

Trabalho de Pesquisa: Simulação de Memória em Árvore B

Caio Macedo Lima da Cruz¹, José Carlos Seben de Souza Leite¹,
Igor Carvalho Marchi¹, Jefferson Korte Júnior¹

¹Universidade Tecnológica Federal do Paraná (UTFPR) - Campus Santa Helena

{caiomacedo, joseleite, Jefferson.2024, igormarchi}@alunos.utfpr.edu

Abstract. This paper aims to simulate secondary memory (disk) using the B-Tree data structure, due to its similarity to the block-based memory model. The implementation was developed in the C programming language, defining the data structure and its operations, as well as input and output procedures. The results were satisfactory, yielding a precise and optimized simulation with no errors.

Resumo. Este artigo tem como objetivo realizar a simulação de uma memória secundária (disco), utilizando a estrutura de dados árvore B, dada a sua semelhança com o modelo por blocos da memória. A implementação foi realizada em C, onde foram definidas as operações e a estrutura de dados, além das operações de entrada e saída. Os resultados foram satisfatórios, com uma simulação precisa e otimizada, sem apresentar nenhum erro.

1. Introdução

Um dos grandes desafios da computação é a gestão de memória que é base para sistemas de gerenciamento de banco de dados (SGBD) e sistemas de arquivos com grande infraestrutura. Nesse contexto, é necessário que as operações que envolvam a gestão dos arquivos (inserção, remoção e recuperação) seja extremamente eficiente para que informações não se percam e sejam manipuladas com o mínimo de latência possível. Uma forma de garantir essa eficiência é utilizando a estrutura de dados Árvore B, que servirá de alicerce fundamental para organização de informações em armazenamento secundário.

A Árvore B foi introduzida em 1970 por Rudolf Bayer e Edward M. McCreight, pesquisadores que atuavam nos Laboratórios de Pesquisa Científica da Boeing (Boeing Scientific Research Laboratories). O cenário tecnológico da época era dominado por dispositivos de armazenamento de acesso sequencial ou pseudo-aleatório, como fitas magnéticas e discos rígidos com cabeças móveis pesadas e tempos de busca (seek time) elevados. As estruturas de dados prevalentes para indexação em memória, nomeadamente as árvores binárias de busca (BSTs) e suas variantes balanceadas (como árvores AVL), mostravam-se inadequadas para esse ambiente.

2. Árvore B

Árvores B, uma generalização das árvores AVL, que são para pesquisa binária, é um estrutura de dados de árvore auto-balanceada responsável por armazenar os dados organizados e permitir a busca, acessos sequenciados, inserções e remoções em tempo logarítmico.

Nesse sentido, a Árvore B melhora o conceito de AVL: cada nó pode possuir dois ou mais filhos; Está sempre balanceada; E o tamanho do nó é igual ao tamanho do bloco em disco, o que facilita sua utilização para memória secundária.

Em uma Árvore B de ordem m, cada nó possui no máximo m filhos, deve possuir pelo menos $m/2$ filhos (exceto a raiz), a raiz possui pelo menos dois filhos (caso não seja folha), todas as folhas devem estar no mesmo nível e o número de chaves em um nó não raiz é no máximo $m-1$ e no mínimo $(m/2)-1$ chaves.

2.1. Terminologias

Um nó é uma unidade básica de armazenamento em árvores. Em árvores B, ela representa com bloco com múltiplas chaves e ponteiros, variando com a sua ordem m. Existem 3 tipos diferentes de nós:

- Nό raiz: É o nó superior da árvore e o único nó que não possui um "pai". É o ponto de entrada para qualquer operação de busca.
- Nό interno: É qualquer nó que possui filhos (ou seja, não é uma folha) e não é a raiz. Sua principal função é servir como um índice ou roteador, direcionando a busca para o intervalo correto de valores nos nós inferiores. Em uma Árvore B padrão, nós internos também armazenaam dados associados às chaves (diferente da Árvore B+, onde servem apenas como índices).
- Nό folha: São os nós na camada inferior da árvore que não possuem filhos. Em uma Árvore B, todas as folhas estão obrigatoriamente no mesmo nível (profundidade). Isso garante que a árvore esteja sempre perfeitamente balanceada. É onde a "descida" da árvore termina. Se uma chave não for encontrada ao chegar na folha, e não estiver nos nós internos do caminho, ela não existe na árvore.

