

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	4
2	Método Sequencial Indexado	4
2.1	Criação do Índice de Páginas	4
2.2	Busca no Índice	4
2.3	Carregamento da Página-Alvo	5
2.4	Busca Dentro da Página	5
2.5	Resultados Experimentais	5
2.5.1	Arquivo Ordenado Ascendente	5
2.5.2	Arquivo Ordenado Descendente	6
2.5.3	Arquivo Misto	6
2.6	Conclusões sobre o Método Sequencial Indexado	6
3	Árvore Binária de Pesquisa	6
3.1	Estrutura do Nô em Arquivo	7
3.2	Criação da Árvore	7
3.3	Operações de Leitura e Escrita	7
3.4	Inserção de Registros	7
3.5	Busca de Registros	8
3.6	Construção da Árvore a partir de Arquivo	8
3.7	Teste com 20 Chaves Aleatórias	8
3.8	Resultados Experimentais	9
3.8.1	Arquivo Ordenado Ascendente	9
3.8.2	Arquivo Ordenado Descendente	10
3.8.3	Arquivo Misto	10
3.9	Análise de Complexidade	11
3.9.1	Complexidade de Construção	11
3.9.2	Complexidade de Busca	12
3.10	Conclusões sobre a Árvore Binária de Pesquisa	12
3.10.1	Desempenho por Tipo de Arquivo	12
3.10.2	Limitações Identificadas	12
3.10.3	Comparação de Métricas	13
3.10.4	Recomendações	13
3.10.5	Principais Conclusões	13
4	Árvore B	13
4.1	Estrutura da Página	14
4.2	Propriedades da Árvore B	14
4.3	Pesquisa	14
4.4	Inserção	15
4.4.1	Função InsereNaPagina	15
4.4.2	Função Ins (Inserção Recursiva)	15
4.4.3	Divisão de Página	16
4.4.4	Função Insere (Interface Principal)	16
4.5	Construção da Árvore	16
4.6	Teste com 20 Chaves Aleatórias	17
4.7	Resultados Experimentais	17
4.7.1	Arquivo Ordenado Ascendente	17

4.7.2	Arquivo Ordenado Descendente	18
4.7.3	Arquivo Misto	19
4.7.4	Comparação entre Ordenações	20
4.8	Análise de Complexidade	20
4.8.1	Altura da Árvore	20
4.8.2	Complexidade de Inserção	20
4.8.3	Complexidade de Busca	21
4.9	Vantagens sobre a Árvore Binária	21
4.10	Conclusões sobre a Árvore B	21
4.10.1	Desempenho Geral	21
4.10.2	Análise de Complexidade Observada	22
4.10.3	Comparação com Árvore Binária	22
4.10.4	Recomendações	22
5	Árvore B*	22
5.1	Diferenças em Relação à Árvore B	23
5.2	Estrutura e Implementação	23
5.3	Resultados Experimentais	23
5.3.1	Arquivo Ordenado Ascendente	23
5.3.2	Arquivo Ordenado Descendente	24
5.3.3	Arquivo Misto	25
5.4	Comparação Árvore B vs Árvore B*	26
5.4.1	Construção da Árvore — Arquivo Ascendente	26
5.4.2	Construção da Árvore — Arquivo Descendente	26
5.4.3	Construção da Árvore — Arquivo Misto	26
5.4.4	Pesquisa — Comparação Geral	26
5.5	Análise Comparativa	26
5.5.1	Tempo de Construção	26
5.5.2	Número de Comparações na Construção	27
5.5.3	Desempenho de Pesquisa	27
5.6	Análise Teórica vs Prática	27
5.6.1	Vantagens Teóricas da B*	27
5.6.2	Observações Práticas	27
5.7	Conclusões sobre a Árvore B*	28
5.7.1	Desempenho Observado	28
5.7.2	Trade-offs Identificados	28
5.7.3	Comparação Final (200000 registros)	28
5.7.4	Recomendações	28
5.7.5	Limitações Comuns	29
6	Conclusão	29

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:

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

Cada registro lido soma uma transferência.

