INSTITUTO FEDERAL DO NORTE DE MINAS GERAIS
CAMPUS MONTES CLAROS -MG
CIÊNCIA DA COMPUTAÇÃO

JOÃO KENNEDY SOUZA SOARES
HEULLER RAMOS

RELATÓRIO PRÁTICA DE LABORATÓRIO 7

Relatório proposto pelo Professor Wagner Ferreira de Barros como parte das exigências de avaliação da disciplina de Organização e Sistemas de Arquivos (OSA) do Curso de Bacharelado em Ciência da Computação do Instituto Federal do Norte de Minas Gerais, Campus Montes Claros.

MONTES CLAROS - MG

2025

Relatório Sobre Árvore B com uso de Índices

1. Objetivo

O objetivo deste trabalho foi implementar uma Árvore B, empregando técnicas de manipulação de estruturas dinâmicas e persistência em arquivos. A proposta incluiu o desenvolvimento do algoritmo para inserção, remoção e busca de chaves, bem como a manutenção do balanceamento da árvore por meio de operações como divisão, empréstimo e fusão de nós. Adicionalmente, o sistema implementa a persistência da estrutura em um arquivo binário e a conversão deste para um arquivo de texto, permitindo a visualização e validação da estrutura armazenada.

2. Estrutura do Projeto

2.1. Classes

O projeto foi implementado utilizando as seguintes classes:

NodeDisk: Estrutura responsável por representar um nó da árvore B no arquivo binário. Contém informações como o identificador único, endereços dos nós (na memória e no arquivo), indicação de nó folha, quantidade de chaves armazenadas, os próprios valores das chaves e os identificadores e endereços dos filhos.

BTreeNode: Essa classe que representa os nós da árvore em memória. Cada nó armazena: A ordem mínima (m) da árvore, um vetor de pares (chave, endereço) que contém os dados dos registros, Ponteiros para os filhos e para o nó pai e os métodos para inserção em nós não-cheios, divisão de nós, remoção de chaves (tanto em nós folha quanto em nós internos), além de operações auxiliares como obtenção do predecessor/sucessor, empréstimo entre nós e fusão.

BTree: Essa classe que gerencia a estrutura geral da Árvore B. Responsável por: Manter um ponteiro para a raiz da árvore, fornecer métodos de inserção, remoção e busca de chaves, Gerenciar a persistência da árvore, salvando sua estrutura em um arquivo binário e convertendo-o para um arquivo TXT para visualização, além de implementar a busca de uma chave diretamente no arquivo binário, utilizando a estrutura NodeDisk para recuperar os dados armazenados.

2.2. Arquivos do Projeto

O projeto está organizado nos seguintes arquivos:

BTree.h: Arquivo de cabeçalho que declara as classes, estruturas e funções utilizadas no projeto.

BTree.cpp: Arquivo de implementação que contém a definição dos métodos das classes **BTreeNode** e **BTree**, incluindo as operações de inserção, remoção, divisão, balanceamento e acoplamentos de nós.

main.cpp: Arquivo contendo as função principais, onde são executadas as operações de inserção, remoção e busca na árvore. Este arquivo também cuida da persistência dos dados, salvando a árvore em um arquivo binário e convertendo-a para um arquivo de texto para visualização.

2.3. Arquivos

O projeto utiliza os seguintes arquivos para seu funcionamento:

Arquivo Binário (arvore_b.bin**):** Este arquivo armazena a estrutura completa da Árvore B de forma principal, como: endereços, chaves, endereços dos pais e dos filhos, flag folha só que em formato binário que foi utilizado pelas funções para inserção, busca e remoção de elementos.

Arquivo de Texto (arvore_b.txt**):** Este arquivo armazena apenas a visualização do arquivo binário de forma estruturada com da Árvore B, o que inclui o cabeçalho com o valor da ordem e os registros dos nós representados pela chave e seu endereço, além de manter informações como endereços do nó Pai, endereço dos nós filhos, quantidade de elementos no nó, se é no folha ou não com o bit (0 ou 1) e nos casos que não tem filhos guarda -1 e se nao tiver nó pai no caso da raiz também armazena -1 e assim garantindo relações hierárquicas, facilitando a visualização de cada passo da inserção, remoção e busca.