2.2. Inserção

As principais operações em árvores B são inserção e remoção. Elas permitem que novos dados sejam adicionados ou retirados, adaptando a árvore para tal. A inserção deve sempre ocorrer em um nó folha da seguinte forma: Buscar a folha na qual a chave deve ser inserida; Se a folha não estiver completa, inserir chave na ordem correta; Se não, realizar cisão do bloco.

| | | | |
|----|----|----|--|
| 30 | 40 | 50 | |
|----|----|----|--|

Figura 1. Árvore B antes da inserção de 60

| | | | |
|----|----|----|----|
| 30 | 40 | 50 | 60 |
|----|----|----|----|

Figura 2. Árvore B depois da inserção de 60

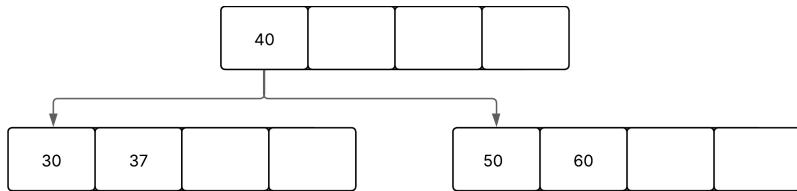


Figura 3. Árvore B depois da inserção de 37 (cisão/split)

3. Simulação de Disco

O modelo de acesso típico empregado em memórias secundária é por blocos, semelhante a estrutura de dados Árvore B. Com isso, busca-se realizar uma simulação de disco, utilizando de Árvores B para gerenciamento. O uso de árvores B para gerenciar a memória pode também reduzir drasticamente a altura da árvore, e, consequentemente, o número de acessos necessários.

3.1. Tecnologias Utilizadas

Para realizar a implementação da simulação de disco, foi utilizada a linguagem de programação C, com a estrutura de 3 arquivos padrão (*header*, *main* e código-fonte). Ela foi escolhida devido a facilidade e eficiência para tratar de manipulação de estrutura de dados e memória secundaria, por ser uma linguagem de baixo nível. As bibliotecas utilizadas foram *stdio.h*, para entrada e saída de dados, *string.h*, para manipular strings, e *stdlib.h*, para controle de processos e alocação de memória.

3.2. Implementação

A implementação das funções e estruturas de dados foi dividida da seguinte forma: Manutenção da Árvore, onde é definida a árvore e suas operações, Manutenção de disco, onde os acessos ao disco são definidos, e Entrada e Saída de dados, onde o usuário consegue utilizar comandos como **INSERT** e **SEARCH** que são armazenados em um arquivo de texto para salvar e dar entrada nas operações.

```

1 #ifndef FUNCTION_H
2 #define FUNCTION_H
3
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <string.h>
7
8 #define MAX_ORDER 20
9 #define MAX_PAGES 1000
10
11 typedef struct {
12     int page_id;           // ID unico desta pagina
13     int num_keys;          // Numero de chaves ocupadas
14     int is_leaf;
15     int keys[2 * MAX_ORDER - 1];
16     int children[2 * MAX_ORDER]; // IDs dos filhos
17 } BTREE_NODE;
18
19 // Metadados da arvore em RAM
20 typedef struct {

