	0.000380	0.000342
200000	1474	1937	0.000421	0.000524

Pesquisa

5.3.3 Análise Comparativa

Número de Comparações

- **Na construção:** A Árvore B* realiza mais comparações que a Árvore B, especialmente em arquivos maiores. Para 200.000 registros, são 61% mais comparações (25.6M vs 15.8M). Isso se deve às operações adicionais de redistribuição de chaves antes das divisões.
- **Na pesquisa:** Os resultados são mistos. Para alguns tamanhos, a B* tem menos comparações (200 registros: 562 vs 849), em outros tem mais (200k registros: 1937 vs 1474). Isso reflete diferentes estruturas internas resultantes dos algoritmos de divisão.

Tempo de Execução

- **Construção:** A Árvore B* é ligeiramente mais lenta que a Árvore B, com diferença crescente para arquivos maiores (7.050s vs 6.312s para 200k). O overhead das operações de redistribuição justifica essa diferença.
- **Pesquisa:** Os tempos são praticamente equivalentes, ambas com desempenho excepcional (ordem de microsegundos).

5.4 Análise Teórica vs Prática

5.4.1 Vantagens Teóricas da B*

- Melhor aproveitamento de espaço (mínimo 66.7% vs 50%)
- Árvores mais compactas com menos níveis
- Menos divisões de páginas no longo prazo
- Melhor desempenho em situações de alta taxa de inserção

5.4.2 Observações Práticas

- O overhead das operações de redistribuição resulta em mais comparações durante a construção inicial.
- Para testes de construção única seguida de pesquisas, a Árvore B tradicional mostrou-se mais eficiente.
- As vantagens da B* se tornariam mais evidentes em cenários com:
  - Inserções e remoções contínuas
  - Limitações críticas de espaço em disco
  - Necessidade de maximizar cache hits

5.5 Conclusões sobre a Árvore B*

5.5.1 Desempenho Observado

- A Árvore B* mantém as características fundamentais de balanceamento e altura logarítmica da Árvore B.
- O desempenho de pesquisa é comparável entre as duas estruturas.
- A construção da B* é 10-18% mais lenta devido ao overhead das operações de redistribuição.
- Ambas estruturas sofrem do mesmo problema de consumo de memória para volumes muito grandes (2M de registros).

5.5.2 Trade-offs Identificados

- **Custo de construção vs economia de espaço:** A B* paga um preço em tempo de construção para obter melhor ocupação de páginas.
- **Complexidade de implementação:** O código da B* é mais complexo, com lógica adicional para redistribuição e divisão 2-para-3.
- **Cenário de uso:** Árvore B é mais eficiente para cargas de trabalho com poucas modificações após construção inicial; B* é melhor para sistemas com alta taxa de inserções/remoções contínuas.

5.5.3 Recomendações

- **Use Árvore B quando:**
  - A construção inicial é crítica para o desempenho
  - A estrutura é construída uma vez e consultada muitas vezes
  - Simplicidade de implementação é importante
- **Use Árvore B* quando:**
  - O espaço em disco é limitado e crítico
  - O sistema tem alta taxa de inserções e remoções contínuas
  - Maximizar a ocupação de páginas é prioritário
  - O overhead na construção é aceitável

5.5.4 Limitações Comuns

Ambas estruturas compartilham a limitação de consumo de memória na implementação atual, que mantém toda a árvore em RAM. Para uso em produção com grandes volumes, seria necessário implementar paginação em disco, transformando-as em verdadeiras estruturas de pesquisa externa.

6 Conclusão

Este relatório apresenta a primeira parte da análise dos métodos de pesquisa externa. Conforme os próximos códigos forem enviados, as seções correspondentes serão adicionadas com explicações detalhadas e tabelas de resultados.