2.4. Funcionalidades

O sistema implementa as seguintes funcionalidades:

Inserção: Ao inserir uma nova chave, o sistema verifica se o nó destino possui espaço. Caso contrário, o nó é dividido e a chave mediana é promovida ao nó pai.

Essa operação é realizada de forma recursiva até que todas as restrições da árvore sejam satisfeitas. Quando um nó atinge a quantidade máxima de chaves (definida como **2 * ordem**), ocorre a divisão do nó, promovendo a chave mediana para o nó pai e garantindo que os nós da árvore continuem dentro das restrições definidas.

Remoção: A remoção de uma chave envolve a identificação do nó onde ela se encontra. Se o nó for folha, a remoção é direta. Em nós internos, a remoção pode envolver a substituição pela chave predecessora ou sucessora, seguida de ajustes, como empréstimo (borrow) ou fusão (merge), para manter o balanceamento e a integridade da árvore. As operações de remoção devem obedecer à quantidade mínima de chaves (definida pela ordem), com tratamento diferenciado para nós folhas e internos. Após cada remoção, o sistema atualiza os arquivos binário e de texto, refletindo a nova estrutura da árvore e garantindo o balanceamento adequado..

Busca: A função de busca opera de forma hierárquica na árvore em memória, o que permite localizar rapidamente a chave desejada ao percorrer os nós de forma rápida. Além disso, para garantir maior flexibilidade, a busca também pode ser realizada diretamente em arquivo, (utilizando a função **searchInFile)**. Esta função permite acessar o arquivo binário de forma estruturada, retornando o endereço do nó correspondente à chave procurada, mesmo quando a árvore está parcialmente ou completamente fora da memória.

Visualização: Para a construção da forma de visualização foi implementada por meio de funções que permitem salvar a árvore em um arquivo binário, registrando de forma detalhada informações cruciais de cada nó, como identificadores, endereços, chaves e dados dos nós filhos. Esse formato binário assegura a integridade e a eficiência no armazenamento da árvore, mas não é bom para análise. Visto isso, para facilitar a análise e verificação da estrutura da árvore, é oferecida uma função que converte esse arquivo binário em um formato de texto legível, o que permite uma visualização clara das relações entre os nós, proporcionando uma maneira prática de auditar e validar a estrutura da árvore de forma compreensível.

3. Pesquisa e Gerenciamento

A pesquisa por uma chave é realizada inicialmente em memória, percorrendo a estrutura hierárquica da Árvore B. A função **searchInFile** permite validar a existência

de uma chave diretamente no arquivo binário, retornando o endereço do nó se a chave for encontrada. Essa abordagem híbrida assegura a integridade da árvore e a consistência entre os dados armazenados em memória e em disco.

4. Metodologia

A metodologia implementada neste trabalho baseia-se em uma abordagem incremental onde inicialmente, foram definidas estruturas de dados específicas para representar os nós da Árvore B em memória e em disco, possibilitando a serialização e persistência da estrutura. Em seguida, foi desenvolvido um conjunto de métodos que gerenciam as operações essenciais – inserção, remoção e busca – incorporando algoritmos de balanceamento como divisão, empréstimo e fusão de chaves para manter a integridade da árvore. Complementarmente, foi implementada a escrita da árvore em um arquivo binário e sua conversão para um formato de texto legível, o que facilita a análise e depuração da estrutura, onde foi validada por testes variados com conjuntos pré-definidos de chaves.

5. Testes Realizados

5.1. Escolhendo a Ordem

Os testes foram realizados utilizando algumas ordens onde a estrutura de grau mínimo e máximo para cada teste realizado é alterada de acordo com a ordem.

```
// Define a ordem da Árvore B (exemplo: ordem = 1, 2 ou 3)
// usando a formula 2 * ordem = grau máximo da árvore
// usando a formula ordem = grau mínimo da árvore
/*-------------------------------------------------------------------------------------------*/
// BTree btree(1); /* Equivale a grau máximo 3 no programa de simulação "B-Trees"*/
BTree btree(2); /* Equivale a grau máximo 5 no programa de simulação "B-Trees"*/
// BTree btree(3); /* Equivale a grau máximo 7 no programa de simulação "B-Trees"*/
/*-------------------------------------------------------------------------------------------*/
```