```

```

21 int root_id;      // ID da pagina da raiz (-1 se vazia)
22 int t;           // Grau minimo
23 } BTree;
24
25 // Simulacao do disco
26 extern BTreeNode* disk[MAX_PAGES];
27 extern int NUM_PAGES;          // Numero de paginas usadas
28 extern int CONTA_LEITURA;     // Contador de leituras
29 extern int CONTA_ESCRITA;     // Contador de escritas
30
31 BTree* create_tree(int t);
32 BTreeNode* create_node(int is_leaf);
33
34 // Leitura de linha com INSERT x ou SEARCH x
35 int ler_comando_insert_search(char* linha, int* valor);
36
37 // Identifica comando isolado (modo escrita)
38 int identificar_comando(char* cmd);
39
40 // Escrita no arquivo
41 void EscreverInsert(FILE* f, int valor);
42 void EscreverSearch(FILE* f, int valor);
43
44 // OPERACOES DE DISCO
45 BTreeNode* disk_read(int node_id);
46 void disk_write(int node_id);
47 int create_node_in_disk(int is_leaf);
48
49 // BUSCA
50 int btree_search(BTree* tree, int node_id, int key);
51
52 // INSERCAO
53 void btree_split_child(int parent_id, int index, int t);
54 void btree_insert_nonfull(int node_id, int key, int t);
55 void btree_insert(BTree* tree, int key);
56
57 // IMPRESSAO
58 void imprimir_arvore(int node_id, int nivel);
59
60 #endif

```

Listagem 1. Header das funções e estruturas (function.h)

3.2.1. Árvore B

Para árvore B, foram implementadas as funções para criar a árvore, criar os nós, executar a inserção e split, que já foram explicados anteriormente. A busca é realizada de forma binária e recursiva, o id das páginas (blocos) são localizados seguindo uma lógica binária, ou seja, se a chave não for encontrada e o nó não for folha, a função chama a si mesma recursivamente, passando o id do filho apropriado. Cada chamada recursiva implica em uma nova leitura de disco.

```

1 // Cria a arvore B com grau minimo t

```

```

2 // Validacoes: t deve estar entre 2 e MAX_ORDER
3 // Retorna ponteiro para a arvore ou NULL se parametros invalidos
4 BTree* create_tree(int t) {
5     if (t > MAX_ORDER) {
6         printf("Erro: O grau t (%d) excede o limite (%d).\n", t, MAX_ORDER)
7         ;
8         return NULL;
9     }
10    if (t < 2) {
11        printf("Erro: O grau t deve ser no minimo 2.\n");
12        return NULL;
13    }
14
15    BTree* tree = (BTree*)malloc(sizeof(BTree));
16    tree->t = t;
17    tree->root_id = -1;
18
19    return tree;
}

```

Listagem 2. Função para Criar Árvore (function.c)

```

1 // Cria um novo nó em memoria nao salva no disco ainda
2 BTreeNode* create_node(int is_leaf) {
3     BTreeNode* node = (BTreeNode*)malloc(sizeof(BTreeNode));
4
5     node->is_leaf = is_leaf;
6     node->num_keys = 0;
7     node->page_id = -1;
8
9     for (int i = 0; i < 2 * MAX_ORDER - 1; i++)
10        node->keys[i] = -1;
11
12     for (int i = 0; i < 2 * MAX_ORDER; i++)
13        node->children[i] = -1;
14
15     return node;
16 }

```

Listagem 3. Função para criar Nó (function.c)

```

1 // Realiza busca binaria no nó atual e desce recursivamente para o
2 // filho apropriado
3 int btree_search(BTree* tree, int node_id, int key) {
4     BTreeNode* node = disk_read(node_id);
5     if (node == NULL) return 0;
6
7     int i = 0;
8
9     while (i < node->num_keys && key > node->keys[i])
10        i++;
11
12     if (i < node->num_keys && key == node->keys[i])
13        return 1;
14
15     if (node->is_leaf)

```

```

15     return 0;
16
17     return btree_search(tree, node->children[i], key);
18 }
```

Listagem 4. Funções de Busca Binária (function.c)

```

1 void btree_split_child(int parent_id, int index, int t) {
2     BTreenode* parent = disk_read(parent_id);
3     int full_child_id = parent->children[index];
4     BTreenode* full_child = disk_read(full_child_id);
5     int new_child_id = create_node_in_disk(full_child->is_leaf);
6     BTreenode* new_child = disk[new_child_id];
7
8     new_child->num_keys = t - 1;
9
10    for (int j = 0; j < t - 1; j++)
11        new_child->keys[j] = full_child->keys[j + t];
12
13    if (!full_child->is_leaf) {
14        for (int j = 0; j < t; j++)
15            new_child->children[j] = full_child->children[j + t];
16    }
17
18    full_child->num_keys = t - 1;
19
20    for (int j = parent->num_keys; j >= index + 1; j--)
21        parent->children[j + 1] = parent->children[j];
22
23    parent->children[index + 1] = new_child_id;
24
25    for (int j = parent->num_keys - 1; j >= index; j--)
26        parent->keys[j + 1] = parent->keys[j];
27
28    parent->keys[index] = full_child->keys[t - 1];
29
30    parent->num_keys++;
31
32    disk_write(parent_id);
33    disk_write(full_child_id);
34 }
```

