Trabalho Prático 3

Consultas ao Sistema de Despacho CabeAí

Luísa Barros Ribeiro Andrade - 202303412
Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil
luisabra@ufmg.br

1 Introdução

O trabalho tem como objetivo desenvolver um sistema de indexação e consulta de endereços para a empresa *Cabe Aí*, capaz de processar uma base de dados de logradouros e localizar coordenadas a partir de termos de busca. Assim, o problema consiste em utilizar estruturas de dados eficientes para construir um índice invertido e garantir a recuperação rápida de informações geográficas a fim de agilizar o despacho de veículos. Portanto, o desafio central é implementar um mecanismo de busca que processe consultas com múltiplos termos, realizando a interseção de resultados para retornar endereços corretos.

Foi necessária a implementação desse sistema de forma manual para todas as estruturas de dados fundamentais, incluindo Listas Encadeadas para armazenamento das ocorrências de cada termo e uma **Árvore Binária Balanceada (Árvore AVL)** para estruturação do índice de palavras.

Além dos requisitos fundamentais, este trabalho apresenta diferentes estruturas de indexação, de maneira que, foram implementadas alternativas baseadas em **Tabela Hash** e **Árvore B**, visando avaliar quantitativamente o *trade-off* entre eficiência temporal, analisando os tempos de indexação e consulta, e custo espacial, analisando o consumo de memória. Assim, foi possível comparar essas três formas de implementação para o índice invertido.

2 Método

O programa foi desenvolvido na linguagem C++, compilado pelo GCC da GNU Compiler Collection. A máquina utilizada possui as seguintes especificações:

- Sistema operacional: Ubuntu 24.04.2 LTS
- Compilador: gcc (Ubuntu 13.3.0-6ubuntu2 24.04) 13.3.0
- Processador: 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz
- RAM: 8 GB ((utilizável: 7,73 GB)

2.1 Estruturas de Dados

O sistema segue o padrão de Orientação a Objetos e foi modularizado em torno de cinco Tipos Abstratos de Dados (TADs) principais, implementados em `endereço.cpp, logradouro.cpp, lista.cpp, palavra.cpp` e `consulta.cpp`, onde cada componente possui responsabilidades isoladas.

Assim, para cumprir com o requisito fundamental do trabalho, que pede a utilização de uma Árvore de Pesquisa Balanceada, no componente central do sistema (Índice de Palavras), optou-se pela implementação da **Árvore AVL**. Porém, para atender aos critérios de pontos extras, o sistema foi expandido para suportar também uma **Tabela Hash** e uma **Árvore B**, permitindo que o índice de palavras opere sobre as três estruturas e, que haja uma avaliação comparativa entre as diferentes implementações. A alternância entre essas estruturas é realizada em tempo de compilação, permitindo isolar e medir o desempenho de cada abordagem, sem modificar a lógica de consulta.

O **TAD Endereço** define a entidade atômica da base de dados, armazenando os atributos brutos lidos do arquivo (ID, logradouro, número, coordenadas, etc.). A principal função deste TAD é servir como um objeto de transferência de dados temporário durante a leitura, permitindo que o sistema ordene e agregue registros dispersos antes de consolidá-los em logradouros únicos.

O **TAD Logradouro** representa a consolidação espacial de múltiplos endereços. Ele implementa a lógica espacial do sistema, calculando o "centro de gravidade", a partir da agregação de múltiplos endereços (coordenadas), permitindo que o sistema responda com um ponto único de destino, conforme especificado no problema, que representa geometricamente toda a extensão da via, mitigando erros de geolocalização em ruas extensas.

O **TAD Lista** é uma estrutura de suporte fundamental do projeto, se tratando de uma lista encadeada simples, implementada manualmente, com ponteiros para o início e o fim, realizando inserção $O(1)$. Ela é utilizada transversalmente em todo o sistema para tratar colisões na Tabela Hash, para armazenar IDs nos nós das árvores (AVL e B) e para manter os conjuntos de resultados durante as operações de interseção na consulta.

