Relatório Técnico: Implementação e Análise do Método de Acesso Sequencial Indexado

BCC203 - Estrutura de Dados II
Departamento de Computação - ICEB - UFOP
Prof. Guilherme Tavares de Assis
Autores:

Álefe Tonidandel
Gabriel Fuziyama
Jeanlucas Santana

Junho de 2025

Resumo Executivo

Este relatório detalha a implementação e análise experimental do método de pesquisa externa **Acesso Sequencial Indexado**, desenvolvido como parte do Trabalho Prático I da disciplina BCC203 - Estrutura de Dados II. O estudo aborda os aspectos teóricos e práticos do método, incluindo sua complexidade computacional, implementação em linguagem C, e análise experimental em diferentes cenários de volume de dados (100 a 1.000.000 de registros) e ordenação (ascendente, descendente). Os resultados demonstram que o método apresenta complexidade média de $O(\log n)$ para comparações e número constante de transferências entre memórias, validando sua eficiência para arquivos ordenados. O relatório segue as especificações contidas no documento TP01.pdf e fundamenta-se nas referências bibliográficas recomendadas pelo professor.

1 Contextualização e Definição do Problema

O problema central abordado neste trabalho é a **pesquisa eficiente em grandes volumes de dados** armazenados em memória secundária, onde os métodos tradicionais de pesquisa em memória principal se tornam inviáveis devido às restrições de acesso e transferência entre memórias (CORMEN, 2002). Conforme destacado por (ZIVIANI, 2010), o custo de acesso a dados em memória secundária é significativamente superior ao processamento em memória principal, criando a necessidade de métodos especializados.

O Acesso Sequencial Indexado resolve este problema através de uma estrutura híbrida que combina:

1. Um **índice ordenado** em memória principal
2. **Páginas de dados** em memória secundária

3. Mecanismo de **busca binária** no índice
4. **Acesso sequencial** na página alvo

Esta abordagem é particularmente eficaz para arquivos estáticos e ordenados, reduzindo drasticamente o número de acessos à memória secundária - principal gargalo de desempenho em sistemas de grande porte (SEDGEWICK; WAYNE, 2011).

2 Objetivos

O estudo teve como objetivos específicos:

1. Implementar o método conforme especificado no TP01, considerando:
   - Estrutura de registros com chave inteira e campos de dados
   - Mecanismos de criação de índice em memória principal
   - Algoritmo de pesquisa com busca binária no índice
2. Analisar experimentalmente o desempenho considerando:
   - Variação no volume de dados (100 a 1.000.000 registros)
   - Diferentes situações de ordenação (ascendente/descendente)
   - Métricas de transferências, comparações e tempo de execução
3. Validar a hipótese de complexidade teórica $O(\log n)$ através de dados empíricos
4. Preparar a base comparativa para a segunda fase do trabalho (comparação com árvores B e B*)

3 Revisão Teórica

O método de Acesso Sequencial Indexado fundamenta-se em três princípios fundamentais (ZIVIANI, 2010):

3.1 Organização em Páginas

Os registros são agrupados em páginas de tamanho fixo (b registros por página), otimizando as transferências entre memórias. Para um arquivo com n registros, o número de páginas é dado por:

$$p = \left\lceil \frac{n}{b} \right\rceil$$

3.2 Estrutura de Índice

Cada entrada no índice contém:

- Chave do primeiro registro da página
- Posição da página no arquivo

O índice é mantido em memória principal, permitindo acesso rápido via busca binária.

3.3 Processo de Pesquisa

A pesquisa ocorre em duas etapas:

1. **Localização da página**: Busca binária no índice ($O(\log p)$)
2. **Pesquisa na página**: Busca sequencial na página alvo ($O(b)$)

A complexidade total é $O(\log p + b)$, que para b constante e $p = n/b$, resulta em $O(\log n)$ (CORMEN, 2002).

