Instituto Federal do Norte de Minas Gerais
Campus Montes Claros
Bacharelado em Ciência da Computação
Organização e Sistemas de Arquivos

PRÁTICA 7

ÁRVORE B EM ARQUIVOS

Discentes:
Aline Soares Santana
João Vitor Ribeiro Botelho

Docente:
Prof. Dr. Wagner Ferreira de Barros

Março
2025

Sumário

1 Objetivo

O presente trabalho tem como objetivo explorar e implementar a estrutura de dados conhecida como Árvore B. O estudo foca na implementação dos métodos essenciais de inserção, remoção e busca, demonstrando como esses algoritmos mantêm o balanceamento e a eficiência na manipulação dos dados. Além disso, o trabalho abrange o salvamento da estrutura em arquivo, permitindo a persistência dos dados e facilitando a análise da árvore após a execução do programa. Essa abordagem visa evidenciar a aplicabilidade das Árvores B em sistemas que requerem operações rápidas e consistentes para gerenciamento de grandes volumes de informação.

2 Introdução

As Árvores B são uma estrutura de dados fundamental na organização e manutenção de grandes índices ordenados, sendo amplamente utilizadas em sistemas de gerenciamento de bancos de dados e sistemas de arquivos. O conceito foi introduzido no artigo seminal *Organization and Maintenance of Large Ordered* Indices (BAYER; MCCREIGHT, 1972), publicado na *Acta Informatica*. Esse trabalho pioneiro estabeleceu as bases para o que ficou conhecido como *"The Ubiquitous B-Tree"*, refletindo sua onipresença em aplicações que exigem eficiência na busca, inserção e remoção de dados em grandes volumes.

A principal característica das Árvores B é a sua estrutura balanceada, garantindo complexidade $O(logn)$ para operações essenciais. Diferente de árvores binárias de busca tradicionais, as Árvores B possuem múltiplos filhos por nó e mantêm seus níveis equilibrados, minimizando o número de acessos a disco — um fator crítico para o desempenho em sistemas de armazenamento secundário. (GEEKS,)

Este relatório apresenta a implementação de uma **Árvore B** em C++ (CPPREFERENCE,), que integra operações de inserção, remoção, busca e persistência (em arquivos binário e de texto). A estrutura do projeto está organizada em três arquivos principais:

- `main.cpp` — contém a função principal e a interação com o usuário.
- `BTree.cpp` — contém a implementação dos métodos das classes `BTree` e `BTreeNode`.
- `BTree.h` — declara as classes e a estrutura auxiliar `ArvoreId` para persistência.

O código também utiliza formatação e cores (via códigos ANSI) para exibir a árvore no terminal de forma visualmente atrativa. (FNKY, 2008; BOTELHO, 2024)

3 Estrutura do Código

3.1 `main.cpp`

Neste arquivo:

- São definidos os nomes dos arquivos de persistência (`btree.bin` e `btree.txt`).
- Um objeto da classe `BTree` é instanciado com uma ordem específica (no exemplo, ordem 2).
- Um vetor de inteiros, `elementos`, é utilizado para testar a inserção de chaves na árvore.

- Após cada inserção, a árvore é impressa com formatação, utilizando caixas e bordas.
- Um vetor de elementos (`elementos2`) é utilizado para testar a remoção. Durante a remoção, são impressas mensagens e a árvore é reexibida.
- Um vetor de buscas (`buscas`) permite testar a função de busca da árvore a partir do arquivo binário.

3.2 `BTree.h`

Contém:

- A definição da **estrutura** `ArvoreId`, que armazena informações de cada nó (ID, endereço, chaves, filhos, etc.) para salvar a árvore em arquivo.
- A **classe** `BTreeNode`, que representa um nó da árvore e possui:
  - Membros de dados: `m` (grau mínimo), `leaf` (indicador se é folha), `keys` (vetor de pares chave-endereço), `children` (vetor de ponteiros para filhos) e `parent`.
  - Declarações dos métodos: inserção, remoção, busca, divisão (`split_child`), fusão (`merge`), empréstimo de chaves (`borrow_from_prev/next`), entre outros.
- A **classe** `BTree`, que representa a árvore e possui:
  - Membro de dados: `root` (ponteiro para a raiz) e `m` (grau mínimo).
  - Declarações dos métodos: inserção (`insert`), remoção (`remove`), impressão (`print_tree`), salvamento em arquivo (`saveToFile`), busca no arquivo (`searchInFile`) e conversão do arquivo binário para texto (`convertBinToTxt`).

