Universidade Federal de Ouro Preto - UFOP
Departamento de Computação - DECOM
Ciência da Computação

Disciplina: BCC203 - Estrutura de Dados II

Métodos de Pesquisa Externa

Trabalho Prático I

Bernardo Sant'Anna Costa
Bruno de Melo Rangel
Henrique Sousa Lauar
Juliana Aparecida Borges
Pedro Parentoni de Almeida
Rafael Stylita Duarte

Ouro Preto -MG
2024

Sumário

1 Introdução

Neste relatório, apresentamos um estudo detalhado dos métodos de pesquisa externa discutidos em sala de aula. Na primeira fase implementamos esses métodos com foco em possíveis melhorias e na segunda fase realizamos os testes solicitados para avaliar sua eficácia em diferentes cenários, como arquivos em ordem crescente, decrescente e ordem aleatória, com diferentes quantidades de registros.

Os métodos abordados incluem:

- **Acesso Sequencial Indexado**: Implementado em memória externa é válido apenas para arquivos em ordem crescente;
- **Árvore Binária**: Adequada à memória externa;
- **Árvore B**: Implementado em memória interna;
- **Árvore B***: Implementado em memória interna;

1.1 Objetivos

Os objetivos deste trabalho prático são:

- Realizar a implementação em Linguagem C dos métodos mencionados, considerando arquivos binários de registros;
- A construção de índices em memória, quando necessário;
- Conduzir uma análise experimental que avalie o desempenho dos métodos mencionados, considerando aspectos como o número de transferência de registros entre a memória externa e interna (leituras), número de comparações entre chaves de pesquisa e tempo de execução;

1.2 Ferramentas Utilizadas

- Ambiente de implementação do trabalho: Visual Studio Code;
- Ambiente de criação da documentação: LaTeX;
- Linguagem utilizada para implementação: C.

1.3 Especificações da Maquina

As especificações das máquinas utilizadas para se realizar os testes são as seguintes:

- AMD Ryzen 7 pro 2700u w/ radeon vega mobile gfx × 8 16GB Ram
- Intel Core™ i5-7200U CPU @ 2.50GHz × 4 8GB Ram

1.4 Instruções de compilação

Visando a facilidade e praticidade na compilação do trabalho foi criado um arquivo MakeFile. Garantindo assim uma simplificação do comando para compilar o projeto:

Compilando o projeto

```
make
```

De acordo com o solicitado pelo trabalho, o comando para testes da implementação é apresentado a seguir:

Realizando as buscas

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

onde:

- <método>
  - 1 - Acesso Sequencial Indexado
  - 2 - Árvore Binária
  - 3 - Árvore B
  - 4 - Árvore B*
- <quantidade>
  - 1 a 1.000.000
- <situação>
  - 1 - Crescente
  - 2 - Decrescente
  - 3 - Aleatório
- -P Durante a pesquisa, imprime o registro que está sendo comparado com a chave de pesquisa.

Comando para realizar a busca automática de 10 chaves bem distintas imprimindo as métricas.

Realizando os testes automáticos com 10 chaves bem distintas

```
./analise <método> <quantidade> <situação>
```

2 Desenvolvimento

2.1 Acesso Sequencial Indexado

Primeiro é verificado se a situação crescente foi passado por comando no terminal, já que o método de Acesso Sequencial Indexado que foi implementado comporta apenas a pesquisa em arquivo ordenado crescentemente.

Em seguida, o código forma o nome do arquivo de índices usando a função *sprintf*. O nome do arquivo é composto pelo prefixo *indices_* seguido do valor da variável *quantidade* e a extensão *.bin*.

Se caso esse arquivo de índices já existir então não é necessário passar pelo processo de criação de índices, indo direto para o processo de pesquisa.

Se não existir esse arquivo de índices então é criado uma tabela de índices das páginas do arquivo. As páginas estão configuradas para ter um tamanho em *bytes* múltiplo de 512 *bytes*, neste caso 1024 registros por página (ITENSPAGINA) e a tabela de índices podendo conter no máximo 1024 índices (MAXTABELA).

* Considerando que cada registro tem 6024 *bytes*.