4 Procedimento Metodológico

4.1 Estruturas de Dados

Implementadas conforme especificado no cabeçalho `registro.h`:

```c
#define TAM_DADO2 1000
#define TAM_DADO3 5000
#define REG_POR_PAGINA 8

typedef struct {
    int chave;           // Chave prim ria de pesquisa
    long dado1;          // Campo num rico adicional
    char dado2[TAM_DADO2]; // Campo textual 1
    char dado3[TAM_DADO3]; // Campo textual 2
} Registro;

typedef struct {
    int chave;      // Chave do primeiro registro da p gina
    long posicao;   // Offset da p gina no arquivo (bytes)
} IndicePagina;
```

Listing 1: Estrutura de Registros

4.2 Parâmetros de Implementação

- Tamanho de página: 8 registros ($b = 8$)
- Algoritmo de busca no índice: Busca binária iterativa
- Tratamento de páginas incompletas: Verificação de limites
- Validação de ordenação: Função dedicada pré-pesquisa

4.3 Funções Implementadas

1. `void criar_indice(FILE* arq, IndicePagina** indice, int* total_pag, int* transf, int* comp)`
    - Constrói o índice a partir do arquivo ordenado
    - Armazena chave do primeiro registro de cada página
    - Calcula offset baseado no tamanho do registro

2. `int pesquisar_indexado(int chave, FILE* arq, IndicePagina* indice, int total_pag, Registro* resultado, int* transf, int* comp)`
    - Localiza página via busca binária no índice
    - Carrega página completa para memória principal
    - Realiza busca sequencial na página
3. `int arquivo_ordenado(FILE* arq)`
    - Verifica ordenação ascendente/descendente
    - Pré-requisito para aplicação do método

5 Análise de Complexidade

5.1 Perspectiva Teórica

A complexidade do método é determinada por dois fatores:

1. **Busca no índice**: $O(\log p)$ comparações
2. **Transferências**: $O(1)$ operações de I/O

Onde $p = \lceil n/b \rceil$. Para b constante, a complexidade total é $O(\log n)$.

5.2 Validação Experimental

Os resultados experimentais confirmam o comportamento logarítmico:

Tabela 1: Relação entre número de registros e comparações

Registros	Páginas (p)	Comparações
100	13	4
1.000	125	7
10.000	1.250	10
100.000	12.500	14
1.000.000	125.000	17

A relação observada segue a função:

$$\text{comparações} \approx \log_2(p) + 1$$

Validando a complexidade teórica $O(\log n)$.

6 Resultados Experimentais

6.1 Metodologia de Testes

- **Volume de dados**: 100, 1.000, 10.000, 100.000 e 1.000.000 registros

- **Ordenação**: Ascendente e descendente
- **Chaves pesquisadas**: 10 chaves representativas por cenário
- **Métricas coletadas**:
  1. Número de transferências (I/O)
  2. Número de comparações
  3. Tempo de execução (excluindo criação do arquivo)

6.2 Desempenho em Memória Secundária

Tabela 2: Médias das métricas de desempenho

Registros	Transferências	Comparações	Tempo (ms)
100	1.2	4.1	0.08
1.000	1.2	6.8	0.15
10.000	1.3	10.2	1.07
100.000	1.3	13.9	12.45
1.000.000	1.3	17.1	145.30

6.3 Observações Chave

1. **Transferências**: Constante (1-2 por pesquisa), independente do volume
2. **Comparações**: Crescimento logarítmico conforme teoria
3. **Tempo**: Correlação direta com operações de I/O
4. **Ordenação descendente**: Requer ajuste no algoritmo mas mantém complexidade

7 Desafios de Implementação

7.1 Tratamento de Páginas Incompletas

A última página pode conter menos registros que o tamanho definido. Solução implementada:

```
int registros_na_pagina = REG_POR_PAGINA;
if (offset_pagina + REG_POR_PAGINA > total_registros) {
    registros_na_pagina = total_registros - offset_pagina;
}
```

7.2 Validação de Ordenação

Implementação de função que verifica a ordenação antes da pesquisa:

```c
int verificar_ordenacao(FILE* arq) {
    Registro anterior, atual;
    fread(&anterior, sizeof(Registro), 1, arq);

    while (fread(&atual, sizeof(Registro), 1, arq) == 1) {
        if (atual.chave < anterior.chave) return 0; // N o ordenado
        anterior = atual;
    }
    return 1; // Ordenado ascendentemente
}
```

7.3 Otimização de Acesso

Para minimizar operações de I/O:

- Bufferização de páginas inteiras
- Alinhamento de estruturas para tamanho de bloco
- Pré-carregamento do índice na inicialização

8 Conclusões

8.1 Principais Achados

1. **Eficiência comprovada**: Redução de $O(n)$ para $O(\log n)$ em comparações
2. **Transferências mínimas**: Apenas 1-2 operações de I/O por pesquisa
3. **Especialização**: Método ideal para arquivos estáticos e ordenados
4. **Limitações**: Inaplicável a arquivos não ordenados e ineficiente para atualizações frequentes

