Implementação de Árvore B com Persistência em Arquivo

Gabriel Davi

14 de dezembro de 2025

1 Introdução

O objetivo deste trabalho foi adaptar uma implementação clássica de Árvore B em memória principal para funcionar como um índice em disco (memória secundária). A principal modificação necessária foi a substituição de ponteiros de memória (endereços RAM) por referências a registros em arquivo, denominadas RRNs (*Relative Record Numbers*).

Esta adaptação é fundamental para sistemas de banco de dados, onde o volume de registros excede a capacidade da memória RAM, exigindo que a estrutura de índice seja carregada sob demanda através de páginas de tamanho fixo.

2 Estrutura dos Arquivos

Para simular um ambiente de banco de dados real, a implementação manipula dois arquivos binários distintos:

- `dados.dat`: Arquivo sequencial que simula o armazenamento da "carga útil"(os dados completos dos alunos). Novos registros são inseridos no final do arquivo (*append*). O índice da Árvore B armazena apenas o *byte offset* indicando onde cada registro começa.
- `btree_index.dat`: Arquivo que contém a estrutura da árvore propriamente dita. Ele é dividido em:
  - **Cabeçalho (*Header*):** Ocupa os primeiros bytes e armazena o RRN da Raiz e o próximo RRN livre.
  - **Páginas (Nós):** Cada nó possui tamanho fixo (‘sizeof(DiskPage)‘). Isso permite o acesso aleatório (`fseek`) calculado pela fórmula:

$$Offset = Sizeof(Header) + (RRN \times Sizeof(Page))$$

3 Decisões de Projeto e Metodologia

3.1 Representação dos Índices

A implementação adota uma estrutura de índice composta pelo par `<Chave, Endereço>`:

- **Chave:** Inteiro (ID do aluno).
- **Endereço:** Long int (Byte offset no arquivo de dados).

Dessa forma, a Árvore B funciona estritamente como um índice secundário denso, sem carregar dados desnecessários (como nomes) para a memória durante a navegação na árvore.

3.2 Gerenciamento de Disco

Foi implementada uma classe `DiskManager` responsável por encapsular todas as operações de I/O (Input/Output). Duas correções críticas foram aplicadas durante o desenvolvimento para garantir a estabilidade:

1. Uso de `file.clear()` antes de operações de `seek` para resetar flags de erro/EOF.
2. Uso de `file.flush()` após escritas para garantir persistência imediata.

3.3 Tratamento de Overflow

Para evitar corrupção de memória durante a inserção (antes do *split*), os arrays internos da `DiskPage` foram dimensionados com tamanho $2b+1$ (para chaves) e $2b+2$ (para filhos), permitindo que uma página comporte temporariamente o elemento excedente antes de ser dividida.

4 Evidências de Funcionamento e Testes

Para validar a corretude do algoritmo, foi inserida uma massa de dados contendo 21 registros com chaves desordenadas. O objetivo foi forçar múltiplos *splits* e o crescimento da árvore em altura.

4.1 Execução no Terminal

Abaixo, a saída do terminal comprovando a inserção dos registros e o cálculo correto dos offsets no arquivo de dados.

```
PS C:\Users\extre\Organizacao-e-Sistemas-de-Arquivos\OSA-04-TP4ArvoreB> mingw32-make run
g++ -c main.cpp -o main.o
g++ main.o -o main.exe -Wall -Wextra -std=c++17
./main.exe
=== INSERCAO ===

=== ESTRUTURA FINAL ===
Nivel 0 [RRN 8]: 25
Nivel 1 [RRN 2]: 10 20
Nivel 2 [RRN 0]: 5 7 8
Nivel 2 [RRN 4]: 13 15 18
Nivel 2 [RRN 1]: 22 24
Nivel 1 [RRN 7]: 30 40
Nivel 2 [RRN 6]: 26 27
Nivel 2 [RRN 3]: 32 35 38
Nivel 2 [RRN 5]: 42 45 46
Arquivo DOT gerado: arvore.dot
```

Figura 1: Saída do programa mostrando inserção e offsets gerados.