3.3 `BTree.cpp`

Contém a implementação de todas as funções declaradas em `BTree.h`. A seguir, detalhamos as funções principais.

4 Detalhamento das Funções em BTree.cpp

4.1 Na classe `BTreeNode`

Construtor: `BTreeNode(int m, bool leaf, BTreeNode* parent)` Inicializa um nó definindo o grau mínimo, se é folha e o ponteiro para o pai.

find_key_index(int key): Percorre o vetor `keys` e retorna o índice onde a chave deve ser inserida ou onde ela está presente.

insert_nonfull(int key, const string& address): Insere uma chave em um nó que não está cheio. Se o nó é folha, insere diretamente na posição correta (movendo as chaves para abrir espaço). Se o nó não for folha, encontra o filho apropriado e, se necessário, divide o filho usando `split_child` antes de inserir.

split_child(int i): Divide o filho `children[i]` que está cheio (possui mais de `2*m` chaves). A chave mediana é promovida ao nó pai, e o filho é dividido em dois nós, com os ponteiros de filhos e chaves ajustados.

remove(int key, const string& binFilename): Remove uma chave do nó. Se a chave está presente:

- Se o nó for folha, chama `remove_from_leaf`.
- Se o nó não for folha, chama `remove_from_nonleaf`, que lida com casos mais complexos (substituição pelo predecessor ou sucessor ou fusão).

Se a chave não está presente e o nó é folha, a função retorna sem alterações.

remove_from_leaf(int idx): Remove a chave que se encontra no índice `idx` do vetor `keys` utilizando o método `erase` do vetor.

remove_from_nonleaf(int idx, const string& binFilename): Remove uma chave de um nó interno. Procura o predecessor ou sucessor da chave para substituí-la. Se os filhos do nó interno possuem poucas chaves, realiza a fusão (`merge`) e remove a chave do nó resultante.

get_predecessor(int idx): Percorre o filho à esquerda da chave no índice `idx` até chegar a um nó folha e retorna a última chave encontrada (o predecessor imediato).

get_successor(int idx): Percorre o filho à direita da chave no índice `idx` até chegar a um nó folha e retorna a primeira chave encontrada (o sucessor imediato).

fill(int idx): Garante que o filho no índice `idx` tenha o número mínimo de chaves. Se o filho possui menos chaves que o mínimo, tenta emprestar uma chave do irmão anterior (`borrow_from_prev`) ou do irmão seguinte (`borrow_from_next`). Se não for possível, realiza a fusão (`merge`) com um dos irmãos.

borrow_from_prev(int idx): Emprsta uma chave do filho anterior para o filho no índice `idx`, ajustando os ponteiros e chaves do nó pai.

borrow_from_next(int idx): Similar a `borrow_from_prev`, mas toma uma chave do filho seguinte.

merge(int idx): Fundi (merge) o filho no índice `idx` com o filho seguinte. A chave do nó pai que separa os dois nós é inserida no nó resultante. O nó que foi mesclado é deletado e os vetores de chaves e filhos são atualizados.

rotate_internal_left(): Realiza uma rotação interna à esquerda entre dois nós, movendo uma chave do nó esquerdo para o nó pai e uma chave do nó pai para o nó direito, com ajustes necessários dos ponteiros dos filhos.

fix_deficiency_upwards(): Após a remoção, se um nó fica com menos chaves do que o mínimo, essa função percorre a árvore para realocar chaves nos nós pais e corrigir a deficiência.

print_node(int level): Imprime o nó atual no terminal com a formatação desejada. O método:

- Imprime uma borda esquerda (caractere "|").
- Cria uma indentação baseada no nível (8 espaços por nível).
- Aplica cores diferentes para cada nível (usando códigos ANSI) e exibe o nível e as chaves.
- Calcula dinamicamente o espaço utilizado pelas chaves para alinhar a borda direita.
- Se o nó não for folha, chama recursivamente `print_node(level+1)` para cada filho.