O **TAD Palavra** implementa o Índice Invertido, sendo o componente mais complexo e alvo das otimizações de Pontos Extras. Ele mapeia termos de busca para listas de IDs de logradouros. Sua estrutura interna é definida via comandos de pré-processamento , podendo assumir cada uma das seguintes versões:

- Árvore AVL (versão padrão): Uma árvore binária de busca auto-balanceada, onde cada nó armazena uma palavra, sua altura e uma lista de ocorrências.
- Tabela Hash (versão extra): Um vetor de ponteiros para listas encadeadas, otimizado para acesso médio $O(1)$.
- Árvore B (versão extra): Uma árvore balanceada multi-caminho de ordem t, projetada para reduzir a altura da estrutura e para agrupar as chaves, maximizando a localidade de referência.

O **TAD Consulta** atua como o módulo responsável pela coordenação da busca, adaptado para interagir com a interface genérica do `IndicePalavras`, fazendo com que ele seja independente do tipo de estrutura de dados usada (AVL, Hash ou Árvore B). O TAD é responsável por resolver consultas compostas por meio da interseção de listas e ranquear os resultados. A estrutura é responsável por receber a string de consulta do usuário, interagir com o índice de palavras para recuperar candidatos e aplicar a lógica de negócios para gerar o relatório final.

2.2 Funções Implementadas

No TAD Endereço, além do construtor de inicialização, que inicializa todos os campos (string e numéricos) a partir da leitura do arquivo, foram implementadas funções básicas de acesso, os **getters** (`getIdLog, getLat, getLon`), que permitem que o módulo principal acesse as chaves de ordenação e as coordenadas necessárias para o cálculo do centro de gravidade no TAD Logradouro.

No TAD Logradouro, a lógica concentra-se na aritmética espacial. Dessa forma, foram implementadas as seguintes funções:

- `adicionarEndereco`, que recebe as coordenadas de um novo endereço pertencente à via e as soma aos acumuladores internos, incrementando também a contagem de endereços.
- `calcularCentroDeGravidade`, que realiza a média aritmética das latitudes e longitudes acumuladas. Essa função é chamada uma única vez após o agrupamento de todos os endereços, definindo a posição final da via.
- `distanciaAte`, que implementa a fórmula da distância euclidiana ($\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$) entre o centro de gravidade do logradouro e o ponto de origem da consulta do usuário.

No TAD Lista as seguintes funções foram implementadas para garantir a gestão dinâmica de memória:

- `inserir`, que adiciona um novo ID de logradouro ao final da lista. Mantém um ponteiro para o último nó, garantindo complexidade constante $O(1)$ para inserções, crucial durante a carga de dados.
- `contem`, que realiza uma busca na lista para verificar se um ID já existe. Esta função é vital para lógica de interseção de conjuntos, ou seja, ele verifica se um logradouro encontrado na palavra A também existe na lista da palavra B.
- `~ListaEncadeada`, que é o destrutor e, percorre a lista liberando a memória de cada nó individualmente, prevenindo vazamentos (*memory leaks*) na estruturas complexas como a Tabela Hash.

No TAD Palavra, as funções variam conforme a estrutura selecionada para os Pontos Extras:

- Na versão de Árvore AVL foram implementadas as funções:
  - `inserirRecursivo`, que percorre a árvore comparando strings lexicograficamente e insere um novo nó, por fim, atualiza a altura dos nós no caminho de volta.

  - `rotacaoDireita/rotacaoEsquerda`, que realizam o rebalanceamento da árvore quando o fator de balanceamento excede $|1|$, garantindo a propriedade $O(\log V)$ da AVL.
  - `buscarRecursivo`, que navega pela árvore para encontrar a palavra chave e retornar sua lista de IDs.

- Na versão de Tabela Hash foram implementadas as funções:
  - `funcaoHash`, que implementa algoritmo *djb2* modificado, que converte a string em um índice numérico e minimiza colisões através de operações de *bitshift*.
  - `inserirPalavra`, que calcula o índice e insere a palavra na lista encadeada correspondente, para tratamento de colisão.
  - `buscarPalavras`, que acessa diretamente a posição do vetor calculado pelo hash, retornando a lista associada.

- Na versão de Árvore B foram implementadas as funções:
  - `splitFilho`, que divide o nó cheio em dois, promovendo a chave mediana para o nó pai, mantendo a propriedade fundamental da Árvore B.
  - `inserirNaoCheio`, que gerencia a inserção de chaves em nós folhas ou direcionamento para o filho correto, lidando com arrays de chaves ordenadas.