5.2. Testes de Inserção

Foram inseridos diversos elementos na Árvore B utilizando o vetor de chaves definido na função main. A cada inserção, a árvore foi impressa no terminal, demonstrando a correta divisão dos nós quando o número de chaves ultrapassava o limite (2 * ordem) e a promoção da chave mediana para o nó pai. Esse processo validou que a estrutura se mantinha balanceada e que todas as operações de divisão eram executadas conforme esperado. Visto isso, os testes foram realizados com entradas de dados maiores mas foram diminuídos para facilitar a visualização neste relatório como mostrados abaixo:

5.2.1. Árvore de ordem 2.

Foi realizado a inserção dos seguintes elementos:

```
vector<int> elementos = {15, 3, 8, 23, 1, 9, 14, 18, 10, 20, 5, 6, 12, 30, 211, 13, 16, 19, 21, 26, 28, 27, 2};
```

```
Inserindo 15...
[15]
----------------------------------------
Inserindo 3...
[3, 15]
----------------------------------------
Inserindo 8...
[3, 8, 15]
----------------------------------------
Inserindo 23...
[3, 8, 15, 23]
----------------------------------------
```

Após as 4 primeiras inserções, de acordo com a definição, a próxima deverá dividir a árvore e balanceá-la.

```
Inserindo 1...
[8]
    [1, 3]
    [15, 23]
----------------------------------------
Inserindo 9...
[8]
    [1, 3]
    [9, 15, 23]
----------------------------------------
Inserindo 14...
[8]
    [1, 3]
    [9, 14, 15, 23]
----------------------------------------
Inserindo 18...
[8, 15]
    [1, 3]
    [9, 14]
    [18, 23]
----------------------------------------
Inserindo 10...
[8, 15]
    [1, 3]
    [9, 10, 14]
    [18, 23]
```

```
Inserindo 20...
[8, 15]
    [1, 3]
    [9, 10, 14]
    [18, 20, 23]
----------------------------------------
Inserindo 5...
[8, 15]
    [1, 3, 5]
    [9, 10, 14]
    [18, 20, 23]
----------------------------------------
Inserindo 6...
[8, 15]
    [1, 3, 5, 6]
    [9, 10, 14]
    [18, 20, 23]
----------------------------------------
Inserindo 12...
[8, 15]
    [1, 3, 5, 6]
    [9, 10, 12, 14]
    [18, 20, 23]
```

```
Inserindo 30...
[8, 15]
    [1, 3, 5, 6]
    [9, 10, 12, 14]
    [18, 20, 23, 30]
----------------------------------------
Inserindo 211...
[8, 15, 23]
    [1, 3, 5, 6]
    [9, 10, 12, 14]
    [18, 20]
    [30, 211]
----------------------------------------
Inserindo 13...
[8, 12, 15, 23]
    [1, 3, 5, 6]
    [9, 10]
    [13, 14]
    [18, 20]
    [30, 211]
----------------------------------------
Inserindo 16...
[8, 12, 15, 23]
    [1, 3, 5, 6]
    [9, 10]
    [13, 14]
    [16, 18, 20]
    [30, 211]
```

```
Inserindo 19...
[8, 12, 15, 23]
    [1, 3, 5, 6]
    [9, 10]
    [13, 14]
    [16, 18, 19, 20]
    [30, 211]
----------------------------------------
Inserindo 21...
[15]
    [8, 12]
        [1, 3, 5, 6]
        [9, 10]
        [13, 14]
    [19, 23]
        [16, 18]
        [20, 21]
        [30, 211]
----------------------------------------
Inserindo 26...
[15]
    [8, 12]
        [1, 3, 5, 6]
        [9, 10]
        [13, 14]
    [19, 23]
        [16, 18]
        [20, 21]
        [26, 30, 211]
----------------------------------------
```

```
Inserindo 2...
[15]
    [3, 8, 12]
        [1, 2]
        [5, 6]
        [9, 10]
        [13, 14]
    [19, 23, 28]
        [16, 18]
        [20, 21]
        [26, 27]
        [30, 211]
----------------------------------------
```

```
Inserindo 28...
[15]
    [8, 12]
        [1, 3, 5, 6]
        [9, 10]
        [13, 14]
    [19, 23]
        [16, 18]
        [20, 21]
        [26, 28, 30, 211]
----------------------------------------
Inserindo 27...
[15]
    [8, 12]
        [1, 3, 5, 6]
        [9, 10]
        [13, 14]
    [19, 23, 28]
        [16, 18]
        [20, 21]
        [26, 27]
        [30, 211]
----------------------------------------
```