4.2 Na classe `BTree`

Construtor `BTree(int ordem)`: Inicializa a árvore criando uma raiz (um nó folha) e definindo o grau mínimo `m`. O grau da árvore determina o número mínimo e máximo de chaves que cada nó pode ter.

insert(int key, const string& address): Insere uma nova chave na árvore. Se a raiz estiver cheia, ela é dividida antes de inserir. Em seguida, o método `insert_nonfull()` é chamado na raiz ou no filho adequado.

remove(int key): Remove uma chave da árvore. Antes de realizar a remoção, a função pode verificar se a chave existe (através de `searchInFile`). Após a remoção, se a raiz ficar sem chaves, a raiz é ajustada para o primeiro filho ou a árvore se torna vazia.

print_tree(): Chama o método `print_node()` a partir da raiz, imprimindo toda a árvore no terminal com formatação, cores e indentação.

saveToFile(const string& filename): Salva a árvore em um arquivo binário. O método:

- Abre o arquivo em modo binário e grava um cabeçalho contendo o valor de `m`.
- Utiliza uma busca em largura (BFS) para percorrer a árvore.
- Para cada nó, preenche uma estrutura `ArvoreId` com informações (ID, endereços, chaves, filhos) e grava no arquivo.

searchInFile(const string& binFilename, int key): Abre o arquivo binário e procura pela chave desejada. O método lê o cabeçalho e, em seguida, os nós gravados. Se a chave é encontrada, retorna um par com `true` e o endereço do nó; caso contrário, retorna `false`.

convertBinToTxt(const string& binFilename, const string& txtFilename): Converte o arquivo binário que contém a árvore para um arquivo de texto, formatando os dados em colunas (usando `setw()` e `left`). Essa função facilita a visualização da estrutura da árvore em um formato legível.

5 Fluxo de Execução

Os discentes optaram por realizar os testes utilizando valores pré-definidos para agilizar o processo e garantir a consistência dos resultados. Essa abordagem permitiu focar na validação da implementação sem a necessidade de inserir dados manualmente a cada execução. Além disso, o uso de valores previamente estabelecidos facilitou a identificação de possíveis erros e a comparação dos resultados esperados com os obtidos através do site *B-Tree Visualization* (GALLES,), tornando o processo de teste mais eficiente e organizado.

5.1 Inserção

- O vetor `elementos` define os valores a serem inseridos.

```
// Vetor de elementos a serem inseridos
vector<int> elementos = {20, 40, 10, 30, 15, 35, 7, 26, 18, 22, 5, 42, 13, 46, 27, 8, 32, 38, 24, 45, 25};
```

Figura 1 – Trecho de código mencionado

- Para cada valor:
  - Exibe uma mensagem de inserção com espaçamento dinâmico (ajustado pelo `setw()`).
  - Chama `btree.insert()` para inserir o valor na árvore.
  - Imprime a árvore após cada inserção, demonstrando a estrutura com cores e indentação.

```
size_t i = 0;
for (int chave : elementos) {
    i++;
    cout << "| + Inserindo  " << chave << setw(52 - to_string(chave).length()) << "|" << endl;
    cout << "|" << setw(66) << "|" << endl;
    btree.insert(chave, "End_" + to_string(chave));
    btree.print_tree();
    cout << "|" << setw(67) << "|\n";

    if (i < elementos.size()) {
        cout << "├──────────────────────────────────────────────────────────────────┤\n";
    }
}
cout << "└──────────────────────────────────────────────────────────────────┘\n" << endl;

btree.saveToFile(binFilename); //Salva a árvore em arquivo
btree.convertBinToTxt(binFilename, txtFilename); //Converte o arquivo binário para txt
```