No TAD Consulta, as funções implementam a heurística de busca:

- `processar`, que divide a string de consulta em palavras individuais. Para cada palavra, busca a lista de logradouros no índice. Se for a primeira palavra, inicia o conjunto resultado e, para as palavras subsequentes realiza a interseção, mantendo apenas IDs presentes em ambas as listas.
- `buscaBinariaLogradouro`, que dado um ID retornado pelo índice, localiza o ponteiro para o objeto Logradouro completo no array mestre. Utiliza busca binária $O(\log N)$ para eficiência.
- `mergeSort`, que ordena o array final de candidatos com base na distância calculada, garantindo que a saída respeite a proximidade geográfica solicitada.

2.3 Lógica de Controle

A orquestração dos TADs e o fluxo de execução do programa foram segregados em dois arquivos distintos, visando separar a implementação estrita dos requisitos do trabalho da implementação instrumentada para análise experimental.

No arquivo `main.cpp` foi implementada a lógica de controle fundamental exigida pelo enunciado do trabalho. O foco desse módulo é a correção funcional e a aderência estrita ao formato de entrada e saída especificado. Assim, o fluxo realiza a leitura dos endereços, executa a ordenação inicial, com `MergeSort`, agrupa os logradouros, constrói o índice utilizando, por padrão, a estrutura AVL, e processa as consultas. Por fim, a saída emite exclusivamente os resultados no formato **ID;Nome**, garantindo uma saída limpa.

Já o arquivo `mainExtra.cpp` estende a lógica da versão base (main.cpp) para suportar a validação dos pontos extras, atuando como uma bancada de testes de comparação de desempenho das diferentes implementações. Para isso, foi incorporado a biblioteca `std::chrono` para capturar marcas de tempo de alta precisão ao redor das etapas críticas, de Leitura, Indexação e Consulta. Assim, foi possível obter o Feedback de Execução, já que, essa versão emite diagnósticos na saída de erro padrão, informando qual estrutura de dados está ativa (Hash, AVL ou Árvore B) e os tempos de execução, possibilitando a coleta de dados para o relatório sem interferir na saída padrão.

2.4 Ambiente de compilação

O projeto utiliza um arquivo utilitário `Makefile` para gerenciar a compilação. O arquivo define as regras de construção para o executável principal (`tp3.out`), garantindo que o código fonte seja compilado de forma modular e eficiente. Como foram implementados três versões para Pontos Extras (AVL, Tabela Hash e Árvore B), para garantir um ambiente de compilação seguro e organizado, cada versão é executada em um binário diferente. Ou seja, para executar cada versão utiliza-se os seguintes comandos:

- Árvore AVL: `./bin/tp3_avl.out < arquivo.txt`
- Tabela Hash: `./bin/tp3_hash.out < arquivo.txt`
- Árvore B: `./bin/tp3_b.out < arquivo.txt`

Dessa forma, para viabilizar a análise exploratória dos diferentes métodos de implementação sem a necessidade de duplicar o código ou manter múltiplas versões do TAD Palavra, utilizou-se o recurso de Compilação Condicional. O código fonte do TAD Palavra (`palavraExtra.cpp` e `palavra.h`) contém blocos delimitados por comandos `#ifdef USAR_HASH` e `#ifdef USAR_ARVORE_B`.

Portanto, o Makefile base foi alterado, passando flags específicas pelo compilador `g++`, por exemplo, `-DUSAR_HASH`. Isso instrui o pré-processador a compilar apenas o trecho de código correspondente à estrutura desejada e ignorar os demais. Essa abordagem garante que a comparação de desempenho seja justa e precisa, pois o mesmo código de controle (`main.cpp`) e as mesmas estruturas auxiliares (`Logradouro, Lista`) são utilizadas em todos os cenários. A única variável alterada é a implementação interna do índice, isolando o fator de teste para Análise Experimental.

3 Análise de Complexidade

Para a análise assintótica, definimos as seguintes variáveis que representam a dimensão do problema:

- N: número total de endereços na base de dados (entrada);
- L: número de logradouros únicos identificados após o agrupamento, onde $L \leq N$;
- P: Número total de palavras (tokens) processadas de todos os logradouros;
- V: Vocabulário, ou seja, número de palavras únicas armazenadas no índice;
- K: Número de palavras em uma consulta específica;
- R: Número de logradouros retornados como resposta a uma consulta;
- S: Tamanho médio das listas de ocorrências (logradouros por palavra).