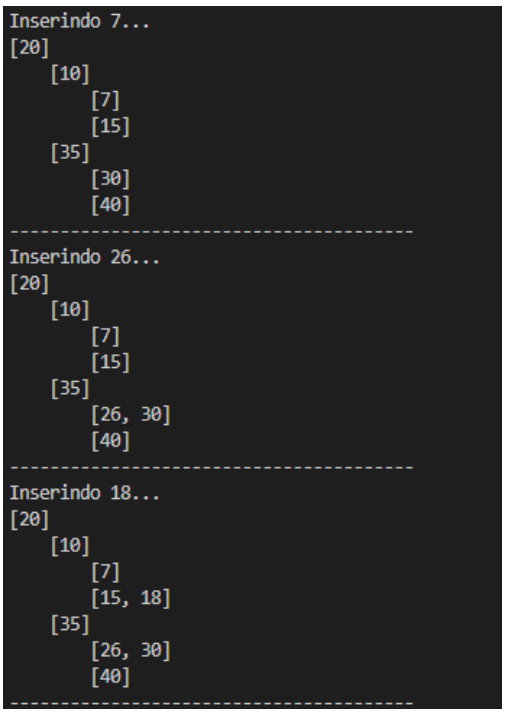
Analisando as inserções, sempre que um nó ultrapassa a quantidade máximas de elementos, definida em 2*ordem, a árvore divide aquele nó e se balanceia.

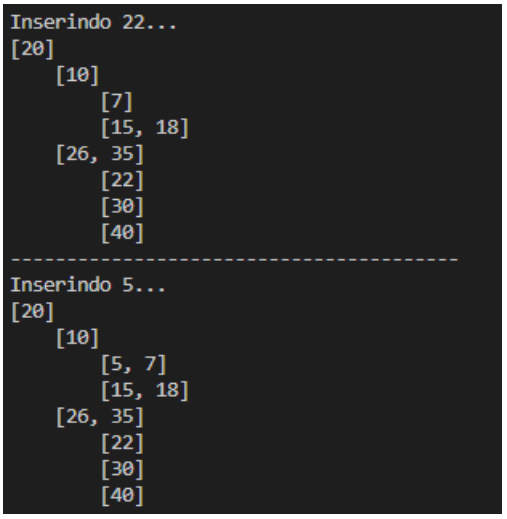
5.2.2. Árvore de ordem 1

Os valores a serem inseridos serão:

```
vector<int> elementos = {20, 40, 10, 30, 15, 35, 7, 26, 18, 22, 5};
```

```
Inserindo 20...
[20]
----------------------------------------
Inserindo 40...
[20, 40]
----------------------------------------
Inserindo 10...
[20]
    [10]
    [40]
----------------------------------------
Inserindo 30...
[20]
    [10]
    [30, 40]
----------------------------------------
Inserindo 15...
[20]
    [10, 15]
    [30, 40]
----------------------------------------
Inserindo 35...
[20, 35]
    [10, 15]
    [30]
    [40]
----------------------------------------
```





De forma semelhante, vemos que para uma árvore de ordem 1, sempre que um nó ultrapassa a capacidade de 2*ordem a árvore é balanceada.