Listagem 5. Função para Split (function.c)

```

1 // Insere chave recursivamente em um no que nao esta cheio
2 // Se interno: desce para filho apropriado, divide se necessario
3 void btree_insert_nonfull(int node_id, int key, int t) {
4     BTreenode* node = disk_read(node_id);
5     int i = node->num_keys - 1;
6
7     if (node->is_leaf) {
8         while (i >= 0 && key < node->keys[i]) {
9             node->keys[i + 1] = node->keys[i];
10            i--;
11        }
12    }
13 }
```

```

13     node->keys[i + 1] = key;
14     node->num_keys++;
15     disk_write(node_id);
16 }
17 else {
18     while (i >= 0 && key < node->keys[i])
19     i--;
20
21     i++;
22
23     int child_id = node->children[i];
24     BTreeNode* child = disk_read(child_id);
25
26     if (child->num_keys == 2 * t - 1) {
27         btree_split_child(node_id, i, t);
28         node = disk_read(node_id);
29         if (key > node->keys[i])
30             i++;
31     }
32
33     btree_insert_nonfull(node->children[i], key, t);
34 }
35 }
36
37 void btree_insert(BTree* tree, int key) {
38     if (tree->root_id == -1) {
39         int root_id = create_node_in_disk(1);
40         BTreeNode* root = disk[root_id];
41         root->keys[0] = key;
42         root->num_keys = 1;
43         tree->root_id = root_id;
44         disk_write(root_id);
45         return;
46     }
47
48     BTreeNode* root = disk_read(tree->root_id);
49
50     if (root->num_keys == 2 * tree->t - 1) {
51         int new_root_id = create_node_in_disk(0);
52         BTreeNode* new_root = disk[new_root_id];
53         new_root->children[0] = tree->root_id;
54
55         btree_split_child(new_root_id, 0, tree->t);
56
57         int i = 0;
58         new_root = disk_read(new_root_id);
59         if (key > new_root->keys[0])
60             i = 1;
61
62         btree_insert_nonfull(new_root->children[i], key, tree->t);
63
64         tree->root_id = new_root_id;
65         disk_write(new_root_id);
66     }
67     else {
68         btree_insert_nonfull(tree->root_id, key, tree->t);

```

```
69 }  
70 }
```

Listagem 6. Funções de Inserção (function.c)

```
1 // Funcao para imprimir a arvore em ordem hierarquica  
2 void imprimir_arvore(int node_id, int nivel) {  
3     BTREE_NODE* node = disk_read(node_id);  
4     if (node == NULL) return;  
5  
6     for (int i = 0; i < nivel * 4; i++)  
7         printf(" ");  
8  
9     printf("Nivel %d: ", nivel);  
10    for (int i = 0; i < node->num_keys; i++) {  
11        printf("%d ", node->keys[i]);  
12    }  
13    printf("\n");  
14  
15    if (!node->is_leaf) {  
16        for (int i = 0; i <= node->num_keys; i++) {  
17            if (node->children[i] != -1) {  
18                imprimir_arvore(node->children[i], nivel + 1);  
19            }  
20        }  
21    }  
22}
```

Listagem 7. Imprimir árvore (function.c)

3.2.2. Manipulação de Disco

Para manipulação de disco foram criadas as seguintes variáveis globais:

- disk[MAX_PAGES]: é a representação da memória secundária. Ele armazena ponteiros para as estruturas BTREE_NODE
- NUM_PAGES:
- CONTA_LEITURA: registra a quantidade de leitura para registro na entrada e saída.
- CONTA_ESCRITA: registra a quantidade de escritas em disco para registro de entrada e saída.

```
1 // Definicao das variaveis globais (simulacao do disco)  
2 BTREE_NODE* disk[MAX_PAGES];  
3 int NUM_PAGES = 0;  
4 int CONTA_LEITURA = 0;  
5 int CONTA_ESCRITA = 0;
```

Listagem 8. Variáveis Globais para Manipulação de disco (function.c)

As funções implementadas foram:

- disk_read: simula a leitura do disco, buscando pelo id do nó e incrementando a variável de contagem de leituras no disco.