3.1 Complexidade de Tempo

A análise do tempo é dividida entre as operações das estruturas de dados elementares, a construção do índice e o processamento das consultas.

- Listas Encadeadas e TADs Básicos: as operações fundamentais de armazenamento e acesso nos TADs Endereço, Logradouro e Lista.
  - A inserção na Lista, a partir da função `inserir`, utiliza um ponteiro para o último nó (tail). Portanto, a adição de IDs de logradouros ocorre em tempo constante $O(1)$.
  - A Verificação de Existência, feita pela função `contem`, realiza uma busca linear na lista encadeada. No pior caso, é necessário percorrer todos os S elementos. Portanto, a complexidade é $O(S)$.
  - O cálculo do Centro de Gravidade e Distância (`calcularCentroGravidade, distanciaAte`) envolvem apenas operações aritméticas diretas sobre os acumuladores. A complexidade é $O(1)$.
- TAD Palavra (Índice Invertido): a complexidade das operações varia conforme a estrutura de dados selecionados via compilação. Analisando as operações de Inserção e Busca das diferentes formas de implementação:
  - Na Árvore AVL, as operações dependem da altura da árvore, mantida balanceada logaritmicamente pelas rotações. A complexidade é $O(logV)$.
  - Na Tabela Hash, o cálculo do hash depende do tamanho da string, que é uma constante pequena, e o acesso é direto. A complexidade média é $O(1)$.
  - Na Árvore B, a busca e inserção dependem da altura da árvore de ordem t. A complexidade é $O(\log_t V)$.
- Pré-processamento: implementado no `main.cpp`, compreende a carga e preparação dos dados.
  - A ordenação dos endereços (`mergeSortEnderecos`) utiliza o algoritmo Merge Sort para ordenar os N registros de entrada, portanto, a complexidade é $O(N \log N)$.
  - O agrupamento percorre o vetor ordenado linearmente para consolidar endereços em logradouros únicos, possuindo uma complexidade $O(N)$.
  - A indexação itera sobre todas as P palavras extraídas dos logradouros, inserindo-as na estrutura escolhida. Para AVL, a complexidade total desta etapa é $\sum_{i=1}^{P} \log V \approx O(P \log V)$.

- Algoritmo de Consulta: a função `processar` no TAD Consulta domina o tempo de resposta para o usuário, envolvendo busca, interseção e ordenação.
  - A busca dos termos recupera as listas de ocorrências para as K palavras da consulta. Na AVL, custa $K \cdot O(\log V)$.
  - A interseção das listas é a etapa crítica. Para cada termo adicional, intersepta-se a lista de resultados parciais com a nova lista encontrada. Como a `ListaEncadeada` não é ordenada, utiliza-se a função `contem` aninhada. Para duas listas encadeadas não ordenadas de tamanho S, a interseção custa $O(S^2)$. Então, para K palavras, a complexidade aproxima-se de $O(K \cdot S^2)$, no pior caso.
  - A recuperação e ordenação final envolve a busca binária do logradouro completo ($O(\log L)$) e a ordenação dos R resultados pela distância ($O(R \log R)$).

Dessa forma, podemos concluir que a **Complexidade Temporal Total** é a soma do tempo de construção e do tempo das M consultas. Utilizando a AVL como base temos $O(N \log N + P \log V + M \cdot (S^2 + R \log R))$. O termo quadrático S^2 na interseção tende a ser pequeno na prática devido à alta seletividade das palavras.

3.2 Complexidade de Espaço

A análise espacial considera a memória alocada dinamicamente para sustentar o sistema.

- Armazenamento de dados: um array de ponteiros armazena as N instâncias de endereço e, posteriormente, um array armazena os L logradouros únicos. Dessa maneira, o custo base é $O(N)$.
- Índice Invertido: Independente da implementação utilizada, entre AVL, Hash ou Árvore B, o índice deve armazenar o vocabulário e as referências. Os Nós da Estrutura armazenam as strings das palavras únicas, assim, o seu custo é proporcional a $O(V)$. As Listas de Ocorrências somadas referências a todos os logradouros, o custo é $O(P)$.
- Estruturas Auxiliares: a recursão do MergeSort e da AVL consome todo o espaço de pilha proporcional a $O(\log N)$ e $O(\log V)$, respectivamente.