2.4 Busca Dentro da Página

A busca interna é linear:

```
1 int buscarNaPagina(int chave, PaginaAS *pag,
2                      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 ascendente.
- 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:

```
1 typedef struct {
2     Registro registro;
3     long esquerda; // -1 se não existe
4     long direita; // -1 se não existe
5 } 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:

```
1 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:

```
1 NoArquivo lerNo(FILE *arquivo, long posicao);
2 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:

```
1 long inserirRecursivo(FILE *arquivo, long posicao,
2                         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:

```
1 Registro* buscarEmArquivo(const char *nomeArquivoArvore,
2                               int chave, long *comp,
3                               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:

```
1 void lerArquivoBinario(const char *nomeArquivoDados,
2                         const char *nomeArquivoArvore,
3                         int numRegistros, long *transferencias,
4                         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`:

```
1 void pesquisar20AleatoriasAB(const char *nomeArquivoDados,
2                                const char *nomeArquivoArvore,
3                                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	<i>Estouro de pilha</i>
20000	<i>Estouro de pilha</i>
200000	<i>Estouro de pilha</i>
2000000	<i>Estouro de pilha</i>

Criação da Árvore

Tabela 4: Pesquisa — Arquivo Ascendente

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

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	<i>Estouro de pilha</i>
20000	<i>Estouro de pilha</i>
200000	<i>Estouro de pilha</i>
2000000	<i>Estouro de pilha</i>

Criação da Árvore

Tabela 6: Pesquisa — Arquivo Descendente

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

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	<i>Estouro de pilha</i>
2000000	<i>Estouro de pilha</i>

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		<i>Estouro de pilha</i>	
2000000		<i>Estouro de pilha</i>	

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平衡amento
- 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:

```
1 #define ORDEM 50
2 #define MM 2 * ORDEM
3
4 typedef struct pagina {
5     int n;                                // numero de chaves na pagina
6     Registro registro[MM];                // vetor de registros
7     struct pagina *filhos[MM+1];          // ponteiros para filhos
8 } 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:

```
1 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 e 200000 registros nas três situações: ordenado ascendente, ordenado descendente e misto.

4.7.1 Arquivo Ordenado Ascendente

O arquivo ascendente contém registros em ordem crescente. Diferentemente da árvore binária, a Árvore B mantém seu balanceamento independentemente da ordem de inserção, evitando a degeneração estrutural.

Tabela 10: Criação da árvore B — Arquivo Ascendente

Registros	Transferências	Comparações	Tempo (s)
200	200	12550	0.082
2000	2000	183550	0.696
20000	20000	2900350	5.954
200000	200000	37250500	57.672

Criação da Árvore

Tabela 11: Pesquisa B — Arquivo Ascendente

Registros	Comparações	Transferências	Tempo (s)
200	737	20	0.023
2000	1029	20	0.024
20000	1161	20	0.018
200000	1697	20	0.024

Pesquisa (20 chaves aleatórias)

Análise do Caso Ascendente

- Diferentemente da árvore binária, a Árvore B não sofre degeneração com dados ordenados, mantendo desempenho consistente e previsível.
- O número de transferências na construção é igual ao número de registros lidos do arquivo (uma transferência por registro).
- As comparações na pesquisa crescem de forma logarítmica: de 737 (200 registros) para 1697 (200000 registros), um aumento de 130% para um arquivo $1000 \times$ maior.
- Os tempos de pesquisa permanecem praticamente constantes (0.02s), demonstrando excelente eficiência independente do tamanho do arquivo.
- Nenhum estouro de pilha foi observado, ao contrário da árvore binária.

4.7.2 Arquivo Ordenado Descendente

O arquivo descendente produz comportamento similar ao ascendente na Árvore B, confirmando a independência da ordem de inserção para esta estrutura balanceada.