Figura 2 – Trecho de código mencionado

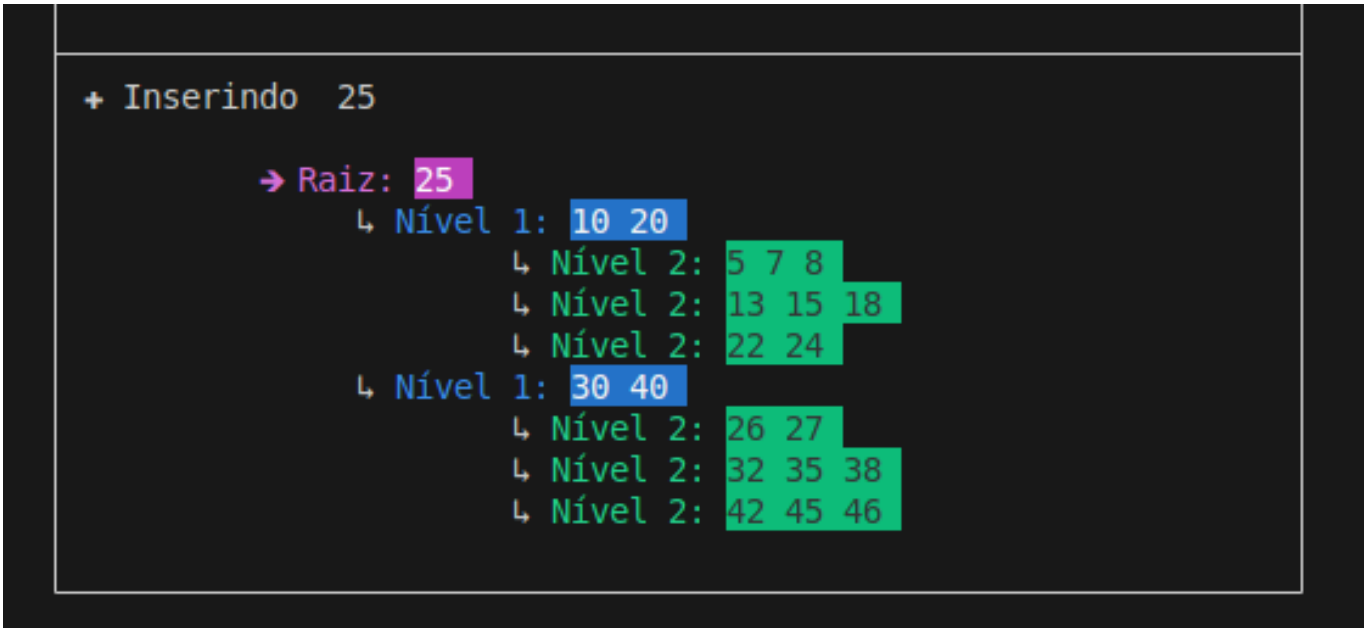

Figura 3 – Exemplo de saída após inserção da chave 25

5.2 Remoção

- O vetor `elementos2` contém os valores a serem removidos.

```
// Vetor de elementos a serem removidos
vector<int> elementos2 = {25, 45, 24};
```

Figura 4 – Trecho de código mencionado

- Para cada valor:
  - Exibe uma mensagem indicando qual chave está sendo removida.
  - Chama `btree.remove()` para remover a chave da árvore.
  - Imprime a árvore após a remoção, evidenciando como a estrutura foi reestruturada.

```
size_t j = 0;
for (int chave : elementos2) {
    j++;
    cout << "| ➤ Removendo  " << chave << "..." << setw(47) << "|" << endl;
    btree.remove(chave);
    cout << "\n|-----------------------------------------------------------------|" << endl;
    cout << "| Árvore pós remoção:" << setw(47) << "|\n";
    cout << "|" << setw(66) << "|" << endl;
    btree.print_tree();
    cout << "|" << setw(67) << "|\n";

    if (j < elementos2.size()) {
        cout << "├─────────────────────────────────────────────────────────────────┤\n";
    }
}
cout << "└─────────────────────────────────────────────────────────────────┘\n" << endl;
```