Depois de criar os índices, os mesmos são salvos no arquivo de nome explicado anteriormente.

Após o arquivo de índices ser aberto o código realiza a pesquisa sequencia da página onde o item pode ser encontrado. Ele percorre a tabela de índices comparando a chave do item com as chaves na tabela até encontrar uma chave maior ou igual.

Se a chave desejada for menor que a primeira chave na tabela de índices, isso significa que o item não existe no arquivo e a função retorna false.

Caso contrário o código lê a página desejada do arquivo e faz a pesquisa sequencial passando por cada item da página até encontrar a chave desejada retornando *true* ou chegar ao fim da página retornando *false*.

2.2 Árvore Binária adequada à memória externa

No arquivo **main.c**, o processo começa verificando se já existe um arquivo de árvore montado anteriormente. Caso esse arquivo exista, ele é utilizado diretamente para a busca.

Após conferir os arquivos, se for identificado que não há uma árvore montada com o número de argumentos requisitados, o processo para montar um novo arquivo é iniciado. Primeiramente, lê-se a quantidade de dados requerida e utiliza-se esses dados para montar uma árvore binária na memória interna. Em seguida, a árvore em memória é usada para gerar o arquivo em memória externa, percorrendo a árvore recursivamente.

Por fim, a busca é realizada de forma recursiva.

2.3 Árvore B

No arquivo **main.c**, o processo começa com a inicialização da árvore B por meio da função **inicializaArvoreB**, que define o ponteiro da árvore como **NULL**. Em sequência, os registros são lidos de um arquivo usando a função **fread**. Cada registro lido é inserido na árvore B com a função **insereArvoreB**, incrementando a métrica de leituras.

Após a inserção de todos os registros, um item com uma chave específica é preparado para busca. A função **pesquisaArvoreB** é então chamada para verificar a presença desse item na árvore. Essa função realiza uma busca recursiva, comparando a chave do item buscado com as chaves armazenadas nas páginas da árvore. A métrica de comparações é incrementada a cada comparação feita. Se a chave for encontrada, o item é impresso e uma mensagem de sucesso é exibida.

No arquivo **arvoreB.c**, encontram-se as funções que implementam as operações da árvore B. Durante a inserção de registros, a função **insereArvoreB** é utilizada para encontrar a página apropriada onde o novo registro deve ser inserido. Se a página estiver cheia, a função **insereNaPaginaArvoreB** divide a página e ajusta os ponteiros para manter a árvore balanceada. Se necessário, a função **insArvoreB** é chamada para dividir a página e garantir que a árvore continue balanceada, ajustando a estrutura da árvore conforme os registros são inseridos. Quando a raiz da árvore cresce devido à inserção de um novo registro, uma nova página é criada, aumentando a altura da árvore.

2.4 Árvore B*

O algoritmo implementa a inserção e pesquisa em uma árvore B*, utilizando diversas funções para gerenciar páginas internas e externas. A função **inicializaBEstrela** aloa memória para uma nova página da árvore a inicializa como uma página externa vazia, preparando a estrutura base para a árvore. A função **insereNaPagExt** é responsável por inserir registros em páginas externas, mantendo-os ordenados, enquanto **insereNaPagInt** insere chaves e ponteiros em páginas internas, garantindo a ordenação dos elementos.

Para inserir registros na árvore, a função principal **insereBEstrela** chama recursivamente **insBEstrela**, que lida com a inserção nas páginas internas e externas, dividindo-as quando necessário. Se uma página está cheia, o algoritmo divide a página, redistribui os elementos e ajusta as conexões entre as páginas para manter a árvore balanceada. Caso a árvore cresça em altura devido a uma divisão na raiz, uma nova raiz é criada para acomodar o crescimento.

A pesquisa de registros na árvore é realizada pela função **pesquisaBEstrela**, que percorre a árvore de forma recursiva, navegando entre páginas internas e externas até encontrar o registro ou determinar que ele não está presente. Durante o processo, o algoritmo utiliza a estrutura **Metrica** para contabilizar o número de comparações feitas, permitindo medir a eficiência das operações, assim como proposto no enunciado do trabalho. Desta forma o algoritmo mantém as propriedades da árvore B*, garantindo inserções e pesquisas eficientes mesmo com um grande volume de dados.

