# Relatório do Trabalho 2: Comparação de Desempenho em Árvores Vermelho-Preto e Árvores 2-3

João Marcos Sousa Rufino Leal

jsousarufinoleal@ufpi.edu.br

*Estruturas de Dados II - Juliana Oliveira de Carvalho*

**Abstract**

O presente relatório descreve a aplicação das estruturas de dados em árvore, especificamente a Árvore 2-3 (A23) e a Árvore Vermelho e Preto (AVP), para a solução de um problema predefinido. Nesse contexto, foram desenvolvidos algoritmos para a implementação de ambas as estruturas. A partir dos resultados obtidos, realizou-se uma análise comparativa de desempenho focada no tempo de execução da operação de busca e inserção, na qual são expostos detalhes com o objetivo de determinar qual estrutura de dados demonstra maior eficiência.

**Palavras-chave:** Estruturas de Dados, Árvore Vermelho e preto, Árvore 2-3, Balanceamento, Operações de Inserção e Busca

## 1 Introdução

Este relatório apresenta uma análise comparativa da implementação e aplicabilidade de Árvores Vermelho-Preta e Árvores 2-3, estruturas de dados auto-balanceáveis empregadas na linguagem C. Ambas foram utilizadas para construir um sistema de informações geográficas e demográficas, que articula múltiplas estruturas: uma lista duplamente encadeada e ordenada para os Estados, onde cada um aponta para uma árvore de Cidades, que por sua vez contém uma árvore de CEPs. Adicionalmente, uma árvore separada é usada para o cadastro de Pessoas.

O objetivo central é avaliar duas versões completas do sistema: a primeira implementada com Árvores Vermelho-Preta e a segunda, recriada com Árvores 2-3. A metodologia envolveu o desenvolvimento de todas as funcionalidades de cadastro, remoção e consulta exigidas. Isso inclui desde o cadastro progressivo de dados (como adicionar cidades a um estado já existente) até a execução de buscas complexas, como encontrar o estado mais populoso ou o número de pessoas que não moram na cidade onde nasceram.

A análise comparativa focada no desempenho computacional como também na complexidade de implementação e na manipulação das árvores referidas, com foco nas operações de inserção e busca. O documento está organizado em seções que descrevem as estruturas de dados de cada modelo, a metodologia de desenvolvimento, os resultados funcionais e, por fim, as conclusões sobre as vantagens e desvantagens de cada estrutura para a resolução do problema proposto.

# 2 Hardware Utilizado

Todos os testes demonstrados neste relatório foram feitos utilizando o mesmo hardware com as seguintes especificações.

Table 1: Especificações de Hardware

| **Componente** | **Especificação** |
|---|---|
| Marca/Modelo | Samsung Galaxy Book 2 |
| Processador | Intel Core i5-1235U (12ª geração) |
| Núcleos / Threads | 10 núcleos (2 P-cores + 8 E-cores) / 12 threads |
| Memória RAM | 16 GB LPDDR4x 3200 MHz |
| Armazenamento | 256 GB SSD NVMe |
| Sistema Operacional | Windows 11 Home |

# 3 Seções Específicas

As seções a seguir descrevem as estruturas utilizadas, o funcionamento das principais funções implementadas e os testes realizados para avaliação de desempenho.

## 3.1 Estruturas de Dados Utilizadas

Foram utilizadas duas estruturas de dados lineares e não lineares: lista duplamente encadeada, árvore vermelha e preta e a árvore 2-3.

- **Lista Duplamente Encadeada**

  A lista duplamente encadeada é uma estrutura de dados linear na qual cada elemento, ou nó, armazena não apenas uma referência para o nó próximo, mas também uma para o nó anterior na sequência. Essa estrutura de ponteiro duplo é sua característica fundamental, permitindo que a lista seja percorrida eficientemente em ambas as direções, para frente e para trás. Graças a essa capacidade de travessia bidirecional, operações como a inserção ou remoção de um nó em qualquer posição se tornam mais simples e diretas, pois o acesso ao elemento anterior é imediato, sem a necessidade de percorrer a lista desde o início.

