Sistema de Arquivos Simples Utilizando Árvore B

Universidade do Vale do Itajaí

Gabriel Bartolomeu da Silva[1]
Vinicius Andriani Mazera[2]

01/07/2025

RESUMO

Este trabalho apresenta a implementação de um sistema de arquivos simplificado utilizando a estrutura de dados árvore B. O sistema permite operações básicas de criação, exclusão e navegação hierárquica de arquivos .txt e diretórios, simulando o comportamento de um sistema de arquivos real através de uma interface de linha de comando.

1. Introdução

Com o crescente volume de dados e a necessidade de estruturas eficientes para armazenamento e recuperação de informações, o estudo de sistemas de arquivos se tornou essencial. Este projeto implementa um sistema de arquivos virtual que utiliza árvores B para organizar arquivos e diretórios de forma hierárquica.

A árvore B foi escolhida pelo professor devido à sua ampla utilização em sistemas reais de banco de dados e arquivos, proporcionando operações balanceadas de busca, inserção e remoção mesmo com grandes volumes de dados.

2. Fundamentação Teórica

2.1 Árvore B

A árvore B é uma estrutura de dados auto-balanceada que mantém os dados ordenados e permite operações eficientes. Características principais:

- **Grau mínimo (t):** Define o número mínimo de chaves em cada nó
- **Balanceamento automático:** Mantém a altura da árvore balanceada

2.2 Sistema de Arquivos

Um sistema de arquivos organiza e gerencia dados em dispositivos de armazenamento, fornecendo:

- **Hierarquia:** Estrutura de diretórios e subdiretórios
- **Metadados**: Informações sobre arquivos (nome, tamanho, tipo)
- **Operações:** Criação, leitura, escrita e exclusão

3. Metodologia

3.1 Estruturas de Dados

O sistema utiliza as seguintes estruturas principais:

```c
typedef enum { FILE_TYPE, DIRECTORY_TYPE } NodeType;

typedef struct File {
    char* name;
    char* content;
    size_t size;
} File;

typedef struct Directory Directory;

typedef struct TreeNode {
    char* name;
    NodeType type;
    union {
        File* file;
        Directory* directory;
    } data;
} TreeNode;

typedef struct BTreeNode {
    int n;
    bool leaf;
    char **keys;
    TreeNode **values;
    struct BTreeNode **children;
} BTreeNode;

typedef struct BTree {
    BTreeNode *root;
    int t;
} BTree;

struct Directory {
    BTree* tree;
};
```

3.2 Operações Implementadas

3.2.2 Operações da Árvore B

- **btree_create():** Cria uma nova árvore B
- **btree_insert():** Insere um nó na árvore
- **btree_search():** Busca um elemento por nome
- **btree_delete():** Remove um elemento da árvore
- **btree_traverse():** Percorre a árvore em ordem

3.2.3 Operações de Arquivos

- **Create_txt_file():** Cria arquivos .txt com conteúdo
- **Delete_txt_file():** Remove arquivos .txt do sistema

3.2.4 Operações de Diretórios

- **create_directory():** Cria novo diretório
- **delete_directory():** Remove diretório (vazio)
- **change_directory():** Navega entre diretórios
- **list_directory_contents():** Lista conteúdos de tal diretório

4. Implementação

4.1 Algoritmos Principais

4.1.2 Inserção na Árvore B

```
void btree_insert(BTree* tree, TreeNode* node)
{
    if (tree == NULL)
        return;

    BTreeNode* root = tree->root;

    if (root->n == 2 * tree->t - 1)
    {
        BTreeNode* new_root = create_btree_node(tree->t, false);
        new_root->children[0] = root;
        tree->root = new_root;
        split_child(new_root, 0, tree->t);
        insert_non_full(new_root, node, tree->t);
    }
    else
    {
        insert_non_full(root, node, tree->t);
    }
}
```

```
void insert_non_full(BTreeNode* node, TreeNode* value, int t)
{
    int i = node->n - 1;

    if (node->leaf)
    {
        while (i >= 0 && strcmp(value->name, node->keys[i]) < 0)
        {
            node->keys[i + 1] = node->keys[i];
            node->values[i + 1] = node->values[i];
            i--;
        }
        node->keys[i + 1] = strdup(value->name);
        node->values[i + 1] = value;
        node->n += 1;
    }
    else
    {
        while (i >= 0 && strcmp(value->name, node->keys[i]) < 0)
            i--;
        i++;

        if (node->children[i]->n == 2 * t - 1)
        {
            split_child(node, i, t);
            if (strcmp(value->name, node->keys[i]) > 0)
                i++;
        }
        insert_non_full(node->children[i], value, t);
    }
}
```