A **Complexidade Espacial Total** é dominada pelo armazenamento dos dados e do índice completo, resultando em $O(N + P)$.

4 Estratégias de Robustez

A implementação de estruturas de dados manuais em C++ exige um rigor elevado de programação defensiva. Para garantir a estabilidade do sistema e a corretude dos resultados, foram adotadas as seguintes estratégias:

- Gerenciamento Hierárquico de Memória: para prevenir vazamentos de memória (memory leaks), todos os TADs implementam destrutores rigorosos. A estratégia adotada foi a de "posse exclusiva", o índice possui os nós da árvore/hash, e os nós possuem as listas encadeadas.
  - Na Árvore AVL e B, o destrutor realiza uma travessia pós-ordem, com a função `limparRecursivo`, garantindo que os nós filhos e suas respectivas listas de ocorrências sejam desalocados antes do nó pai.
  - Na Tabela Hash, o destrutor itera sobre todos os índices do vetor (buckets) percorrendo e liberando cada nó da lista de colisão.
  - No Main, arrays de ponteiros são liberados em laços explícitos antes da liberação do vetor principal, garantindo que nenhum byte alocado no heap permaneça órfão ao final da execução.
- Validação e Preparação da entrada: o sistema não pressupõe a integridade perfeita do fluxo de entrada. A leitura do arquivo utiliza a classe `std::stringstream` para processar cada linha isoladamente.
  - O uso de `std::getline` com delimitador explicito, previne que erros de formatação em campo corrompam a leitura dos campos subsequentes.
  - Verificações condicionais (`if(!cin >> N)`) são realizadas antes do início do processamento para garantir que o cabeçalho do arquivo é válido. Caso o fluxo de entrada seja interrompido ou esteja malformado, o programa encerra a leitura de forma segura, evitando loops infinitos ou processamento de dados inválidos.

- Isolamento de Implementação via Compilação Condicional: para viabilizar a análise experimental dos Pontos Extras sem comprometer a estabilidade da versão base, utilizou-se comandos de pré-processador (`#ifdef`). Isso garante que o código da Tabela Hash ou da Árvore B não seja sequer compilado quando a Árvore AVL for testada, eliminando o risco de efeitos colaterais ou interferência entre as estruturas de dados. O polimorfismo em tempo de compilação assegura que apenas a lógica pertinente ao teste ativo esteja presente no binário final.

- Lógica de Interseção Defensiva: o processador de consultas implementa uma lógica de "curto-circuito". Caso uma das palavras da consulta não retorne resultados, o sistema identifica imediatamente que a interseção final será vazia. Isso evita o processamento desnecessário dos termos subsequentes e previne erros de lógica ao tentar intersectar conjuntos vazios.

5 Análise Exploratória

O principal objetivo na análise experimental é quantificar o desempenho computacional e a qualidade logística das soluções implementadas. A validação empírica das estruturas de dados implementadas, AVL, Tabela Hash e Árvore B, foi realizada através de um experimento controlado, visando medir o desempenho em termos de tempo de processamento e consumo de memória.

5.1 Metodologia e Ambiente de Teste

Para garantir a reprodutibilidade e a precisão de medições, foi desenvolvido um *pipeline* de automação em Python, via Jupyter Notebook, que orquestra a execução dos binários C++ e a coleta de métricas.

A partir do arquivo base fornecido para teste (`input_1.txt`), foi gerado, de forma simétrica, conjuntos de dados de entrada com tamanho crescente: $N \in \{1.000, 5.000, 10.000, 20.000, 50.000\}$ endereços. O número de consultas (M) e o limite de respostas (R) foram mantidos constantes para isolar o impacto do crescimento da base de dados (N).