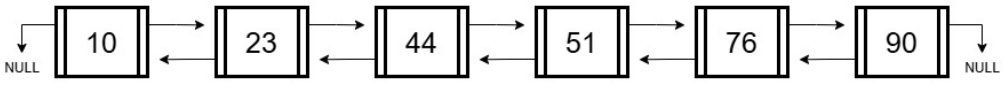


Figure 1: Lista Duplamente Encadeada

- **Árvore Vermelha e Preta**

  A árvore vermelha e preta consiste em um tipo especial de árvore binária de busca, na qual cada nó tem um atributo de cor, podendo ter os valores VERMELHO e PRETO. O tipo de árvore vermelha e preta utilizada é a de pendente para a esquerda, ou seja, é organizada de forma que os nós de cor vermelha fiquem sempre à esquerda de um nó pai

(Específico da Árvore vermelha e preta de inclinação para a esquerda); o nó raiz é sempre preto; um novo nó é criado vermelho. Para que essas regras sejam cumpridas durante a inserção, é feito o balanceamento da árvore com base em 3 condições comparativas: se o nó à direita é vermelho, se há 2 nós seguidos à esquerda da cor vermelha e se os dois filhos são da cor vermelha, onde para cada um destes casos há uma modificação específica, sendo respectivamente a rotação para a esquerda, a rotação para a direita e a troca de cores.

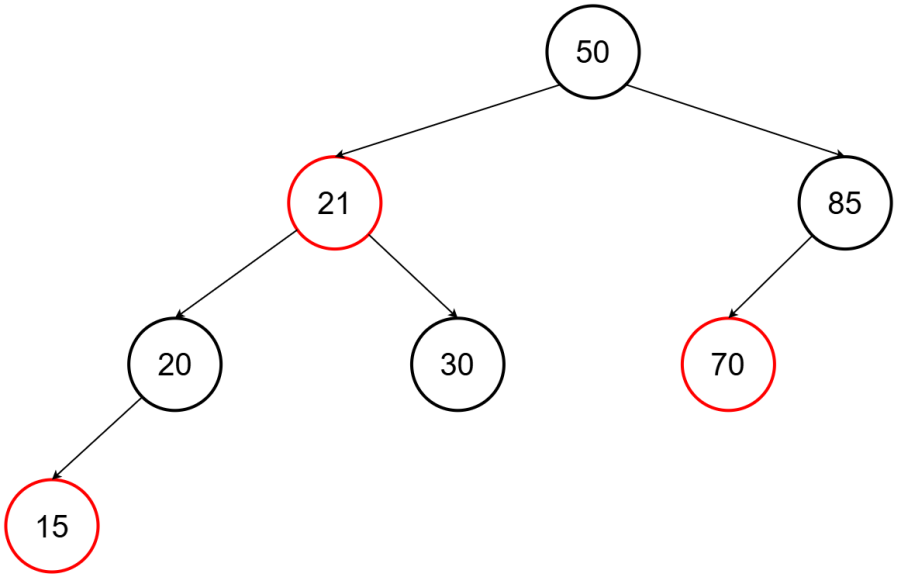


Figure 2: Árvore Vermelha e Preta

- **Árvore 2-3**

  A árvore 2-3 é um tipo de estrutura em árvore que pode armazenar até dois elementos em um único nó e, consequentemente, possuir dois ou três filhos, que são ponteiros para outros nós de mesma natureza. Sempre que um nó com um elemento não for uma folha, ele terá dois filhos, um à esquerda e outro à direita; já se um nó com dois elementos não for folha, ele deverá ter três filhos, sendo um à esquerda, um no centro e outro à direita. Após encontrar o local de inserção, o balanceamento da árvore deve ser feito de acordo com suas propriedades. A principal operação para isso é a de dividir o nó (ou "quebrar"), que é acionada quando se tenta inserir um elemento em um nó já completo, sempre promovendo uma das chaves para o nó pai a fim de manter a estrutura equilibrada.