- disk_write: registra escrita em um nó do disco simulado e incrementa a variável de contagem de escritas no disco.
- create_node_in_disk: Aloca um novo nó no disco simulado e incrementa a contagem de escritas.

```

1 // Le um no do disco simulado
2 // Incrementa CONTA_LEITURA para registro de I/O
3 BTreenode* disk_read(int node_id) {
4     if (node_id == -1) return NULL;
5     CONTA_LEITURA++;
6     return disk[node_id];
7 }
8
9 // Registra escrita de um no no disco simulado
10 // Incrementa CONTA_ESCRITA para contabilizar operacoes I/O
11 void disk_write(int node_id) {
12     if (node_id != -1) {
13         CONTA_ESCRITA++;
14     }
15 }
16
17 // Cria e aloca um novo no no disco simulado
18 // Incrementa CONTA_ESCRITA automaticamente ao criar
19 int create_node_in_disk(int is_leaf) {
20     BTreenode* node = create_node(is_leaf);
21
22     node->page_id = NUM_PAGES;
23     disk[NUM_PAGES] = node;
24     NUM_PAGES++;
25     CONTA_ESCRITA++;
26     return node->page_id;
27 }
```

Listagem 9. Funções para manipulação de disco (function.c)

3.2.3. Entrada e Saída

A entrada e saída dentro do projeto é feito por meio da interface de linha de comando, onde o comando é identificado e as funções são chamadas.

```

1 // Identifica comando inserido pelo usuario (modo escrita)
2 int identificar_comando(char* cmd) {
3     if (strcmp(cmd, "INSERT") == 0) return 1;
4     if (strcmp(cmd, "SEARCH") == 0) return 2;
5     return 0;
6 }
7
8
9 // Escreve comando INSERT no arquivo operacoes.txt
10 void EscreverInsert(FILE* f, int valor) {
11     fprintf(f, "\nINSERT %d", valor);
12 }
13
14 void EscreverSearch(FILE* f, int valor) {
```

```
15     fprintf(f, "\nSEARCH %d", valor);  
16 }  
17
```

Listagem 10. Funções para entrada e saída (function.c)

4. Conclusão

Conclui-se que o projeto de simulação de disco implementado apresentou resultado positivo, com eficiência na execução e manipulação de memória, facilidade de uso, e entrada e saída. No que tange às considerações de desempenho, a simulação demonstrou a eficácia da Árvore B na redução de latência simulada. Através do monitoramento das variáveis globais de controle de I/O, observou-se que a propriedade de auto-balanceamento e a ordem m da árvore mantêm a altura da estrutura reduzida, minimizando significativamente o número de acessos necessários para operações de busca e inserção. O custo computacional das inserções, embora envolva eventuais operações de split e escrita em novos blocos, manteve-se dentro da complexidade logarítmica esperada, garantindo escalabilidade mesmo com o aumento do volume de dados. Dessa forma, acredita-se que o objetivo principal, simular memória secundária com árvore B, foi alcançado com sucesso.

Referências

- Bayer, R. (2008). B-tree and ub-tree. Acesso em: 9, 12 e 2025.
- Byjus (2025). Design and characteristics of memory hierarchy notes. Acesso em: 9 dez. 2025.
- Contributors, W. (2025a). B-tree. Acesso em: 9 dez. 2025.
- Contributors, W. (2025b). Hierarquia de memória. Acesso em: 9 dez. 2025.
- Contributors, W. (2025c). Unidade de disco rígido. Acesso em: 9 dez. 2025.
- HostMidia (2025). Memórias: o que é e para que serve. Acesso em: 9 dez. 2025.