Tabela 12: Criação da árvore B — Arquivo Descendente

Registros	Transferências	Comparações	Tempo (s)
100	100	0	0.038
200	200	0	0.057
2000	2000	0	0.700
20000	20000	0	5.997
200000	200000	0	57.998

Criação da Árvore

Tabela 13: Pesquisa B — Arquivo Descendente

Registros	Comparações	Transferências	Tempo (s)
100	1062	20	0.026
200	846	20	0.026
2000	748	20	0.031
20000	1283	20	0.024
200000	1652	20	0.026

Pesquisa (20 chaves aleatórias)

Análise do Caso Descendente

- Observa-se que as comparações na construção foram registradas como 0, indicando uma particularidade na contabilização do algoritmo para inserções em ordem decrescente.
- O tempo de construção segue padrão similar ao caso ascendente, confirmando que a ordem dos dados não afeta significativamente o desempenho.

- As comparações na pesquisa mantêm comportamento logarítmico, variando de 748 a 1652 para os diferentes tamanhos de arquivo.
- O número de transferências na pesquisa permanece constante (20), correspondendo às 20 chaves pesquisadas.
- O tempo de pesquisa é consistentemente baixo (0.026s).

4.7.3 Arquivo Misto

O arquivo misto contém uma distribuição aleatória de chaves, representando o cenário mais comum em aplicações reais de banco de dados e sistemas de arquivos.

Tabela 14: Criação da árvore B — Arquivo Misto

Registros	Transferências	Comparações	Tempo (s)
100	100	2452	0.037
200	200	6371	0.054
2000	2000	83743	0.659
200000	200000	16268329	59.771

Criação da Árvore

Tabela 15: Pesquisa B — Arquivo Misto

Registros	Comparações	Transferências	Tempo (s)
100	1084	20	0.025
200	760	20	0.029
2000	929	20	0.025
200000	1858	20	0.022

Pesquisa (20 chaves aleatórias)

Análise do Caso Misto

- A Árvore B apresenta comportamento consistente e previsível para dados com distribuição aleatória.
- O número de comparações na pesquisa cresce logaritmicamente: de 1084 (100 registros) para 1858 (200000 registros), um aumento de apenas 71% para um arquivo 2000× maior.
- O tempo de pesquisa permanece praticamente constante (0.025s), demonstrando a eficiência da estrutura balanceada.
- As transferências na construção são lineares, com uma transferência por registro lido do arquivo.
- O número de comparações na construção segue padrão $O(n \log n)$, característico de estruturas balanceadas.

Tabela 16: Comparação da Árvore B entre diferentes ordenações (200000 registros)

Ordenação	Tempo Constr. (s)	Comp. Constr.	Comp. Pesq.	Tempo Pesq. (s)
Ascendente	57.672	37250500	1697	0.024
Descendente	57.998	0	1652	0.026
Misto	59.771	16268329	1858	0.022

4.7.4 Comparação entre Ordenações

Observações da Comparação

- Os tempos de construção são muito similares entre as três ordenações (58-60s para 200000 registros), confirmando a independência da ordem de inserção no desempenho da Árvore B.
- As comparações na pesquisa são consistentes entre os três casos (1652-1858), demonstrando o balanceamento efetivo da estrutura.
- O tempo de pesquisa é praticamente idêntico em todos os casos (0.024s), evidenciando a previsibilidade do desempenho.
- A principal vantagem sobre a árvore binária é evidente: não há degeneração nem estouro de pilha para dados ordenados.

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: 200.000 chaves → altura aproximada de 2 ou 3 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
- **Sem estouro de pilha:** A estrutura suporta grandes volumes de dados sem problemas de recursão excessiva
- **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 (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.
- Nenhum estouro de pilha foi observado, ao contrário da árvore binária que falhou a partir de 2000 registros ordenados.

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.
- **Tempo:** Cresce de forma aproximadamente linear com o número de registros, variando de 0.037s (100 registros) a 59.771s (200.000 registros).