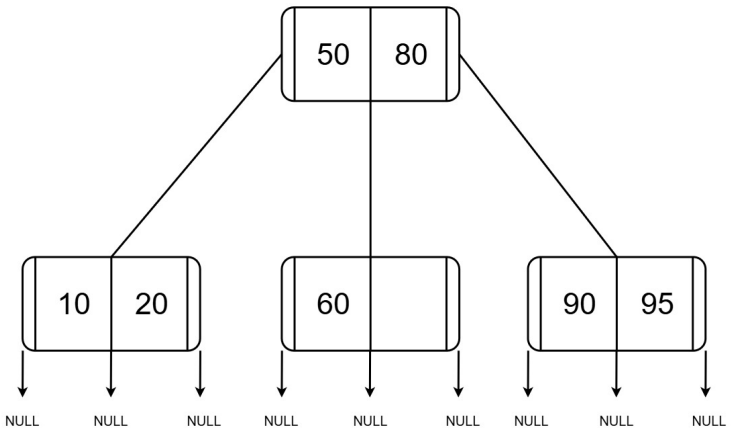


Figure 3: Árvore 2-3

### 3.1.1 Implementação da Lista Duplamente Encadeada

Esta seção apresentará as funções utilizadas para a implementação da Lista Duplamente Encadeada, que no sistema é responsável por organizar os Estados.

1. **Estados_*.c**

   - `existe_estado`: Recebe a cabeça da lista (cabeca) e um nome de estado. Percorre a lista duplamente encadeada comparando os nomes e retorna o ponteiro para o nó do estado, se encontrado, ou NULL, caso contrário.
   - `aloca_estado`: Aloca memória para um novo nó ESTADOS, recebendo o nome do estado como parâmetro. Inicializa seus campos numéricos como 0 e os ponteiros (nome_capital, cidade, prox, ant) como NULL. Retorna o ponteiro para o novo nó.
   - `inserir_estado_rec`: Insere um novo estado de forma ordenada na lista duplamente encadeada. A função opera de maneira recursiva, garantindo que a ordem alfabética pelo nome do estado seja mantida. Retorna 1 em caso de sucesso e 0 se o estado já existir.
   - `cadastro_estado`: Solicita ao usuário que digite o nome de um estado, chama aloca_estado para criar o nó e retorna o novo estado pronto para ser inserido na lista.
   - `liberar_todos_estados`: Percorre toda a lista de estados a partir da cabeça e chama liberar_no_estado para cada um, desalocando toda a memória utilizada pela lista e suas sub-estruturas.
   - `liberar_no_estado`: Libera a memória de um único nó de estado, incluindo a desalocação do nome, da capital e da árvore de cidades associada a ele.
   - `mostrar_estado`: Recebe um ponteiro para um nó de estado e imprime na tela suas informações principais, como nome e capital.
   - `mostrar_todos_estados`: Percorre a lista de estados e utiliza a função mostrar_estado para imprimir as informações de todos os estados cadastrados.
   - `verifica_estado_mais_populoso`: Percorre a lista de estados, comparando a população de cada um para encontrar e retornar um ponteiro para o estado com a maior população.
     ***Essa implementação de estados é seguida para as duas versões do sistema.***

### 3.1.2 Implementação da Árvore Vermelha e Preto

Esta seção apresentará as funções utilizadas para a implementação das Árvores Vermelho-Preta, que no sistema são responsáveis por organizar as Cidades, os CEPs e as Pessoas.

1. **Funções Comuns de Manipulação da Árvore Vermelho-Preta**

   - **Inserção**: Funções como `inserir_no_*` e `inserir_*` adicionam um novo nó de forma recursiva e garantem que a raiz da árvore permaneça com a cor PRETA.
   - **Alocação e Cadastro**: Funções como `aloca_*` e `cadastrar*` são responsáveis por alocar memória para um novo nó e preenchê-lo com dados fornecidos pelo usuário.

- **Balanceamento**: Um conjunto de funções como `rotacao_esquerda_*`, `rotacao_direita_*`, `Cor_*`, `trocar_cor_*` e `balancear_RB_*` aplicam as regras da Árvore Vermelho-Preta para manter o balanceamento durante as inserções e remoções.
- **Busca e Impressão**: Funções como `existe_*` buscam um nó pela chave. Já `imprimir_*`, `imprimir_*_em_ordem` e `imprimir_todos_*` exibem as informações armazenadas nos nós.
- **Remoção**: Um conjunto de funções como `remover_*_arvore`, `remover_*_no`, `encontrar_menor_*`, entre outras, localiza um nó pela chave, o remove e rebalanceia a árvore para manter suas propriedades.
- **Desalocação**: Funções como `limpar_*` ou `desalocar_*` liberam a memória alocada por um nó específico ou por toda a árvore de maneira recursiva.