3 Análise de Desempenho

No contexto de estruturas de dados como acesso sequencial indexado, árvores binárias, árvores B e árvores B*, a análise de desempenho nos permite identificar a mais adequada para diferentes cenários de uso, assegurando operações mais rápidas e eficientes em sistemas computacionais.

3.1 Acesso Sequencial

		Crescente				
		100	1000	10000	100000	1000000
Pesquisa	leitura	1	1	1	1	1
	comparação	49	454	821,2	503,5	1109,3
	tempo (s)	0,0028582	0,0045235	0,004688	0,0046186	0,0089318
Criação	leitura	100	1000	10000	100000	1000000
	comparação	0	0	0	0	0
	tempo (s)	0,000361	0,002599	0,032146	0,246802	5,688888

3.2 Árvore Binária

		Crescente				
		100	1000	10000	100000	1000000
Pesquisa	leitura	48.8	472.6	-	-	-
	comparação	48.8	472.6	-	-	-
	tempo (s)	0,000538	0,004713	-	-	-
Criação	leitura	199	1999	-	-	-
	comparação	4950	499500	-	-	-
	tempo (s)	0,007267	0,244843	-	-	-

		Decrescente				
		100	1000	10000	100000	1000000
Pesquisa	leitura	48.6	472.6	-	-	-
	comparação	48.6	472.6	-	-	-
	tempo (s)	0,000483	0,004533	-	-	-
Criação	leitura	199	1999	-	-	-
	comparação	4950	499500	-	-	-
	tempo (s)	0,007961	0,226098	-	-	-

		Aleatório				
		100	1000	10000	100000	1000000
Pesquisa	leitura	8	13.5	16.3	18.6	24.3
	comparação	8	13.5	16.3	18.6	24.3
	tempo (s)	0,0000764	0,00011	0,00011	0,000199	0,001203
Criação	leitura	199	1999	19999	199999	1999999
	comparação	628	11446	162450	2077264	25020488
	tempo (s)	0,001251	0,017885	0,171841	1,667932	20,319598

3.3 Árvore B

	Crescente					
		100	1000	10000	100000	1000000
Pesquisa	leitura	0	0	0	0	0
	comparação	11	20	25	30	44
	tempo (s)	0,000005	0,000009	0,000006	0,000006	0,000022
Criação	leitura	100	1000	10000	100000	1000000
	comparação	2300	39350	557243	7224783	88674163
	tempo (s)	0,001452	0,016186	0,148954	1,343969	15,305574

	Decrescente					
		100	1000	10000	100000	1000000
Pesquisa	leitura	0	0	0	0	0
	comparação	12	20	27	35	43
	tempo (s)	0,000005	0,000005	0,000005	0,000006	0,00002
Criação	leitura	100	1000	10000	100000	1000000
	comparação	2076	25912	291645	3212685	35036218
	tempo (s)	0,001549	0,015384	0,145778	1,44872	15,096613

	Aleatório					
		100	1000	10000	100000	1000000
Pesquisa	leitura	0	0	0	0	0
	comparação	17	24	25	35	39
	tempo (s)	0,000004	0,000005	0,000005	0,000006	0,000006
Criação	leitura	100	1000	10000	100000	1000000
	comparação	2138	31688	406542	4887068	57609515
	tempo (s)	0,001438	0,016166	0,1612	1,655	19,5396

3.4 Árvore B*

	Crescente					
		100	1000	10000	100000	1000000
Pesquisa	leitura	0	0	0	0	0
	comparação	10	16	20	26	31
	tempo (s)	0,000002	0,000003	0,000003	0,000004	0,00004
Criação	leitura	100	1000	10000	100000	1000000
	comparação	1648	27126	375716	4806853	58511997
	tempo (s)	0,001238	0,012037	0,111291	1,093143	13,775579

	Decrescente					
		100	1000	10000	100000	1000000
Pesquisa	leitura	0	0	0	0	0
	comparação	12	17	23	29	33
	tempo (s)	0,000002	0,000004	0,000004	0,000004	0,000004
Criação	leitura	100	1000	10000	100000	1000000
	comparação	2954	37583	442735	5070198	57060755
	tempo (s)	0,001423	0,015429	0,132383	1,27309	15,039945