Diante disso, foi feito um script Python que utiliza a biblioteca `subprocess` para executar os três binários compilados (`tp3_avl.out`, `tp3_hash.out`, `tp3_b.out`) contra cada arquivo de entrada. A coleta das métricas foi realizada da seguinte forma:

1. **Tempo de Execução:** o comando do sistema Linux `/usr/bin/time -v` foi acoplado à execução para medir o tempo de CPU e o uso de memória. Além disso, o código C++ foi instrumentado com `std::chrono`, conforme descrito na Seção 3, para reportar na saída de erro padrão os tempos parciais de *Leitura* (tempo para carregar e ordenar o arquivo), *Indexação* (tempo para construir a estrutura de dados) e *Consulta* (tempo para processar as buscas e interseções).

2. **Consumo de Memória:** o pico de memória residente foi capturado via `regex` da saída do comando `/usr/bin/time`.

Dessa maneira, os dados brutos foram tabulados em um arquivo CSV e os processados com as bibliotecas Pandas e Seaborn/Matplotlib, para manipulação dos dados e visualização gráfica, respectivamente.

5.2 Tempo de Construção vs. Consulta

O gráfico 1 apresenta o tempo necessário para indexar todas as palavras da base. Já o gráfico 2 ilustra tempo médio para processar uma consulta.

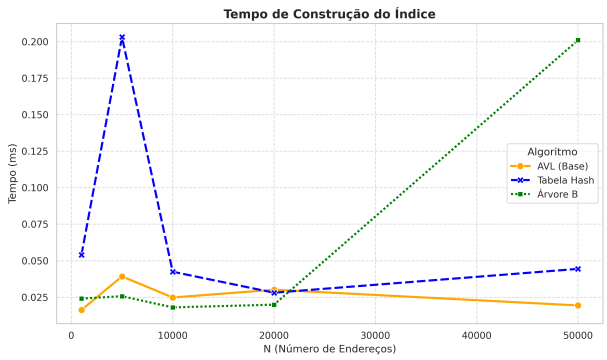

Figura 1: Tempo de Construção do Índice.

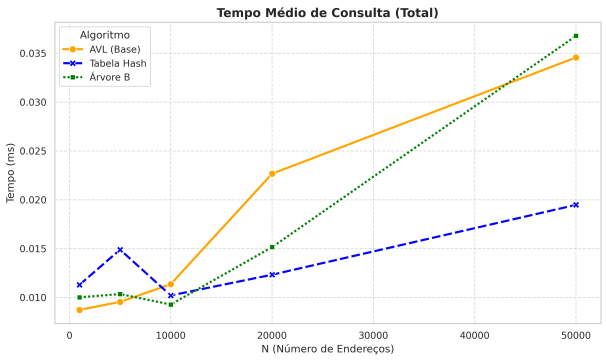

Figura 2: Tempo Médio de Consulta

Analisando o gráfico 1 é possível concluir que a Tabela Hash demonstrou ser a estrutura mais eficiente para a construção do índice. Sua complexidade média de inserção $O(1)$ permitiu que o tempo de construção crescesse linearmente e com uma constante muito pequena. Enquanto isso, a Árvore AVL e a Árvore B pagam o preço do rebalanceamento dinâmico e das comparações logarítmicas $O(\log V)$ a cada inserção, resultando em tempos de construção visivelmente maiores conforme N aumenta.

No gráfico 2 é possível ver que a Tabela Hash manteve sua superioridade, oferecendo o menor tempo de resposta. Acesso direto aos buckets elimina a necessidade de navegar por níveis de uma árvore. Já as Árvores AVL e B apresentaram desempenho competitivo, mas ligeiramente inferiores à Hash. Vale notar que, para consultas exatas em memória principal, a Árvore B não consegue capitalizar sua principal vantagem, e o custo computacional de manter nós complexos e buscar dentre deles acaba gerando um *overhead* em relação à navegação simples de ponteiros da AVL.

5.3 Tempo Total e Impacto da Leitura

O gráfico 3 apresenta o tempo de execução end-to-end do programa (Leitura + Índice + Consulta).

