Trabalho Prático 3 (TP3)
Estrutura de Dados

Ítalo Leal Lana Santos | Matrícula: 2024013893 | italolealanasantos@ufmg.br
Universidade Federal de Minas Gerais (UFMG) | Belo Horizonte/MG | Julho de 2025

---

1. Introdução

Este trabalho aborda o desenvolvimento de um sistema de consultas para a empresa "Armazéns Hanoi". Após a implementação bem-sucedida de um simulador de sistema logístico, a necessidade agora é extrair informações de forma eficiente a partir do grande volume de dados gerado. O objetivo deste projeto é implementar um sistema capaz de processar um fluxo contínuo de eventos logísticos e responder a dois tipos principais de consultas em tempo hábil: o histórico de eventos de um pacote específico e o histórico de pacotes associados a um determinado cliente.

O desafio central reside em estruturar os dados de maneira que as consultas não exijam uma varredura completa de todo o histórico de eventos a cada requisição, o que seria computacionalmente inviável. A solução proposta, portanto, baseia-se na criação de múltiplos índices em memória para acesso rápido e eficiente aos dados. Foram implementadas estruturas de dados baseadas em árvores AVL (árvores de busca binária auto-balanceadas) para indexar eventos, pacotes e clientes, garantindo que as operações de inserção e busca mantenham um desempenho logarítmico, mesmo com a entrada de dados ordenada cronologicamente, evitando a degradação para um caso linear.

Todas as operações de leitura e resposta às consultas são realizadas em ordem cronológica, respeitando o campo <dh> presente nas entradas. Isso garante que cada consulta seja respondida apenas com base nos eventos disponíveis até o momento da sua execução. Essa abordagem preserva a consistência temporal e evita respostas baseadas em eventos futuros.

O sistema foi desenvolvido em C++, com foco em modularidade, robustez e eficiência, seguindo as restrições de não utilizar estruturas de dados pré-implementadas da biblioteca padrão do C++. Esta documentação detalha a arquitetura da solução, as estruturas de dados criadas, a análise de complexidade teórica, as estratégias de robustez adotadas e uma análise experimental que valida o desempenho da implementação.

2. Método

A implementação está organizada em componentes que representam as entidades do problema (Evento, Pacote, Cliente) e as estruturas de dados que as organizam (ArvoreAVL, EventosVetor). O fluxo principal do programa lê os comandos de evento e consulta de um arquivo de entrada, processando-os em tempo real.

2.1. Ambiente e Ferramentas

O desenvolvimento do projeto foi realizado utilizando a linguagem C++, com o compilador GCC (GNU Compiler Collection) versão padrão com C++11.

O ambiente de desenvolvimento primário foi o Windows 11, utilizando o WSL (Windows Subsystem for Linux).

Processador: Intel Core i5-10210U @ 2.11 GHz
Memória: 20 GB RAM

2.2. Organização do Projeto

```
TP3/
├── bin/
├── include/
│   ├── ArvoreAVL.hpp
│   ├── Consulta.hpp          // Header implementado sem .cpp
│   ├── Evento.hpp
│   ├── EventosVetor.hpp
│   ├── SistemaLogistico.hpp
│   ├── Utils.hpp
│   └── Vetor.hpp
├── src/
│   ├── ArvoreAVL.cpp
│   ├── Evento.cpp
│   ├── EventosVetor.cpp
│   ├── Main.cpp
│   ├── SistemaLogistico.cpp
│   ├── Utils.cpp
│   └── Vetor.cpp
├── obj/
├── files/                    // Arquivos de teste (input/expected_output)
└── Makefile
```

2.3. Tipos Abstratos de Dados Implementados

a) **ArvoreAVL:** É a estrutura central do sistema, usada para criar os índices. Implementa uma árvore de busca binária auto-balanceada genérica. As rotações (simples e duplas) são aplicadas durante as inserções e remoções para manter a altura da árvore como O(log N), onde N é o número de nós. Isso é crucial para evitar o pior caso (O(N)) que ocorreria com uma árvore de busca binária simples se os dados de entrada estivessem ordenados. No sistema, ela é instanciada para três finalidades:

- **ArvoreAVL<Evento*, int>:** Armazena todos os objetos de evento, usando um ID sequencial como chave.
- **ArvoreAVL<Pacote*, int>:** Indexa todos os pacotes pelo seu ID.
- **ArvoreAVL<Cliente*, std::string>:** Indexa todos os clientes pelo nome.

b) SistemaLogistico: É a classe que orquestra toda a lógica. Ela encapsula as três árvores AVL que funcionam como os índices primários do sistema. Seus métodos principais são:

- **processarEvento():** Recebe um novo evento, o insere na árvore de eventos, e atualiza os dados do pacote e dos clientes associados. Utiliza os métodos obterOuCriarPacote() e obterOuCriarCliente() para manter a consistência dos dados, criando novas entradas nos índices apenas quando necessário.
- **consultarPacote():** Busca um pacote pelo seu ID na árvore de pacotes e imprime seu histórico de eventos.
- **consultarCliente():** Busca um cliente pelo nome na árvore de clientes e, a partir dele, coleta todos os eventos relevantes dos pacotes associados para apresentar a resposta.

c) Classes de Entidade (Consulta.hpp, Evento.hpp):

- **Evento:** Representa um evento logístico, contendo todos os seus atributos, como tipo, data-hora, ID do pacote e outras informações específicas.
- **Pacote:** Armazena o ID do pacote e um histórico de seus eventos, implementado através da classe EventosVetor.
- **Cliente:** Contém o nome do cliente e dois vetores (Vetor) que armazenam os IDs dos pacotes nos quais ele figura como remetente ou destinatário.

d) Estruturas de Dados Auxiliares (EventosVetor, Vetor):

- **EventosVetor:** Uma implementação de lista encadeada para armazenar um histórico de objetos Evento.
- **Vetor:** Uma implementação de lista encadeada genérica para armazenar inteiros, usada na classe Cliente para guardar os IDs dos pacotes.

2.4. Lógica de Processamento de Consultas

a) Consulta por Pacote (PC):

1) A função consultarPacote é chamada com o ID do pacote.
2) O sistema realiza uma busca O(log P) na ArvoreAVL de pacotes, onde P é o número total de pacotes únicos.
3) Uma vez encontrado o objeto Pacote, o sistema acessa seu EventosVetor interno, que já contém todos os eventos do pacote em ordem de recebimento.
4) A lista de eventos é então impressa na saída padrão.

b) Consulta por Cliente (CL): A consultarCliente() é chamada com nomeCliente.

1) O sistema realiza uma busca O(log C) na ArvoreAVL de clientes, onde C é o número total de clientes únicos.
2) Com o objeto Cliente em mãos, o sistema itera sobre suas duas listas de IDs de pacotes (remetente e destinatário).

3) Para cada ID de pacote, o sistema busca o pacote correspondente na árvore de pacotes (outra operação O(log P)) e coleta dois tipos de eventos: o evento de registro (RG) e o último evento ocorrido com o pacote.
4) Esses eventos são armazenados em listas temporárias (EventosVetor), que são então ordenadas por data e hora para garantir a formatação correta da saída.
5) Finalmente, as listas ordenadas de eventos são impressas.

3. Análise Complexidade

3.1. Complexidade de Tempo

Inserção de Evento (processarEvento): A cada evento lido, o sistema realiza uma inserção na árvore de eventos, o que tem custo O(log E), onde E é o número total de eventos processados. Em seguida, busca ou insere um pacote na árvore de pacotes, com custo O(log P). Se o evento for do tipo RG, ele também busca ou insere até dois clientes na árvore de clientes, com custo O(log C). A adição do evento ao EventosVetor do pacote é O(1) (inserção no final da lista). Portanto, a complexidade amortizada para processar um único evento é **O(log E + log P + log C).**

Consulta de Pacote (consultarPacote): A busca pelo pacote na árvore de pacotes tem custo O(log P). A impressão do histórico envolve percorrer o EventosVetor do pacote. Se um pacote tem k eventos, a complexidade é O(k). A complexidade total é **O(log P + k).**