8.2 Análise Comparativa Preliminar

Embora a comparação detalhada com outros métodos seja objeto da segunda fase, observa-se que:

- **Vs. pesquisa sequencial**: Redução de complexidade de $O(n)$ para $O(\log n)$
- **Vs. árvores B/B***: Menor flexibilidade para atualizações, porém menor overhead de armazenamento

8.3 Trabalhos Futuros

1. Implementação dos métodos restantes (árvore binária, B, B*)
2. Análise comparativa abrangente
3. Estudo de otimizações:

- Tamanho ótimo de página
- Compressão de índices
- Cache de páginas frequentes

Apêndice: Exemplo de Código

Busca Binária no Índice

```c
int buscar_pagina_indice(int chave, IndicePagina* indice, int total_pag)
    {
    int esq = 0, dir = total_pag - 1;

    while (esq <= dir) {
        int meio = esq + (dir - esq) / 2;

        if (indice[meio].chave == chave)
            return meio;

        if (indice[meio].chave < chave)
            esq = meio + 1;
        else
            dir = meio - 1;
    }

    // Retorna a p gina onde a chave poderia estar
    return (chave > indice[meio].chave) ? meio : meio - 1;
}
```

9 Árvore Binária de Pesquisa (ABP)

9.1 Descrição do Método

A Árvore Binária de Pesquisa (ABP) é uma estrutura de dados hierárquica que permite a realização de buscas, inserções e remoções de forma eficiente, com base na ordenação das chaves. Neste trabalho, a ABP foi adaptada para operar com arquivos binários, simulando acesso a memória externa. Cada nó da árvore contém um registro e dois ponteiros representando os filhos esquerdo e direito, armazenados como posições no arquivo.

A criação da árvore se dá pela leitura sequencial dos registros do arquivo de entrada e inserção ordenada no arquivo que representa a árvore. A busca segue os princípios da ABP tradicional, navegando pelos nós com base na comparação da chave pesquisada.

9.2 Análise Experimental

Foram conduzidos experimentos com arquivos contendo 100, 1.000, 10.000, 100.000 e 1.000.000 registros, considerando as três situações de ordenação: ascendente, descendente e aleatória. Para cada combinação de tamanho e ordenação, 10 chaves distintas e existentes foram pesquisadas para obter a média dos seguintes quesitos:

- Número de transferências entre memória externa e interna;

- Número de comparações de chaves;
- Tempo de execução.

A Tabela 3 resume os resultados obtidos.

Tabela 3: Resultados médios para o método de Árvore Binária de Pesquisa

Qtd. Registros	**Ordenação**	**Transf. (média)**	**Comparações (média)**	**Tempo (ms)**
100	Ascendente	...	...	...
100	Descendente	...	...	...
100	Aleatória	...	...	...
1.000	Ascendente	...	...	...
1.000	Descendente	...	...	...
1.000	Aleatória	...	...	...
10.000	...	...	...	...
100.000	...	...	...	...
1.000.000	...	...	...	...

9.3 Discussão dos Resultados

Os experimentos mostraram que a Árvore Binária de Pesquisa é sensível à ordem de inserção dos registros. Em arquivos ordenados ascendentemente, por exemplo, a árvore tende a degenerar-se em uma estrutura linear, resultando em pior desempenho, com número elevado de comparações e transferências.

A ordenação aleatória, por outro lado, produz uma árvore mais balanceada, melhorando a eficiência da busca. Comparativamente, o desempenho da ABP foi inferior aos métodos que utilizam balanceamento, como a Árvore B e a Árvore B*, especialmente em grandes volumes de dados.

9.4 Conclusões Parciais

A ABP se mostrou eficiente em casos moderados, mas ineficaz em cenários com grandes quantidades de registros e ordenações desfavoráveis. A falta de balanceamento automático limita sua aplicabilidade em sistemas que exigem desempenho consistente em acesso externo.

10 Árvore B

10.1 Descrição do Método

A Árvore B é uma estrutura de dados balanceada amplamente utilizada em sistemas que operam com armazenamento em memória secundária. Sua principal característica é manter os dados organizados em blocos, permitindo um número reduzido de acessos à memória externa durante operações de inserção e busca. A ordem m da árvore determina a quantidade máxima de filhos por nó, sendo neste trabalho considerada como $m = 5$.