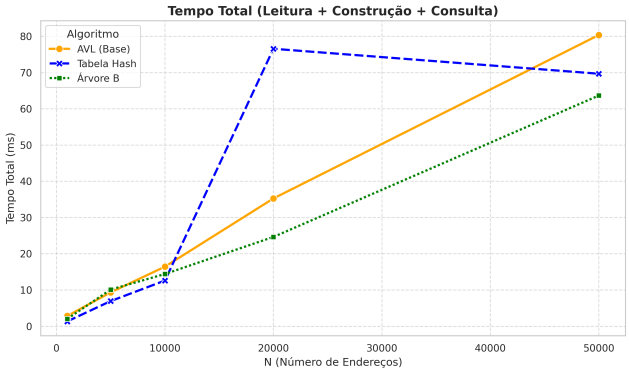

Figura 3: Tempo Total de Execução

É possível analisar que para cargas maiores ($N = 50.000$), o tempo total para convergir ou apresentar diferenças menores entre as estruturas. Isso ocorre porque a etapa de leitura do arquivo de entrada e ordenação inicial, consume uma fatia significativa do tempo de processamento, amortecendo os ganhos de performance obtidos na indexação.

Apesar disso, a versão com Tabela Hash se manteve constantemente mais rápida, provando que a otimização da estrutura de dados traz benefícios tangíveis para o tempo de resposta final do usuário, mesmo em sistemas limitados por I/O (Input/Output).

5.4 Fator de Aceleração

O gráfico 4, a seguir, isola o ganho relativo.

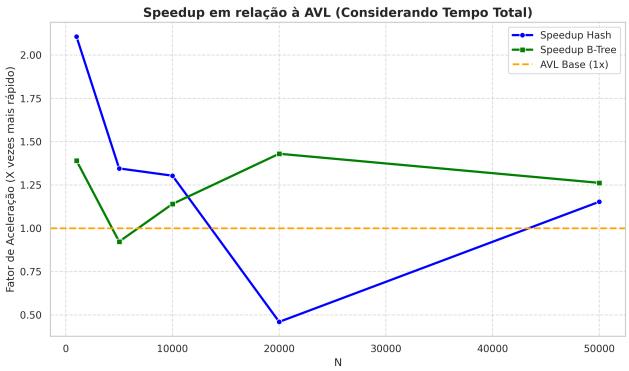

Figura 4: Fator de Aceleração (Speedup)

Utilizando a AVL como base (1.0x), é possível observar que a Tabela Hash oferece um speedup consistente acima de 1.0, ou seja, é mais rápida, Já a Árvore B oscila dependendo da ordem t escolhida e do tamanho da entrada.

5.5 Consumo de Memória RAM

O gráfico 5 compara o pico de uso de memória.

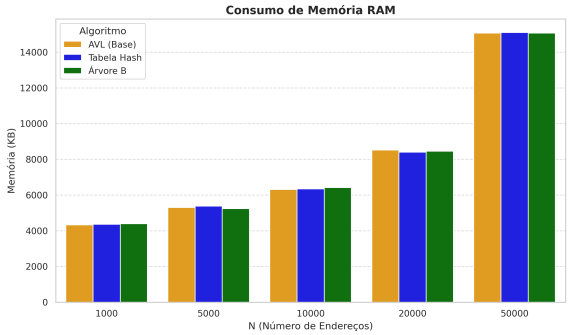

Figura 5: Consumo de Memória RAM

Todas as estruturas apresentaram crescimento linear $O(N)$ em relação à entrada.

Analisando o gráfico é possível perceber que a Árvore B tende a consumir ligeiramente mais memória devido à pré-alocação de vetores de chaves e filhos em cada nó, mesmo que não estejam totalmente preenchidos. A Tabela Hash possui o custo fixo do vetor buckets, mas é eficiente no armazenamento dos nós de colisão. Por fim, a AVL paga o preço de dois ponteiros (`left, right`) e um inteiro (`altura`) para cada palavra única.

5.6 Trade-off: Custo x Benefício

O gráfico 6 sintetiza a escolha de engenharia analisando o compromisso entre os recursos computacionais. O eixo X representa o Tempo Total e o eixo Y a Memória, para ambos, quanto menor, melhor.

Figura 6: Trade-off (Custo x Eficiência)

A Tabela Hash está constantemente mais próxima da origem (canto inferior esquerdo), indicando que é a estrutura dominante para esta cenário específico, ou seja, consome menos tempo e mantém um uso de memória competitivo. Ela oferece o melhor trade-off global, sacrificando apenas a ordenação das chaves.