Figura 5 – Trecho de código mencionado

```
                    REMOÇÃO NA ÁRVORE B
-------------------------------------------------------------
 ➤ Removendo  25...
 - Chave 25 removida!
-------------------------------------------------------------
 Árvore pós remoção:

          ➔ Raiz: 24
               ↳ Nível 1: 10 18
                         ↳ Nível 2: 5 7 8
                         ↳ Nível 2: 13 15
                         ↳ Nível 2: 20 22
               ↳ Nível 1: 30 40
                         ↳ Nível 2: 26 27
                         ↳ Nível 2: 32 35 38
                         ↳ Nível 2: 42 45 46
```

Figura 6 – Exemplo de saída após remoção da chave 25

5.3 Busca

- O vetor `buscas` define as chaves a serem procuradas.
- Para cada chave, o método `searchInFile()` é chamado para procurar a chave no arquivo binário.

```
// Vetor de elementos a serem buscados na árvore através do arquivo binário
vector<int> buscas = {3, 69, 15, 2};
```

Figura 7 – Trecho de código mencionado

O resultado (chave encontrada ou não) é exibido no terminal.

```
// Vetor de elementos a serem buscados na árvore através do arquivo binário
vector<int> buscas = {3, 69, 15, 2};

cout << "\n┌──────────────────────────────────────────────────────────────┐\n";
cout << "|\t\t\tBUSCA NA ÁRVORE B\t\t\t|";
cout << "\n|--------------------------------------------------------------|" << endl;

for (int chave : buscas) {
    auto resultado = btree.searchInFile(binFilename, chave);
    if(resultado.first) {
        cout << "| - Chave " << dec << chave << " - Encontrada no nó com endereço: 0x"
            << hex << resultado.second << dec << "\t|" << endl;
    } else {
        cout << "| - Chave " << chave << " - Não encontrada!\t\t\t\t\t|" << endl;
    }
}
cout << "└──────────────────────────────────────────────────────────────┘\n" << endl;
```

Figura 8 – Trecho de código mencionado

```
                    BUSCA NA ÁRVORE B
----------------------------------------------------------------
 - Chave 3 - Não encontrada!
 - Chave 69 - Não encontrada!
 - Chave 15 - Encontrada no nó com endereço: 0x651ccd583d10
 - Chave 2 - Não encontrada!
```

Figura 9 – Exemplo de busca de chaves

6 Considerações Finais

Esta implementação da Árvore B em C++ demonstra:

- A manipulação dinâmica da estrutura de dados em memória, mantendo as propriedades da Árvore B (balanceamento, número mínimo/máximo de chaves).
- A persistência da árvore em arquivos, permitindo salvar e visualizar a estrutura fora da memória.
- Uma interface visual aprimorada com formatação, cores e caracteres especiais para exibição no terminal.
- Uma modularização eficaz do código, facilitando a manutenção e a expansão futura.

Referências

BAYER, R.; MCCREIGHT, E. M. Organization and maintenance of large ordered indices. *Acta Informatica*, Springer, v. 1, n. 3, p. 173–189, 1972. Este artigo seminal é a fonte original do conceito da Árvore B, frequentemente referido como "The Ubiquitous B-Tree".

BOTELHO, J. V. R. *cores_cmd GitHub Repository*. 2024. ⟨https://github.com/Vitor-Ribe/cores_cmd⟩. Accessed: March 11, 2025.

CPPREFERENCE. *C++ Documentation – DevDocs*. ⟨https://devdocs.io/cpp/⟩. Accessed: March 11, 2025.

FNKY. *ANSI Escape Codes for Terminal Colors and Formatting*. 2008. Disponível em ⟨https://www1.folha.uol.com.br/fsp/cotidian/ff2304200831.htm⟩. Acesso em: 07 set. 2024.

GALLES, D. *B-Tree Visualization*. ⟨https://www.cs.usfca.edu/~galles/visualization/BTree.html⟩. Accessed: March 11, 2025.

GEEKS, G. for. *Introduction of B-Tree*. ⟨https://www.geeksforgeeks.org/introduction-of-b-tree-2/⟩. Accessed: March 11, 2025.