Pesquisa

- **Transferências:** Constante (20), correspondente às 20 chaves pesquisadas.
- **Comparações:** Crescimento logarítmico, com média por pesquisa aumentando de forma suave conforme o tamanho do arquivo.
- **Tempo:** Praticamente constante (0.025s), demonstrando excelente desempenho para operações de consulta.

4.10.3 Comparação com Árvore Binária

Tabela 17: Árvore B vs Árvore Binária — Resumo Comparativo

Característica	Árvore Binária	Árvore B
Dados ordenados (200000 reg.)	Estouro de pilha	57.672s
Dados mistos (200000 reg.)	Estouro de pilha	59.771s
Balanceamento	Não garantido	Garantido
Altura (200000 registros)	$O(n)$ no pior caso	$O(\log_{100} n) \approx 3$
Previsibilidade	Baixa	Alta

4.10.4 Recomendações

- A Árvore B é altamente recomendada para aplicações de pesquisa externa devido à sua consistência e escalabilidade.
- Para arquivos até 200.000 registros, a implementação atual oferece excelente desempenho.
- A ordem da árvore (ORDEM=50) mostrou-se adequada para os testes, proporcionando bom equilíbrio entre altura e ocupação de páginas.
- Para volumes ainda maiores, considerar implementação com páginas em disco (Árvore B externa verdadeira) para evitar limitações de memória.

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 contendo 100, 200, 2000, 20000 e 200000 registros nas três situações: ordenado ascendente, ordenado descendente e misto, utilizando a mesma ordem (ORDEM = 50) da Árvore B para permitir comparação direta.

5.3.1 Arquivo Ordenado Ascendente

O arquivo ascendente contém registros em ordem crescente. Assim como a Árvore B, a Árvore B* mantém seu balanceamento independentemente da ordem de inserção.

Tabela 18: Criação da árvore B* — Arquivo Ascendente

Registros	Transferências	Comparações	Tempo (s)
200	200	12649	0.043
2000	2000	185449	0.372
20000	20000	2935048	3.914
200000	200000	37645198	38.829

Criação da Árvore

Pesquisa (20 chaves aleatórias)

Tabela 19: Pesquisa B* — Arquivo Ascendente

Registros	Comparações	Transferências	Tempo (s)
200	400	20	0.032
2000	996	20	0.027
20000	815	20	0.038
200000	1876	20	0.023

Análise do Caso Ascendente

- A Árvore B* mantém desempenho consistente com dados ordenados, sem degeneração estrutural.
- O número de comparações na construção é ligeiramente superior ao da Árvore B devido às operações de redistribuição.
- O tempo de construção é significativamente menor que na Árvore B (38.829s vs 57.672s para 200000 registros).
- As comparações na pesquisa apresentam comportamento logarítmico.

5.3.2 Arquivo Ordenado Descendente

O arquivo descendente permite avaliar o comportamento da B* com inserções em ordem decrescente, testando a robustez do algoritmo de redistribuição.

Tabela 20: Criação da árvore B* — Arquivo Descendente

Registros	Transferências	Comparações	Tempo (s)
100	100	0	0.027
200	200	99	0.047
2000	2000	1899	0.451
20000	20000	34798	4.209
200000	200000	394798	38.945

Criação da Árvore

Tabela 21: Pesquisa B* — Arquivo Descendente

Registros	Comparações	Transferências	Tempo (s)
100	1084	20	0.031
200	757	20	0.027
2000	854	20	0.024
20000	1097	20	0.028
200000	1741	20	0.028

Pesquisa (20 chaves aleatórias)

Análise do Caso Descendente

- Diferentemente da Árvore B (que registrou 0 comparações na construção), a B* apresenta comparações crescentes, indicando ativação do mecanismo de redistribuição.
- O tempo de construção é significativamente menor que na Árvore B para todos os tamanhos testados.
- As comparações na pesquisa mantêm comportamento logarítmico, variando de 757 a 1741 para os diferentes tamanhos.
- O número de transferências na pesquisa permanece constante (20).