4.1.3 Busca na Árvore B

```
TreeNode* btree_search(BTree* tree, const char* name)
{
    if (tree == NULL || tree->root == NULL)
        return NULL;

    BTreeNode* current = tree->root;
    while (current != NULL)
    {
        int i = 0;
        while (i < current->n && strcmp(name, current->keys[i]) > 0)
            i++;

        if (i < current->n && strcmp(name, current->keys[i]) == 0)
            return current->values[i];

        if (current->leaf)
            break;
        else
            current = current->children[i];
    }
    return NULL;
}
```

4.1.4 Remoção na Árvore B

```
void delete_from_node(BTree* tree, BTreeNode* node, const char* name, int t)
{
    int idx = 0;
    while (idx < node->n && strcmp(name, node->keys[idx]) > 0)
        idx++;

    if (idx < node->n && strcmp(name, node->keys[idx]) == 0)
    {
        if (node->leaf)
            remove_from_leaf(node, idx);
        else
            remove_from_non_leaf(tree, node, idx, t);
    }
    else
    {
        if (node->leaf)
            return;

        bool is_last = (idx == node->n);

        if (node->children[idx]->n < t)
        {
            if (idx > 0 && node->children[idx - 1]->n >= t)
            {
                BTreeNode* child = node->children[idx];
                BTreeNode* sibling = node->children[idx - 1];

                for (int i = child->n - 1; i >= 0; i--)
                {
                    child->keys[i + 1] = child->keys[i];
                    child->values[i + 1] = child->values[i];
                }

                if (!child->leaf)
                {
                    for (int i = child->n; i >= 0; i--)
                        child->children[i + 1] = child->children[i];
                }

                child->keys[0] = node->keys[idx - 1];
                child->values[0] = node->values[idx - 1];

                if (!child->leaf)
                    child->children[0] = sibling->children[sibling->n];

                node->keys[idx - 1] = sibling->keys[sibling->n - 1];
                node->values[idx - 1] = sibling->values[sibling->n - 1];

                child->n++;
                sibling->n--;
            }
            else if (idx < node->n && node->children[idx + 1]->n >= t)
            {
                BTreeNode* child = node->children[idx];
                BTreeNode* sibling = node->children[idx + 1];

                child->keys[child->n] = node->keys[idx];
                child->values[child->n] = node->values[idx];

                if (!child->leaf)
                    child->children[child->n + 1] = sibling->children[0];

                node->keys[idx] = sibling->keys[0];
                node->values[idx] = sibling->values[0];

                for (int i = 1; i < sibling->n; i++)
                {
                    sibling->keys[i - 1] = sibling->keys[i];
                    sibling->values[i - 1] = sibling->values[i];
                }

                if (!sibling->leaf)
                {
                    for (int i = 1; i <= sibling->n; i++)
                        sibling->children[i - 1] = sibling->children[i];
                }
```

```
                child->n++;
                sibling->n--;
            }
            else
            {
                if (idx < node->n)
                    merge_nodes(node, idx, t);
                else
                    merge_nodes(node, idx - 1, t);
            }
        }

        if (is_last && idx > node->n)
            delete_from_node(tree, node->children[idx - 1], name, t);
        else
            delete_from_node(tree, node->children[idx], name, t);
    }
}
```

5. “Interface” de Usuário

O sistema oferece uma interface de linha de comando com os seguintes comandos:

- **mkdir <nome>:** Cria diretório
- **touch <nome> "<conteúdo>":** Cria arquivo
- **rm <nome>:** Remove arquivo
- **rmdir <nome>:** Remove diretório
- **ls:** Lista conteúdo
- **cd <caminho>:** Navega entre diretórios
- **exit:** Encerra o programa