2. **Cidades_VP.c**

- `verifica_cidade_mais_populosa_nao_capital`: Percorre a árvore de cidades de um estado e retorna um ponteiro para a cidade mais populosa, ignorando a capital.
- `cep_pertence_a_cidade`: Verifica se uma string de CEP pertence à árvore de CEPs de uma determinada cidade.
- `cidade_natal_dado_cep`: Busca em uma árvore de cidades qual delas contém um determinado CEP, retornando o ponteiro para a cidade correspondente.
- `quantas_pessoas_nascidas_na_cidade_nao_moram_na_cidade`: Conta o número de pessoas que nasceram em uma cidade específica, mas cujo CEP de moradia atual não pertence a essa mesma cidade.

3. **CEPs_VP.c**

- `consulta_CEP`: Verifica se um CEP existe na árvore de CEPs, retornando 1 se encontrado e 0 caso contrário.
- `percorre_estados_procurando_CEP`: Percorre toda a estrutura de dados de estados e suas respectivas cidades, verificando se um determinado CEP já existe em qualquer local do sistema.
- `percorre_cidades_procurando_CEP`: Percorre uma árvore de cidades buscando um CEP específico em suas respectivas árvores de CEPs.

4. **Pessoas_VP.c**

- `trocar_informacoes_pessoas`: Troca todas as informações (nome, CPF, CEPs, data) entre dois nós de pessoa. Utilizada no processo de remoção de nós da árvore de pessoas.
- `verifica_pessoa_nascida_ou_que_mora_na_cidade`: Percorre a árvore de pessoas verificando se alguma possui o CEP informado como CEP de nascimento ou CEP atual.
- `quantas_pessoas_nao_moram_na_cidade_natal_ESTADO`: Função de alto nível que percorre a lista de estados para contar todas as pessoas que não residem atualmente em suas cidades natais.
- `quantas_pessoas_nao_moram_na_cidade_natal_PESSOAS`: Percorre a árvore de pessoas e, para cada uma, verifica se seu CEP atual pertence à sua cidade natal, incrementando um contador caso não pertença.

- `quantas_pessoas_moram_na_cidade_nao_nasceram_nela`: Para uma cidade específica, percorre a árvore de pessoas e conta quantas moram nela (com base no CEP atual), mas não nasceram nela (CEP de nascimento diferente).

### 3.1.3 Implementação da Árvore 2-3

Esta seção apresentará as funções utilizadas para a implementação das Árvores 2-3, que no sistema são responsáveis por organizar as Cidades, os CEPs e as Pessoas..

1. **Funções Comuns — Árvore 2-3**

   - `cria_no_*`: Aloca memória para um novo nó da árvore 2-3, inicializando-o como um 2-nó, com uma informação e dois filhos nulos.
   - `eh_folha_*`: Verifica se um determinado nó da árvore 2-3 é uma folha, ou seja, não possui filhos.
   - `adiciona_infos_*`: Adiciona uma segunda informação a um 2-nó, convertendo-o em um 3-nó, sem alterar a estrutura da árvore.
   - `quebra_no_*`: Quando um 3-nó recebe uma terceira informação, esta função divide o nó em dois 2-nós e promove a informação do meio para o nó pai.
   - `insere_23_recursivo_*`, `insere_23_*`: Funções principais de inserção em árvore 2-3. Descendem na árvore, inserem informações, realizam divisões e podem aumentar a altura da árvore se necessário.
   - `buscar_info_*`: Percorre a árvore em busca de um nó que contenha uma informação específica.
   - `buscar_menor_elemento_*`: Encontra o menor valor em uma subárvore, sendo geralmente utilizada durante o processos de remoção.
   - `remover_23_*`, `remover_23_*_recursivo`: Funções que localizam e removem uma informação da árvore 2-3, gerenciando a reorganização dos nós.
   - `tratar_underflow_*`: Em casos de underflow (quando um nó fica com menos informações que o permitido), esta função coordena as estratégias de correção.
   - `redistribuir_com_irmao_*`: Rebalanceia a árvore redistribuindo informações com um nó irmão adjacente para resolver o underflow.
   - `fundir_com_irmao_*`: Realiza a fusão entre o nó em underflow e um nó irmão, descendo uma informação do pai e formando um novo nó válido.
   - `imprime_23_em_ordem_*`: Percorre e imprime os elementos da árvore 2-3 em ordem crescente.
   - `libera_arvore_*`: Desaloca recursivamente todos os nós da árvore, liberando toda a memória utilizada.