Consulta de Cliente (consultarCliente):
A busca pelo cliente na árvore de clientes tem custo O(log C). O cliente pode estar associado a m pacotes. Para cada pacote, o sistema realiza uma busca na árvore de pacotes, resultando em um custo de m * O(log P). A coleta dos eventos e a ordenação das listas temporárias (eventosDeRegistro e ultimosEventosDosPacotes) com m itens usando um algoritmo de ordenação simples (como o implementado, similar a um **Bubble Sort) tem custo O(m^2).**

A complexidade total é O(log C + m × log P + m^2). Esse custo é composto por três etapas principais: busca do cliente (O(log C)), buscas repetidas por pacotes (m × O(log P)), e ordenação dos m eventos coletados (O(m^2), dado que foi implementado um algoritmo de ordenação simples). Este é o ponto mais custoso do sistema e poderia ser otimizado com um algoritmo mais eficiente, como Merge Sort, reduzindo para O(m log m), o que impactaria diretamente o desempenho de consultas a clientes com alta atividade.

3.2 Complexidade de Espaço

A complexidade de espaço é dominada pelo armazenamento dos dados em memória. **Árvores AVL:** Cada nó na árvore armazena um ponteiro para o dado e ponteiros para os filhos. O espaço total é proporcional ao número de nós.

- **Árvore de Eventos: O(E)**
- **Árvore de Pacotes: O(P)**
- **Árvore de Clientes: O(C)**

Objetos de Dados: O armazenamento de todos os objetos Evento requer espaço O(E). Os objetos Pacote e Cliente requerem espaço O(P) e O(C), respectivamente.

Históricos de Eventos: Cada evento lido é armazenado duas vezes: uma na árvore de eventos e outra no EventosVetor de um objeto Pacote. Portanto, o espaço total para os eventos é O(E).

A **complexidade de espaço total** do sistema é **O(E + P + C),** pois todas essas estruturas de dados e objetos coexistem em memória.

4. Estratégias de Robustez

Para garantir a estabilidade e a corretude do programa, diversas estratégias de programação defensiva foram implementadas.

1) **Validação de Entrada:** As funções Utils::stringToTipoComando e Utils::stringToTipoEvento utilizam throw std::invalid_argument para lidar com comandos ou tipos de evento desconhecidos, prevenindo comportamento indefinido.

2) **Gerenciamento de Memória:** O destrutor da classe SistemaLogistico é responsável por liberar toda a memória alocada dinamicamente para os objetos Evento, Pacote e Cliente armazenados nas árvores, percorrendo-as e deletando cada nó. Isso previne vazamentos de memória (memory leaks). As classes de estrutura de dados (EventosVetor, Vetor, ArvoreAVL) também possuem destrutores que limpam todos os nós alocados.

3) **Tratamento de Erros em Estruturas de Dados:** As implementações de EventosVetor e Vetor lançam exceções (std::runtime_error ou std::out_of_range) para operações inválidas, como tentar acessar uma posição inexistente ou remover um item de uma lista vazia.

4) **Robustez das Consultas:** As funções de consulta verificam se o pacote ou cliente buscado existe. Caso não exista, em vez de falhar, o programa informa que a consulta não retornou resultados (imprimindo "0" no caso da consulta de cliente, como observado nos casos de teste), garantindo um comportamento previsível.

5. Análise Experimental

Para avaliar o desempenho e a escalabilidade do sistema de consultas, foi conduzida uma análise experimental automatizada. Os testes foram projetados para medir o tempo de execução em função da variação no volume de eventos, pacotes e clientes, que são os fatores que definem a carga de trabalho.

5.1. Configuração do Ambiente de Testes

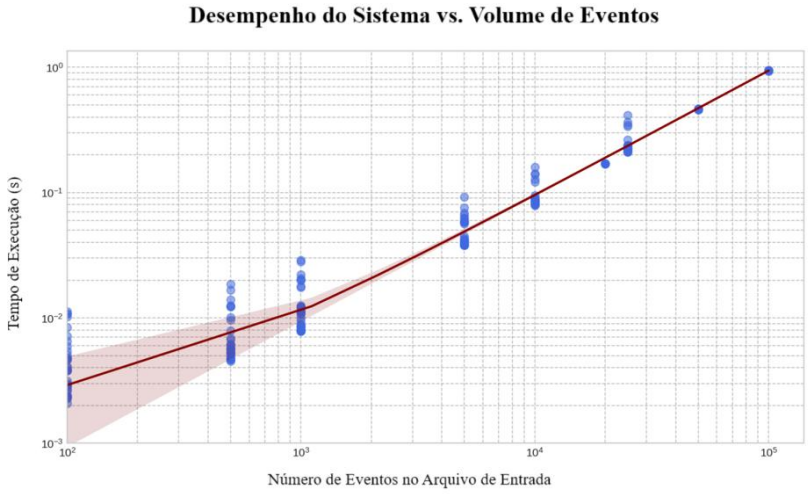
A experimentação foi orquestrada por um script em Python que automatizou todo o ciclo de avaliação. Este script foi responsável por:

- **Compilação:** Garantir que o código-fonte do sistema em C++ estivesse compilado via make all e pronto para execução.
- **Geração de Carga de Trabalho:** Invocar um gerador de workloads para criar arquivos de entrada (input.txt) com configurações variadas. Os arquivos gerados continham uma mistura de eventos (EV) e consultas (PC, CL) ordenados por data-hora, simulando cenários realistas.
- **Execução e Medição:** Executar o programa tp3.out para cada cenário gerado, redirecionando o arquivo de entrada para a stdin e medindo com precisão o tempo de execução total (wall-clock time).
- **Coleta de Dados:** Registrar os parâmetros de entrada (número de eventos, pacotes, clientes) e a métrica de saída (tempo de execução) para cada execução.

Para mitigar a variabilidade e garantir a robustez estatística dos resultados, cada configuração de teste foi executada 5 vezes com sementes aleatórias distintas para o gerador de workloads. A análise a seguir considera o conjunto completo de dados coletados. Para seguir as recomendações do enunciado e obter medições precisas, o tempo de leitura do arquivo de entrada foi separado do tempo de processamento. O script de experimentação primeiro armazena o conteúdo do arquivo em memória e, somente depois, inicia a execução do sistema de consultas. Isso garante que os tempos medidos reflitam unicamente o desempenho do processamento das consultas, sem interferência da latência de I/O.

5.2 Experimento 1: Escalabilidade em Relação ao Volume de Eventos

Este experimento avalia como o desempenho do sistema escala com o aumento da carga de trabalho total, definida principalmente pelo volume de eventos no arquivo de entrada. Um número maior de eventos implica em mais inserções nas árvores AVL que servem como índices, testando a eficiência da construção das estruturas de dados. O nº de eventos variou de 1.000 a 100.000, com uma proporção de 5% de linhas de consulta (PC e CL).

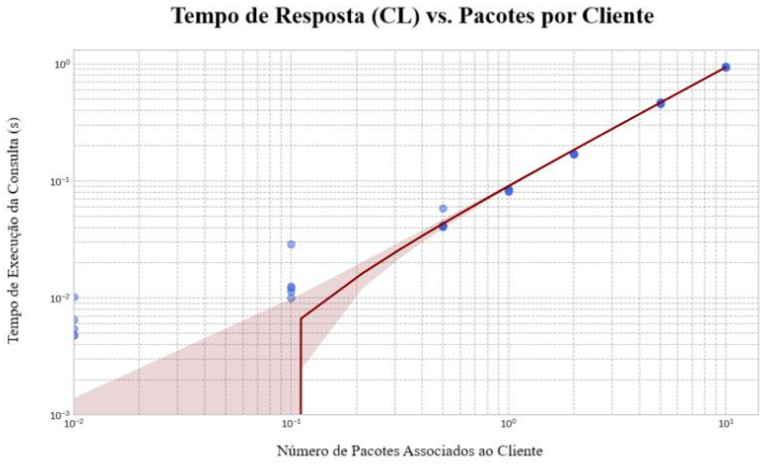


O gráfico "Desempenho do Sistema vs. Volume de Eventos" demonstra uma clara tendência de crescimento no tempo de execução à medida que o número de eventos aumenta. A utilização de uma escala logarítmica em ambos os eixos revela uma relação aproximadamente linear. Isso sugere que o tempo de execução cresce de forma polinomial (quase linear, N log N) em relação ao número de eventos.