6. Resultados

6.1 Testes Funcionais e Exemplos na Execução

Teste 1: Criação de Estrutura Hierárquica

/> mkdir documentos

/> mkdir imagens

/> cd documentos

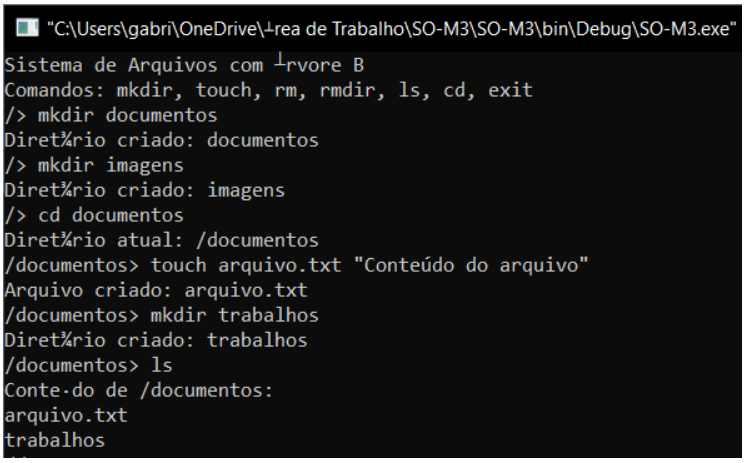
/documentos> touch arquivo.txt "Conteúdo do arquivo"

/documentos> mkdir trabalhos

/documentos> ls

| arquivo.txt

| trabalhos



Teste 2: Navegação e Listagem

/documentos> cd trabalhos

/documentos/trabalhos> touch relatorio.txt

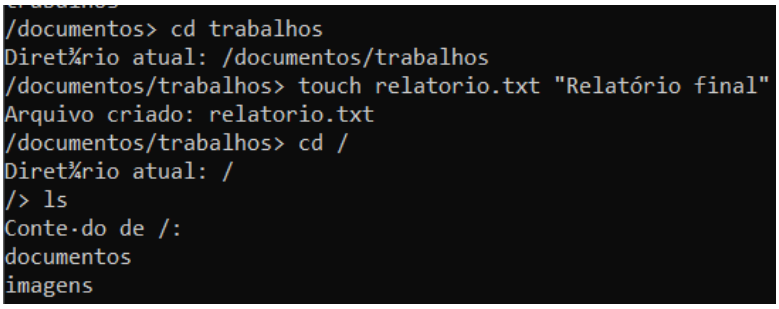
"Relatório final"

/documentos/trabalhos> cd /

/> ls

| documentos

| imagens



Teste 3: Remoção de Elementos

/> cd documentos

/documentos/> mkdir vazio

/documentos> rmdir trabalhos

/documentos> rmdir vazio

/documentos> rm arquivo.txt

/documentos> ls

| trabalhos

```
/> cd documentos
Diret¼rio atual: /documentos
/documentos> mkdir vazio
Diret¼rio criado: vazio
/documentos> rmdir trabalhos
Diret¼rio nÒo estß vazio: trabalhos
Diret¼rio removido: trabalhos
/documentos> rmdir vazio
Diret¼rio removido: vazio
/documentos> rm arquivo.txt
Arquivo removido: arquivo.txt
/documentos> ls
Conte·do de /documentos:
trabalhos
```

(não é apagado o diretório com conteúdo)

7. Conclusão

O sistema de arquivos implementado demonstra com sucesso a aplicação prática de árvores B em um contexto de organização hierárquica de dados. A estrutura escolhida proporciona eficiência nas operações fundamentais, mantendo o desempenho balanceado mesmo com o crescimento do volume de dados.

O projeto atendeu aos requisitos estabelecidos, implementando todas as operações solicitadas. A interface de linha de comando fornece uma experiência familiar aos usuários de sistemas Unix/Linux.

8. Referências

1. **Árvores B (B-trees) para implementação de tabelas de símbolos**. Disponível em: <https://www.ime.usp.br/~pf/estruturas-de-dados/aulas/B-trees.html>. Acesso em: 27 jun. 2025.