2. **Cidades_23.c**

   - `cadastra_cidade`: Solicita ao usuário o nome e a população, retornando uma estrutura `CIDADES` preenchida.

- `printar_informacoes_cidade`: Imprime os detalhes de uma única cidade, como nome e população.
- `verifica_cidade_mais_populosa_nao_capital_23`: Percorre a árvore 2-3 de cidades de um estado para encontrar e retornar um ponteiro para a cidade mais populosa que não seja a capital.
- `cep_pertence_a_cidade`: Verifica se uma string de CEP existe na árvore 2-3 de CEPs associada a uma cidade.
- `cidade_dado_cep`: Busca em uma árvore de cidades qual delas contém um determinado CEP, retornando o ponteiro para a cidade correspondente.

3. **CEPs_23.c**

- `consulta_CEP`: Verifica se uma string de CEP existe em uma árvore 2-3 de CEPs, retornando 1 em caso afirmativo e 0 caso contrário.
- `percorre_estados_procurando_CEP`: Percorre a lista de estados e, para cada um, chama a função `percorre_cidades_procurando_CEP` para validar a existência de um CEP em todo o sistema.
- `percorre_cidades_procurando_CEP`: Percorre a árvore 2-3 de cidades para encontrar um CEP específico.

4. **Pessoas_23.c**

- `cadastra_pessoa`: Solicita ao usuário os dados de uma pessoa (nome, CPF, data de nascimento) e retorna a estrutura `PESSOAS` preenchida.
- `imprimir_dados_PESSOAS`: Imprime todos os detalhes de uma pessoa.
- `verificar_se_existe_pessoa_associada_a_um_CEP`: Percorre a árvore de pessoas para verificar se alguma delas nasceu ou mora em um local com o CEP fornecido, usado para impedir a remoção de CEPs em uso.
- `quantas_pessoas_nao_moram_na_cidade_natal_ESTADO`: Função de alto nível que percorre todos os estados para iniciar a contagem de pessoas que não residem em sua cidade natal.
- `quantas_pessoas_moram_na_cidade_nao_nasceram_nela`: Para uma cidade específica, conta quantas pessoas moram nela (CEP atual), mas não nasceram lá (CEP natal diferente).
- `quantas_pessoas_nao_moram_na_cidade_natal`: Conta o número de pessoas que não moram na cidade correspondente ao seu CEP de nascimento.
- `quantas_pessoas_nascidas_na_cidade_nao_moram_nela`: Conta quantas pessoas nasceram em uma cidade específica, mas cujo CEP de moradia atual não pertence a essa mesma cidade.

# 4 Considerações sobre a Árvore 4-5

A árvore 4-5, que seria usada para gerenciar os blocos de memória, acabou não sendo implementada por completo por alguns motivos. Primeiro, a proposta era bastante complexa, principalmente porque envolvia manter nós intercalados para representar blocos livres e ocupados. Isso exigiria manipulações constantes para unir blocos adjacentes e atualizar a estrutura

da árvore toda vez que um bloco mudasse de estado (de livre para ocupado, por exemplo), o que demandaria bastante tempo e atenção.

Além disso, outros compromissos acadêmicos e profissionais acabaram limitando bastante o tempo disponível, o que dificultou o desenvolvimento completo dessa parte do projeto. Por isso, a prioridade foi concluir as árvores 2-3 e vermelho-preto, que já envolviam várias funcionalidades e testes.

Apesar disso, como as funções de inserção e busca da árvore 4-5 foi implementada, ela será utilizada para realizar uma análise de desempenho. No entanto, as demais funções específicas pedidas não foram desenvolvidas, então optamos por não incluir no trabalho informações completas sobre a estrutura da árvore 4-5.

# 5 Testes de Desempenho

A presente seção dedica-se a uma análise comparativa detalhada do desempenho das árvores estudadas, com foco na eficiência computacional de suas operações fundamentais: inserção e busca. Para simular um cenário de uso em larga escala e avaliar o comportamento das estruturas sob carga, o desempenho foi aferido com base no tempo necessário para popular cada árvore com um universo de 1.000.000 de números e, posteriormente, para executar 1.000 consultas distintas na estrutura resultante. Os dados coletados a partir deste experimento fornecem a base para as tabelas a seguir.