4.2 Visualização da Árvore (Graphviz)

A estrutura final da árvore foi exportada para o formato DOT. A visualização abaixo comprova que os nós estão conectados corretamente através dos RRNs, com a raiz (nó 8) apontando para seus filhos e estes para as folhas.

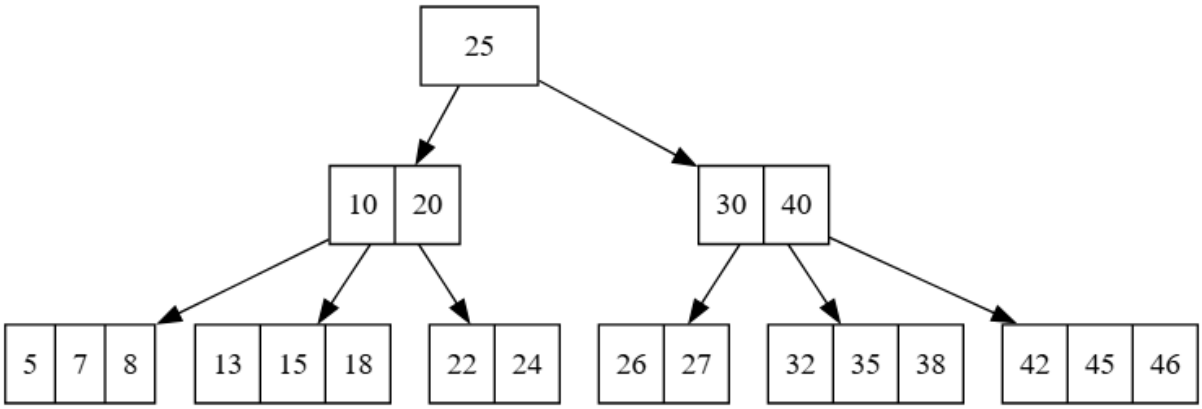

Figura 2: Estrutura final da Árvore B de Ordem 2 (gerada via Graphviz).

5 Código Fonte Principal

Abaixo, apresentamos os trechos essenciais da implementação da classe `BTree` adaptada para disco.

```
// Estrutura da Pagina de Disco
struct DiskPage {
    int pageRRN;
    int numKeys;
    bool isLeaf;
    RecordIndex keys[MAX_KEYS];
    int children[MAX_CHILDREN];
    // ... construtor omitido
};

// Logica de Insercao Recursiva em Disco
InsertResult insertRecursive(int currentRRN, const RecordIndex& item) {
    // 1. Le a pagina do disco
    DiskPage page = dm.readPage(currentRRN);

    // 2. Encontra posicao e verifica duplicata
    int i = 0;
    while (i < page.numKeys && item.key > page.keys[i].key) {
        i++;
    }
    if (i < page.numKeys && page.keys[i].key == item.key) {
        return {false, RecordIndex(), NULL_PTR};
    }

    if (page.isLeaf) {
        // Abre espaco movendo chaves para direita (Shift)
        for (int k = page.numKeys; k > i; k--) {
            page.keys[k] = page.keys[k-1];
```

```
        }
        page.keys[i] = item;
        page.numKeys++;

        dm.writePage(currentRRN, page);

        if (page.numKeys > 2 * ORDER_B) {
            return splitNode(page);
        }
        return {false, RecordIndex(), NULL_PTR};
    }
    else {
        // No interno: desce para filho
        InsertResult res = insertRecursive(page.children[i], item);
        // ... (tratamento de retorno omitido)
    }
}
```

Listing 1: Estruturas de Dados e Inserção

6 Conclusão

O trabalho alcançou com êxito o objetivo de implementar a persistência de uma Árvore B. Os testes visuais e a análise dos arquivos binários gerados confirmam que a gestão de RRNs e o controle de páginas em disco estão funcionando conforme a teoria de Organização de Arquivos.