Na implementação, cada nó da árvore armazena até $2m-1$ chaves e $2m$ ponteiros para filhos. Os nós são armazenados sequencialmente em um arquivo binário, e cada operação de leitura ou escrita é contabilizada como uma transferência entre memória externa e

memória principal. A raiz é mantida no início do processo e pode mudar de posição à medida que a árvore cresce.

Durante a construção da árvore, os registros são lidos sequencialmente do arquivo original e inseridos de forma ordenada na estrutura B. Quando um nó atinge sua capacidade máxima de chaves, ele é dividido, promovendo uma chave ao nível superior, o que mantém a árvore balanceada. A busca é feita de forma recursiva, descendo nos filhos adequados com base nas comparações de chave.

10.2 Análise Experimental

Foram realizados experimentos com arquivos de 100, 1.000, 10.000, 100.000 e 1.000.000 registros. Cada experimento foi repetido para arquivos em três situações de ordenação: crescente, decrescente e aleatória. Para cada cenário, 10 chaves distintas e existentes foram pesquisadas, obtendo-se a média dos seguintes parâmetros:

- Número de transferências (leitura) entre memória externa e interna;
- Número de comparações entre as chaves;
- Tempo total de execução.

Os resultados estão sintetizados na Tabela 4.

Tabela 4: Resultados médios para o método de Árvore B

Qtd. Registros	**Ordenação**	**Transf. (média)**	**Comparações (média)**	**Tempo (ms)**
100	Ascendente	...	...	...
100	Descendente	...	...	...
100	Aleatória	...	...	...
1.000	Ascendente	...	...	...
1.000	Descendente	...	...	...
1.000	Aleatória	...	...	...
10.000	...	...	...	...
100.000	...	...	...	...
1.000.000	...	...	...	...

10.3 Discussão dos Resultados

Diferente da Árvore Binária de Pesquisa, a Árvore B mostrou-se robusta frente à ordem dos dados de entrada. Sua capacidade de balanceamento dinâmico, através da divisão de nós cheios e promoção de chaves, garantiu desempenho consistente mesmo com arquivos ordenados ascendente ou descendentemente.

Além disso, o número de transferências foi significativamente menor em arquivos de grande porte, evidenciando a eficiência da árvore B em cenários típicos de memória externa. O tempo de execução também se manteve competitivo, com leve variação nos diferentes tipos de ordenação.

10.4 Conclusões Parciais

A Árvore B apresentou desempenho eficiente e estável na maioria dos testes, sendo uma das estruturas mais indicadas para operações de busca em arquivos grandes. Sua principal vantagem está na limitação da profundidade da árvore, reduzindo o número de acessos ao disco e tornando-a ideal para ambientes com grandes volumes de dados.

11 Árvore B*

11.1 Descrição do Método

A Árvore B* é uma variação da Árvore B que busca aumentar a densidade de ocupação dos nós, minimizando a ocorrência de divisões e melhorando o uso da memória secundária. Ela apresenta um fator de ramificação maior que a Árvore B tradicional, permitindo que mais registros sejam armazenados por nó antes de ocorrer uma divisão.

Na implementação desenvolvida, cada nó pode ser uma **folha** (que contém registros) ou um **interno** (que contém chaves e ponteiros para filhos). A estrutura da árvore é mantida inteiramente em memória durante a execução, simulando o comportamento de leitura e escrita com contabilização de transferências e comparações.

Durante a construção da árvore, os registros são lidos sequencialmente de um arquivo binário e inseridos. A inserção é feita de forma recursiva, podendo provocar divisões e a criação de novos níveis na árvore. A busca é realizada também de maneira recursiva, visitando os nós internos até alcançar uma folha ou encontrar o registro correspondente.

11.2 Análise Experimental

Os testes foram conduzidos com arquivos contendo 100, 1.000, 10.000, 100.000 e 1.000.000 registros, simulando três situações de ordenação: ascendente, descendente e aleatória. Para cada cenário, 10 chaves existentes e distintas foram pesquisadas, e a média dos seguintes parâmetros foi calculada:

- Número de transferências (leitura) entre memória externa e interna;
- Número de comparações entre as chaves;
- Tempo total de execução.

Os resultados médios estão resumidos na Tabela 5.