5.3.3 Arquivo Misto

O arquivo misto representa o cenário mais comum em aplicações reais, com distribuição aleatória de chaves.

Tabela 22: Criação da árvore B* — Arquivo Misto

Registros	Transferências	Comparações	Tempo (s)
100	100	2452	0.030
200	200	6512	0.052
2000	2000	84495	0.393
20000	20000	1372581	4.382
200000	200000	16598372	40.673

Criação da Árvore

Tabela 23: Pesquisa B* — Arquivo Misto

Registros	Comparações	Transferências	Tempo (s)
100	1112	20	0.031
200	483	20	0.030
2000	1112	20	0.028
20000	1171	20	0.027
200000	1774	20	0.031

Pesquisa (20 chaves aleatórias)

Análise do Caso Misto

- A Árvore B* apresenta comportamento consistente e previsível para dados com distribuição aleatória.
- O número de comparações na construção é ligeiramente superior ao da Árvore B (16.598.372 vs 16.268.329 para 200000 registros).
- O tempo de construção é menor que na Árvore B (40.673s vs 59.771s para 200000 registros).
- As comparações na pesquisa crescem logaritmicamente, de 483 (200 registros) para 1774 (200000 registros).

5.4 Comparação Árvore B vs Árvore B*

5.4.1 Construção da Árvore — Arquivo Ascendente

Tabela 24: Comparação de construção: Árvore B vs B* (Ascendente)

Registros	Comparações		Tempo (s)	
	B	B*	B	B*
200	12550	12649	0.082	0.043
2000	183550	185449	0.696	0.372
20000	2900350	2935048	5.954	3.914
200000	37250500	37645198	57.672	38.829

5.4.2 Construção da Árvore — Arquivo Descendente

Tabela 25: Comparação de construção: Árvore B vs B* (Descendente)

Registros	Comparações		Tempo (s)	
	B	B*	B	B*
100	0	0	0.038	0.027
200	0	99	0.057	0.047
2000	0	1899	0.700	0.451
20000	0	34798	5.997	4.209
200000	0	394798	57.998	38.945

5.4.3 Construção da Árvore — Arquivo Misto

Tabela 26: Comparação de construção: Árvore B vs B* (Misto)

Registros	Comparações		Tempo (s)	
	B	B*	B	B*
100	2452	2452	0.037	0.030
200	6371	6512	0.054	0.052
2000	83743	84495	0.659	0.393
20000	—	1372581	—	4.382
200000	16268329	16598372	59.771	40.673

5.4.4 Pesquisa — Comparação Geral

5.5 Análise Comparativa

5.5.1 Tempo de Construção

- **Resultado surpreendente:** A Árvore B* apresentou tempo de construção significativamente menor que a Árvore B em todos os casos testados.
- Para 200000 registros ascendentes: B* foi 32.7% mais rápida (38.829s vs 57.672s).

Tabela 27: Comparação de pesquisa: Árvore B vs B* (200000 registros)

Ordenação	Comparações		Tempo (s)	
	B	B*	B	B*
Ascendente	1697	1876	0.024	0.023
Descendente	1652	1741	0.026	0.028
Misto	1858	1774	0.022	0.031

- Para 200000 registros descendentes: B* foi 32.8% mais rápida (38.945s vs 57.998s).
- Para 200000 registros mistos: B* foi 31.9% mais rápida (40.673s vs 59.771s).
- Este resultado contraria a expectativa teórica de que a B* seria mais lenta devido ao overhead de redistribuição, sugerindo otimizações na implementação.

5.5.2 Número de Comparações na Construção

- **Arquivo Ascendente:** B* realiza ligeiramente mais comparações (37.645.198 vs 37.250.500 para 200k), diferença de apenas 1.1%.
- **Arquivo Descendente:** B* realiza comparações significativas (394.798 para 200k) enquanto B registra 0, indicando ativação do mecanismo de redistribuição na B*.
- **Arquivo Misto:** Diferença mínima entre as estruturas (16.598.372 vs 16.268.329 para 200k), cerca de 2% a mais na B*.