## 5.1 Tabelas de Resultados

A seguir, são apresentadas as tabelas que sintetizam os resultados obtidos nos experimentos. Os valores expostos representam o tempo médio, expresso em milissegundos (ms), para a execução das operações em cada tipo de árvore testada.

Table 2: Desempenho da Árvore 2-3

| Tipo de Teste | Inserção (ms) | Consulta (ms) |
|---|---|---|
| Crescente | 0,000 244 | 0,000 000 |
| Decrescente | 0,000 233 | 0,000 500 |
| Aleatório | 0,000 734 | 0,000 000 |

Table 3: Desempenho da Árvore 4-5

| Tipo de Teste | Inserção (ms) | Consulta (ms) |
|---|---|---|
| Crescente | 0,000 186 | 0,000 000 |
| Decrescente | 0,000 209 | 0,000 000 |
| Aleatório | 0,000 590 | 0,000 000 |

Table 4: Desempenho da Árvore Rubro-Negra

| Tipo de Teste | Inserção (ms) | Consulta (ms) |
|---|---|---|
| Crescente | 0,000 429 | 0,000 500 |
| Decrescente | 0,000 720 | 0,000 500 |
| Aleatório | 0,000 770 | 0,001 000 |

Table 5: Comparação Geral de Desempenho das Árvores

| Árvore | Tipo de Teste | Inserção (ms) | Consulta (ms) |
|---|---|---|---|
| 2-3 | Crescente | 0,000 244 | 0,000 000 |
| 2-3 | Decrescente | 0,000 233 | 0,000 500 |
| 2-3 | Aleatório | 0,000 734 | 0,000 000 |
| 4-5 | Crescente | 0,000 186 | 0,000 000 |
| 4-5 | Decrescente | 0,000 209 | 0,000 000 |
| 4-5 | Aleatório | 0,000 590 | 0,000 000 |
| Rubro-Negra | Crescente | 0,000 429 | 0,000 500 |
| Rubro-Negra | Decrescente | 0,000 720 | 0,000 500 |
| Rubro-Negra | Aleatório | 0,000 770 | 0,001 000 |

- **Árvore 4-5:** Apresenta o melhor desempenho em inserção (0.000186–0.000590 ms), com consultas extremamente rápidas (0 ms), devido à maior capacidade de nós, o que reduz a altura da árvore.
- **Árvore 2-3:** Tempos de inserção (0.000233–0.000734 ms) são ligeiramente mais altos que a 4-5, com consultas rápidas (0–0.0005 ms), consistentes com a complexidade esperada de $O(\log n)$.
- **Árvore Rubro-Negra:** Inserção mais lenta (0.000429–0.000770 ms) devido a ajustes de balanceamento mais complexos, e consultas (0.0005–0.001 ms) um pouco mais altas, mas ainda dentro do esperado para $O(\log n)$.

**Visão geral:** Os tempos obtidos refletem a complexidade teórica de $O(\log n)$ para um universo de 1.000.000 de elementos. Os resultados práticos posicionam a árvore 4-5 como a mais eficiente para este conjunto de dados, seguida pela 2-3 e, por fim, pela Rubro-Negra, que apresentou custos operacionais ligeiramente maiores, como é esperado pela sua estrutura.

# 6 Conclusão

A análise dos tempos de inserção revela uma clara vantagem de desempenho para as árvores baseadas na estrutura B-Tree, com a árvore 4-5 se destacando como a mais eficiente em todos os cenários propostos. Em contrapartida, a árvore Rubro-Negra demonstrou ser a mais lenta para inserir elementos, especialmente nos casos com dados em ordem decrescente e aleatória. Este comportamento sugere que suas operações de balanceamento, como rotações e

trocas de cor, foram mais intensivas em comparação com as operações de divisão de nó (split) das árvores 2-3 e 4-5 para o conjunto de dados utilizado. No que tange ao desempenho de consulta, a superioridade das árvores 2-3 e 4-5 é ainda mais pronunciada, com tempos de busca praticamente instantâneos na maioria dos testes. A árvore Rubro-Negra foi a única a apresentar um custo de consulta consistentemente mensurável, atingindo seu pico também no cenário com dados aleatórios. A combinação desses resultados indica um desempenho geral superior das variantes de B-Tree para este problema, que se mostraram mais rápidas tanto na construção da estrutura quanto na posterior recuperação dos dados.