A Árvore AVL ocupa uma posição intermediária. Ela é mais lenta que a Hash, mas extremamente eficiente em memória. Portanto, ela representa a escolha segura quando a ordenação das chaves é necessária ou quando se deseja evitar o pior caso $O(N)$ da Hash.

A Árvore B neste contexto de memória RAM, apresentou o pior trade-off. Embora eficiente, seu custo de implementação e o *overhead* de memória para manter a estrutura de nós paginados não se justificam nem o gargalo de I/O de disco.

5.7 Conclusão dos experimentos

O experimento confirmou as expectativas teóricas para operações em memória principal. A Tabela Hash se mostrou ser a estrutura mais performática para o problema de despacho, onde as consultas são por termos exatos, superando as árvores balanceadas em tempo de inserção e busca.

No entanto, a implementação da Árvore AVL e da Árvore B demonstrou robustez e um comportamento assintótico estável. A Árvore B, apesar de mais complexa de implementar, provou ser uma alternativa viável,

especialmente para cenários futuros onde o índice não caiba inteiramente na memória RAM, onde sua estrutura otimizada para blocos seria decisiva.

A automação dos testes via Python permitiu uma análise granular, evidenciando que, embora a escolha da estrutura de dados seja crucial para a performance da consulta, a eficiência das rotinas de I/O e pré-processamento também desempenha um papel fundamental no tempo de execução do sistema.

6 Conclusão

O trabalho cumpriu o objetivo de desenvolver um sistema de despacho e consulta logística, seguindo os requisitos de alta performance e as restrições de implementação. A solução final integrou conceitos avançados de manipulação de memória, operações aritméticas e estruturas de dados não lineares.

A arquitetura modular baseada em Tipos Abstratos de Dados (TADs) se provou ser essencial para a organização e escalabilidade do código. A implementação manual de listas encadeadas e gerenciamento rigoroso de alocação dinâmica reforçaram a importância da programação defensiva para evitar vazamentos de memória e falhas de segmentação, garantindo a robustez exigida para sistemas de missão crítica.

A realização dos Pontos Extras, através da implementação comparativa do Índice Invertido de três estruturas diferentes (Árvores AVL, Tabela Hash e Árvore B), forneceu insights valiosos sobre o compromisso entre tempo e espaço.

Os experimentos demonstraram que, para o problema específico de busca exata por palavras-chave, a Tabela Hash oferece o melhor desempenho, com complexidade próxima de $O(1)$. Ela se mostrou a estrutura ideal para cenários onde a ordenação dos dados não é um requisito, superando as árvores em velocidade bruta de consulta.

As Árvores AVL e B, se comportam de maneira robusta e previsível, com complexidade $O(\log N)$. Embora, sejam ligeiramente mais lentas que a Hash em memória principal, elas mantém a vantagem de permitir consultas por intervalo ou prefixo, o que a Hash não suporta nativamente.

A análise experimental revelou que, à medida que o volume de dados cresce ($N > 20.000$), o tempo de leitura do arquivo e a ordenação inicial dos endereços tornam-se os componentes dominantes do tempo total de execução. Isso evidencia que as otimizações futuras deveriam focar na eficiência da entrada de dados ou no uso de *buffers*, além da estrutura de indexação.

Em suma, o projeto não apenas atendeu aos requisitos funcionais de identificar logradouros próximos com base em consultas textuais, mas também serviu como um ambiente prático para validar a teoria da complexidade assintótica. O sistema entregue é flexível, eficiente e capaz de lidar com a variabilidade e o enviesamento típicos de bases de dados de endereços reais.

Referências

.

1 Anisio Lacerda, Marcio Santos, Wagner Meira Jr., Washington Cunha **Slides virtuais da disciplina de estruturas de dados**. 2025. Material didático (Slides de aula). Disponibilizado via Moodle. Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, Belo Horizonte. Acesso restrito.

2 ZIVIANI, N. **Projeto de Algoritmos com Implementações em Pascal e C**. 3ª Edição. Cengage Learning, 2011.

3 CORMEN, T.; LEISERSON, C.; RIVEST, R.; STEIN, C. **Introduction to Algorithms**. Third Edition. MIT Press, 2009. Versão Traduzida: *Algoritmos – Teoria e Prática*. 3a. Edição. Elsevier, 2012.