5.3. Testes de Remoção

Após as inserções, foi realizado um conjunto de operações de remoção de chaves específicas. Cada remoção envolveu a verificação se a chave estava presente na árvore, seguida pela aplicação dos métodos de remoção específicos para nós folhas ou internos. Durante esses testes, a árvore foi reimpressa no terminal após cada remoção para confirmar que as operações de empréstimo (borrow) e fusão (merge) eram acionadas corretamente, mantendo a estrutura balanceada, como mostrado abaixo:

5.3.1. Árvore de ordem 2

Os valores a serem removidos foram:

```
vector<int> elementos2 = {15, 3, 8, 23, 1, 9, 14, 18, 10, 20, 5, 6};
```

```
Removendo chave 15...
[14]
    [3, 8]
        [1, 2]
        [5, 6]
        [9, 10, 12, 13]
    [19, 23, 28]
        [16, 18]
        [20, 21]
        [26, 27]
        [30, 211]
---------------------------------------
Removendo chave 3...
[19]
    [8, 14]
        [1, 2, 5, 6]
        [9, 10, 12, 13]
        [16, 18]
    [23, 28]
        [20, 21]
        [26, 27]
        [30, 211]
```

```
Removendo chave 8...
[19]
    [6, 14]
        [1, 2, 5]
        [9, 10, 12, 13]
        [16, 18]
    [23, 28]
        [20, 21]
        [26, 27]
        [30, 211]
---------------------------------------
Removendo chave 23...
[6, 14, 19, 28]
    [1, 2, 5]
    [9, 10, 12, 13]
    [16, 18]
    [20, 21, 26, 27]
    [30, 211]
---------------------------------------
Removendo chave 1...
[6, 14, 19, 28]
    [2, 5]
    [9, 10, 12, 13]
    [16, 18]
    [20, 21, 26, 27]
    [30, 211]
```

```
Removendo chave 9...
[6, 14, 19, 28]
    [2, 5]
    [10, 12, 13]
    [16, 18]
    [20, 21, 26, 27]
    [30, 211]
---------------------------------------
Removendo chave 14...
[6, 13, 19, 28]
    [2, 5]
    [10, 12]
    [16, 18]
    [20, 21, 26, 27]
    [30, 211]
---------------------------------------
Removendo chave 18...
[6, 13, 20, 28]
    [2, 5]
    [10, 12]
    [16, 19]
    [21, 26, 27]
    [30, 211]
---------------------------------------
```

```
Removendo chave 12...
[19, 28]
    [2, 13, 16]
    [21, 26, 27]
    [30, 211]
---------------------------------------
Removendo chave 30...
[19, 27]
    [2, 13, 16]
    [21, 26]
    [28, 211]
---------------------------------------
Removendo chave 2...
[19, 27]
    [13, 16]
    [21, 26]
    [28, 211]
---------------------------------------
```

```
Removendo chave 10...
[13, 20, 28]
    [2, 5, 6, 12]
    [16, 19]
    [21, 26, 27]
    [30, 211]
---------------------------------------
Removendo chave 20...
[12, 19, 28]
    [2, 5, 6]
    [13, 16]
    [21, 26, 27]
    [30, 211]
---------------------------------------
Removendo chave 5...
[12, 19, 28]
    [2, 6]
    [13, 16]
    [21, 26, 27]
    [30, 211]
---------------------------------------
Removendo chave 6...
[19, 28]
    [2, 12, 13, 16]
    [21, 26, 27]
    [30, 211]
---------------------------------------
```

Pode-se perceber que ao remover um elemento de um nó, caso o nó fique com menos de 2 elementos ou seja, o valor da ordem a árvore é balanceada.

5.3.2. Árvore de ordem 1

Elementos a serem removidos:

```
vector<int> elementos2 = {20, 15,  7, 18, 22, 5, 6, 12, 30};
```

```
Removendo chave 20...
[18]
    [10]
        [5, 7]
        [15]
    [26, 35]
        [22]
        [30]
        [40]
---------------------------------------
Removendo chave 15...
[18]
    [7]
        [5]
        [10]
    [26, 35]
        [22]
        [30]
        [40]
```

```
Removendo chave 7...
[26]
    [18]
        [5, 10]
        [22]
    [35]
        [30]
        [40]
---------------------------------------
Removendo chave 18...
[26]
    [10]
        [5]
        [22]
    [35]
        [30]
        [40]
---------------------------------------
Removendo chave 22...
[26, 35]
    [5, 10]
    [30]
    [40]
---------------------------------------
```

```
Removendo chave 5...
[26, 35]
    [10]
    [30]
    [40]
---------------------------------------
Removendo chave 6...
Chave 6 n├úo encontrada na ├írvore.
[26, 35]
    [10]
    [30]
    [40]
---------------------------------------
Removendo chave 12...
Chave 12 n├úo encontrada na ├írvore.
[26, 35]
    [10]
    [30]
    [40]
---------------------------------------
Removendo chave 30...
[35]
    [10, 26]
    [40]
---------------------------------------
```