	Aleatório					
		100	1000	10000	100000	1000000
Pesquisa	leitura	0	0	0	0	0
	comparação	11	15	21	26	330
	tempo (s)	0,000002	0,000002	0,000002	0,000003	0,000004
Criação	leitura	100	1000	10000	100000	1000000
	comparação	2148	29119	367425	4425580	515872120
	tempo (s)	0,001264	0,013076	0,127847	1,372573	16,70456

4 Análise Detalhada dos Testes

4.1 Acesso Sequencial Indexado

Observa-se que é possível notar o aumento considerável no tempo de pesquisa do acesso sequencial indexado sendo o mais demorado para realizar as buscas das chaves.

Quanto menor for a página para realizar as buscas sequencialmente na memória principal, mais será necessário ler no disco.

O método pode melhorar de desempenho quando se tem mais memória disponível para ser lida de uma vez da memória externa para a memória principal.

4.2 Árvore Binária

Na implementação da criação de um arquivo externo para os métodos crescente e decrescente entramos dificuldade com falha de segmentação por causa da quantidade de chamadas recursivas que os registros sequenciais geram.

Isso pode ser resolvido implementando um método diferente para criar cada arquivo externo para cada situação.

Na situação de registros aleatórios a árvore binaria em arquivo externo não apresentou falha de segmentação, pelo fato de que com os registros aleatórios a árvore fica mais balanceada fazendo menos chamadas recursivas sem estourar a pilha de execução.

4.3 Árvore B

A criação da árvore B é bem rápida, com tempos de execução baixos e um número constante de leituras e comparações. Ela fica um pouco mais lenta conforme o número de registros aumenta, mas ainda mantém uma boa eficiência para leitura e comparação.

Pesquisar também é rápido, embora o número de comparações possa crescer um pouco com mais registros. Quando se trata de criar uma árvore B, o tempo de execução e o número de leituras e comparações aumentam mais, especialmente com grandes volumes de dados.

Mesmo assim, a pesquisa continua sendo eficiente, com tempos bem baixos, apesar de haver um aumento no número de comparações.

4.4 Árvore B*

À medida que o número de registros aumenta, o tempo para criar a árvore e o número de comparações também crescem, mas a árvore B* ainda consegue fazer buscas de forma bem eficiente.

A estrutura dos nós na árvore B* é mais inteligente, aproveitando melhor o espaço e evitando a necessidade de redistribuições constantes, diferente da árvore B.

O balanceamento é mais eficaz, reduzindo o número de níveis e melhorando a eficiência geral da árvore. A árvore B* traz ótimas vantagens, especialmente quando lidamos com grandes quantidades de dados, oferecendo uma busca eficiente e um uso de espaço bem otimizado.

5 Conclusão

Os integrantes do grupo enfrentaram desafios ao estruturar e adaptar os algoritmos para o problema proposto, variando conforme a função de cada membro.

Para a Árvore Binária em Memória Externa, o principal desafio foi a diferença entre gerar um arquivo a partir de uma árvore em memória interna e gerar o arquivo diretamente. Optou-se por usar a árvore interna como auxílio para montar a árvore externa, após várias tentativas e falhas. Durante os testes, a abordagem recursiva foi mais rápida, mas não eficiente em termos de memória, levando a erros de segmentação devido ao número elevado de camadas necessárias. A abordagem aleatória foi mais eficiente para gerar árvores menos profundas, devido à desordem dos dados.

A conclusão do trabalho destaca o aprofundamento no assunto e o crescimento pessoal e coletivo dos integrantes do grupo.

A Árvore B* apresentou boa escalabilidade na busca, mantendo tempos baixos mesmo com grandes quantidades de dados. No entanto, a construção da árvore se torna mais cara à medida que os dados crescem, tanto em termos de tempo quanto de comparações. Isso é típico para estruturas de dados que precisam manter a eficiência da pesquisa, mas onde a construção inicial pode ser um processo mais caro.