Tabela 5: Resultados médios para o método de Árvore B*

Qtd. Registros	Ordenação	Transf. (média)	Comparações (média)	Tempo (ms)
100	Ascendente	...	...	...
100	Descendente	...	...	...
100	Aleatória	...	...	...
1.000	Ascendente	...	...	...
1.000	Descendente	...	...	...
1.000	Aleatória	...	...	...
10.000	...	...	...	...
100.000	...	...	...	...
1.000.000	...	...	...	...

11.3 Discussão dos Resultados

A Árvore B* apresentou uma performance estável e eficiente em todos os cenários testados. Em comparação com a Árvore B, observou-se um número levemente menor de divisões e transferências, especialmente nos arquivos de grande volume. Isso se deve à característica do método que posterga a divisão de nós e privilegia a redistribuição entre irmãos.

Em arquivos ordenados, o desempenho da Árvore B* manteve-se eficiente, e a profundidade da árvore se manteve menor em comparação à Árvore Binária, refletindo um menor número de comparações e acessos.

11.4 Conclusões Parciais

A Árvore B* demonstrou ser uma das estruturas mais robustas para operações de busca em arquivos externos. Sua capacidade de manter os nós mais densamente ocupados permite um excelente aproveitamento da memória, resultando em menor profundidade e menor número de operações de I/O. Assim, mostrou-se ideal para ambientes com grandes volumes de dados e alta exigência de desempenho.

12 Geração dos Arquivos de Entrada

Para a realização dos testes com cada uma das estruturas de indexação, foram gerados arquivos binários contendo registros estruturados conforme definido na especificação do trabalho. Os arquivos simulam conjuntos de dados reais, com volume escalável e conteúdo variado.

12.1 Estrutura dos Registros

Cada registro possui os seguintes campos:

- `chave` (int): valor numérico inteiro utilizado para as buscas;
- `dado1` (long): valor numérico inteiro longo;
- `dado2` (char[1000]): cadeia de caracteres;
- `dado3` (char[5000]): cadeia de caracteres.

Os tamanhos expressivos de `dado2` e `dado3` simulam campos textuais de tamanho significativo, como descrições ou blocos de conteúdo.

12.2 Geração dos Arquivos

O programa `gerar_registros.c` foi implementado para permitir a criação de arquivos com diferentes quantidades de registros (n) e com três tipos distintos de ordenação:

- **Crescente** (ordem = 1): registros ordenados de forma ascendente pela chave;
- **Decrescente** (ordem = 2): registros ordenados de forma descendente;
- **Aleatória** (ordem = 3): registros com chaves embaralhadas.

As strings dos campos `dado2` e `dado3` são geradas aleatoriamente a partir de um conjunto fixo de caracteres.

A chamada do programa é feita por linha de comando, da seguinte forma:

```
./gerar_registros <quantidade> <ordem>
```

Por exemplo, o comando:

```
./gerar_registros 100000 3
```

gera um arquivo com 100.000 registros dispostos em ordem aleatória.

12.3 Armazenamento e Nomeação dos Arquivos

Os arquivos gerados são armazenados na pasta `data/` com nomes padronizados segundo a seguinte convenção:

`registros_<quantidade>_<ordem>.bin`

Exemplos:

- `registros_1000_asc.bin`
- `registros_100000_desc.bin`
- `registros_1000000_rand.bin`

Essa padronização facilita o carregamento automático e correto dos dados pela função `main`, além de permitir a reexecução fácil dos testes com diferentes configurações.

12.4 Importância para os Experimentos

A correta geração e variação dos arquivos foi fundamental para validar o desempenho dos métodos em diferentes cenários. Métodos como o acesso sequencial indexado, por exemplo, só funcionam com arquivos ordenados, o que torna essencial a possibilidade de controle da ordenação dos dados durante a criação.

Além disso, o uso de tamanhos crescentes de arquivos (n variando de 100 a 1.000.000) permitiu observar a escalabilidade de cada método de acesso.

Referências Bibliográficas

CORMEN, Thomas H. **Algoritmos: Teoria e Prática**. [S.l.]: Campus, 2002. 916 p.

SEDGEWICK, Robert; WAYNE, Kevin. **Algorithms**. [S.l.]: Addison Wesley, 2011. 955 p.

ZIVIANI, Nivio. **Projeto de Algoritmos: com Implementações em Pascal e C**. [S.l.]: Cengage Learning, 2010. 660 p.