5.4. Testes de Busca

Testes de busca foram efetuados utilizando um conjunto de chaves previamente definido. Para cada chave, o sistema realizou a busca tanto na estrutura em memória quanto diretamente no arquivo binário através da função searchInFile. Os resultados, que incluem a confirmação da existência da chave e o endereço do nó correspondente, foram exibidos no terminal. Esses testes garantiram que os registros buscados correspondiam aos dados corretamente indexados, como mostrado abaixo:

5.4.1. Árvore de ordem 2

```
Chave 19 encontrada no no com endereco: 0x1f658602760
-----------------------------------------------------
Chave 3 nao encontrada na arvore.
-----------------------------------------------------
Chave 8 nao encontrada na arvore.
-----------------------------------------------------
Chave 23 nao encontrada na arvore.
-----------------------------------------------------
Chave 26 encontrada no no com endereco: 0x1f658606a00
-----------------------------------------------------
Chave 9 nao encontrada na arvore.
-----------------------------------------------------
Chave 16 encontrada no no com endereco: 0x1f6586023f0
-----------------------------------------------------
Chave 18 nao encontrada na arvore.
-----------------------------------------------------
Chave 27 encontrada no no com endereco: 0x1f658602760
-----------------------------------------------------
Chave 20 nao encontrada na arvore.
-----------------------------------------------------
Chave 211 encontrada no no com endereco: 0x1f658606c40
-----------------------------------------------------
Chave 6 nao encontrada na arvore.
-----------------------------------------------------
Chave 12 nao encontrada na arvore.
-----------------------------------------------------
Chave 30 nao encontrada na arvore.
-----------------------------------------------------
Chave 2 nao encontrada na arvore.
-----------------------------------------------------
```

5.4.2 Árvore de ordem 1

Elementos a serem buscados:

```
vector<int> chavesBusca = {20, 26, 4, 35, 18, 10,22, 40};
```

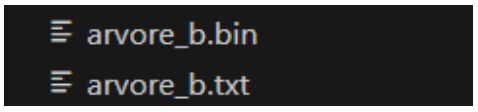
```
Chave 20 nao encontrada na arvore.
-----------------------------------------------------
Chave 26 encontrada no no com endereco: 0x1aad8e623f0
-----------------------------------------------------
Chave 4 nao encontrada na arvore.
-----------------------------------------------------
Chave 35 encontrada no no com endereco: 0x1aad8e666e0
-----------------------------------------------------
Chave 18 nao encontrada na arvore.
-----------------------------------------------------
Chave 10 encontrada no no com endereco: 0x1aad8e623f0
-----------------------------------------------------
Chave 22 nao encontrada na arvore.
-----------------------------------------------------
Chave 40 encontrada no no com endereco: 0x1aad8e668d0
```

5.5. Testes de Consistência e Conversão

Por fim, foram realizados testes com os dados e após as operações de inserção e remoção, a árvore foi salva em um arquivo binário (**arvore_b.bin**) e posteriormente convertida para um arquivo de texto (**arvore_b.txt**). Essa conversão permitiu a visualização detalhada dos nós da Árvore B, incluindo identificadores, endereços, chaves e informações dos filhos, flag folha. A inspeção do arquivo de texto confirmou que todas as informações estavam consistentes com a estrutura.

5.5.1 Arquivos

As imagens abaixo foram retiradas de um arquivo .txt gerado a partir do arquivo .bin para uma melhor visualização.