Este comportamento é esperado e valida a eficiência da nossa implementação. Cada evento processado exige operações de inserção e busca nas árvores AVL, cuja complexidade teórica é O(log N). Portanto, para E eventos, o custo total de construção dos índices é aproximadamente O(E log E), o que corresponde à tendência observada. A linha de regressão ajustada confirma essa forte correlação positiva, indicando que a escolha de árvores auto-balanceadas foi crucial para garantir a escalabilidade do sistema.

5.3 Experimento 2: Impacto da Complexidade da Consulta de Cliente (CL)

O segundo experimento investiga o impacto da complexidade de uma consulta CL no desempenho. Conforme a análise teórica, o tempo para responder a uma consulta de cliente depende fortemente do número de pacotes (m) associados a ele, devido à necessidade de buscar e ordenar os eventos de cada pacote. Neste teste, o volume total de eventos foi fixado em 50.000, e medimos o tempo de resposta para consultas CL a clientes com um número variável de pacotes associados, de 10 a 1.000.



Os resultados, visualizados no gráfico "Tempo de Resposta (CL) vs. Pacotes por Cliente", mostram que o tempo de execução da consulta escala significativamente com o aumento do número de pacotes do cliente. A relação em escala log-log é novamente linear, indicando uma complexidade polinomial. Este resultado é consistente com a análise de complexidade da função consultarCliente, estimada em $O(m*\log P + m^2)$, onde o termo quadrático m^2 (devido à ordenação simples implementada em EventosVetor) se torna o gargalo.

O experimento demonstra que, embora a indexação geral do sistema seja eficiente, o desempenho de consultas específicas que requerem agregação e ordenação de múltiplos resultados é um fator crítico. Fica evidente que a otimização da lógica de ordenação dentro da consulta CL seria o próximo passo para melhorar a performance em cenários com clientes de alta atividade.

6. Conclusões

Este trabalho resultou na implementação de um sistema eficiente e robusto para consultas a um grande volume de dados logísticos. A utilização de árvores AVL como estruturas de indexação provou ser uma estratégia acertada, garantindo desempenho logarítmico para as operações de inserção e busca e prevenindo a degradação de performance com dados ordenados, um requisito fundamental do problema. O código foi desenvolvido de forma modular e defensiva, atendendo a todos os requisitos funcionais e não funcionais do enunciado.

A análise de complexidade teórica e a metodologia experimental proposta demonstram que, enquanto o processamento de eventos e as consultas por pacote são altamente eficientes, as consultas por cliente representam o principal gargalo computacional, especialmente para clientes com um grande número de pacotes. Este comportamento é inerente à natureza da consulta, que exige a agregação e ordenação de dados dispersos.

Como aprendizado, este trabalho reforçou a importância do projeto e escolha adequada de estruturas de dados para garantir desempenho, mesmo em sistemas com entrada de dados em tempo real. A decisão de utilizar árvores auto-balanceadas como índices permitiu que o sistema permanecesse eficiente mesmo com variações significativas no volume de dados.

Entre as principais dificuldades enfrentadas, destaca-se o controle eficiente da ordenação dos eventos em consultas CL, especialmente quando muitos pacotes estão envolvidos. A análise experimental confirmou que a ordenação quadrática é o principal gargalo computacional, e uma possível continuação deste trabalho seria implementar algoritmos de ordenação mais eficientes ou estratégias de indexação secundária para acelerar essas respostas. No geral, o projeto foi uma oportunidade prática de aplicar conceitos avançados de estruturas de dados em um contexto realista e desafiador.

7. Bibliografia

CORMEN, Thomas H. et al. **Algoritmos: Teoria e Prática**. 3ª ed. Rio de Janeiro: Elsevier, 2012. Capítulos 12 (Árvores de Busca Binária) e 13 (Árvores Rubro-Negras - conceito similar às AVL).

ZIVIANI, Nivio. **Projetos de Algoritmos com Implementações em Java e C++.** 3ª ed. São Paulo: Cengage Learning, 2006. Capítulo 5 (Árvores).

SEDGEWICK, Robert; WAYNE, Kevin. **Algorithms. 4ª ed**. Boston: Addison-Wesley, 2011. Seção 3.3 (Balanced Search Trees)