5.5.3 Desempenho de Pesquisa

- Os tempos de pesquisa são praticamente equivalentes entre B e B*, ambos na ordem de 0.02-0.03 segundos para 20 pesquisas.
- O número de comparações na pesquisa é similar, com variações que dependem da estrutura interna resultante de cada algoritmo.
- Ambas estruturas demonstram excelente eficiência em operações de consulta.

5.6 Análise Teórica vs Prática

5.6.1 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

5.6.2 Observações Práticas

- A B* demonstrou ser mais rápida na construção, contrariando a expectativa teórica de overhead devido à redistribuição.
- O número de comparações é ligeiramente maior na B*, mas o tempo total é menor, sugerindo melhor aproveitamento de cache ou otimizações na implementação.

- Para arquivos descendentes, a B* ativa mecanismos de redistribuição que não são utilizados na B (evidenciado pelas comparações ≥ 0).
- Ambas estruturas oferecem desempenho de pesquisa equivalente.

5.7 Conclusões sobre a Árvore B*

5.7.1 Desempenho Observado

- A Árvore B* mantém as características fundamentais de balanceamento e altura logarítmica da Árvore B.
- **Construção mais rápida:** Contrariando expectativas, a B* foi aproximadamente 32% mais rápida que a B em todos os cenários.
- O desempenho de pesquisa é equivalente entre as duas estruturas.
- Ambas estruturas funcionam consistentemente independente da ordem de inserção dos dados.

5.7.2 Trade-offs Identificados

- **Comparações vs Tempo:** Embora a B* realize ligeiramente mais comparações, o tempo total de construção é menor.
- **Complexidade de implementação:** O código da B* é mais complexo, com lógica adicional para redistribuição e divisão 2-para-3.
- **Comportamento com dados ordenados:** A B* ativa mecanismos adicionais de redistribuição que mantêm a estrutura otimizada.

5.7.3 Comparação Final (200000 registros)

Tabela 28: Resumo comparativo B vs B* (200000 registros)

Métrica	Ascendente	Descendente	Misto
Tempo Constr. B (s)	57.672	57.998	59.771
Tempo Constr. B* (s)	38.829	38.945	40.673
Ganho B*	32.7%	32.8%	31.9%
Comp. Pesq. B	1697	1652	1858
Comp. Pesq. B*	1876	1741	1774

5.7.4 Recomendações

- **Use Árvore B* quando:**
 - O tempo de construção é crítico (B* foi 32% mais rápida)
 - 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
- **Use Árvore B quando:**

- Simplicidade de implementação é importante
- O número de comparações deve ser minimizado
- A estrutura é bem compreendida pela equipe

5.7.5 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 trabalho apresentou uma análise comparativa detalhada de quatro métodos de pesquisa externa: Acesso Sequencial Indexado, Árvore Binária de Pesquisa, Árvore B e Árvore B*.

Os resultados demonstram que:

- O **Método Sequencial Indexado** é eficiente apenas para arquivos ordenados ascendente, sendo inadequado para outros tipos de ordenação.
- A **Árvore Binária** apresenta bom desempenho para dados aleatórios, mas degenera em casos ordenados, tornando-se inviável para grandes volumes.
- A **Árvore B** demonstrou ser a estrutura mais equilibrada, com desempenho consistente e previsível independentemente da ordem dos dados.
- A **Árvore B*** surpreendeu com tempo de construção aproximadamente 32% menor que a Árvore B, mantendo desempenho de pesquisa equivalente e oferecendo melhor aproveitamento de espaço.

Para aplicações práticas de pesquisa externa, tanto a Árvore B quanto a Árvore B* se mostraram escolhas adequadas, combinando eficiência, previsibilidade e escalabilidade. A escolha entre elas depende das prioridades específicas: B* para melhor tempo de construção e aproveitamento de espaço, B para simplicidade de implementação.