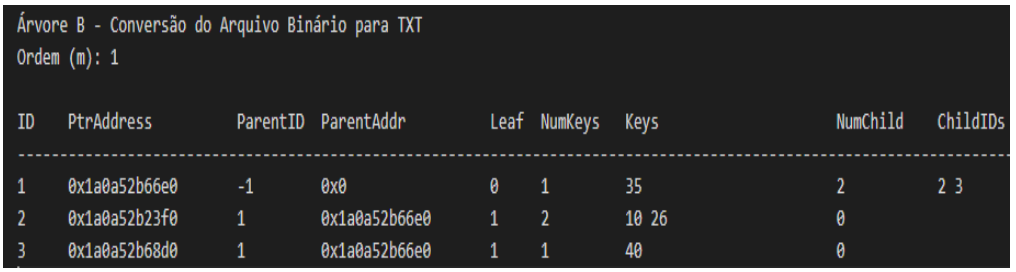


5.5.2 Arquivo da árvore de ordem 2

```
Árvore B - Conversão do Arquivo Binário para TXT
Ordem (m): 2

ID    PtrAddress          ParentID  ParentAddr          Leaf  NumKeys   Keys                NumChild    ChildIDs
-------------------------------------------------------------------------------------------------------------
1     0x1a64f5a2760       -1        0x0                 0     2         19 28               3           2 3 4
2     0x1a64f5a23f0       1         0x1a64f5a2760       1     4         2 12 13 16          0
3     0x1a64f5a6a00       1         0x1a64f5a2760       1     3         21 26 27            0
4     0x1a64f5a6c40       1         0x1a64f5a2760       1     2         30 211              0
```

5.5.3 Arquivo da árvore de ordem 1

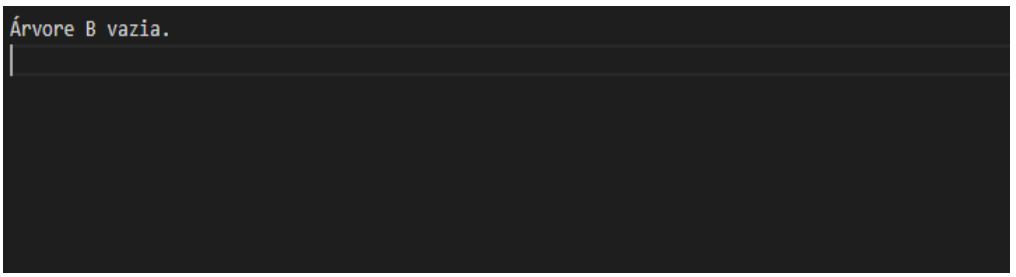


```
Árvore B - Conversão do Arquivo Binário para TXT
Ordem (m): 1

ID    PtrAddress         ParentID  ParentAddr         Leaf  NumKeys   Keys                NumChild    ChildIDs
-------------------------------------------------------------------------------------------------------------
1     0x1a0a52b66e0      -1        0x0                0     1         35                  2           2 3
2     0x1a0a52b23f0      1         0x1a0a52b66e0      1     2         10 26               0
3     0x1a0a52b68d0      1         0x1a0a52b66e0      1     1         40                  0
```

5.5.4 Árvore vazia

Caso a árvore esteja vazia após todas remoções de chaves na árvore, o arquivo .bin e guarda a seguinte mensagem em binário e o de .txt retorna (Árvore B Vazia) fazendo assim, ao buscar elementos na árvore retornam “valor não encontrado” uma vez que estes não existem mais na árvore, como mostradas abaixo:



```
Árvore B vazia.
```

6. Resultado

O projeto atendeu a todas as exigências propostas. A implementação da Árvore B demonstrou eficiência nas operações de inserção, remoção e busca, tanto em memória quanto em disco. Os testes realizados confirmaram que a árvore se mantém balanceada e que as funções de persistência e visualização (em arquivos binário e texto) funcionam conforme o esperado.

7. Conclusão

A implementação do projeto de Árvore B foi bem-sucedida, proporcionando uma solução robusta para o gerenciamento de grandes volumes de dados. As operações de inserção, remoção e busca foram executadas com eficiência, e a persistência dos dados em arquivo garantiu a integridade da estrutura mesmo após modificações. Os desafios enfrentados, como o balanceamento dinâmico da árvore e a sincronização entre a memória e o disco, foram superados com técnicas de manipulação de memória e arquivos.