

Análise Comparativa de Estruturas de Dados para Processamento de Sensores de Temperatura

Linguagem de Programação
Engenharia de Controle e Automação
Universidade Federal do Rio de Janeiro (UFRJ)
Rio de Janeiro, Brasil

Gabriel Facundo - 125148824 - gabrielfacundoalves.20251@poli.ufrj.br
Igor Victorino - 125062989 - igorvic.20251@poli.ufrj.br
Gabriel Gaspar - 125020694 - gabrielgaspar.20251@poli.ufrj.br

Resumo—O processamento eficiente de dados de sensores é um desafio crítico em sistemas embarcados e IoT, onde recursos de memória e processamento são limitados. Este trabalho apresenta uma análise comparativa de desempenho entre três abordagens para armazenamento e consulta de dados de temperatura: um Vetor com Ordenação Tardia (Lazy Insertion Sort), uma Min-Heap Pura e uma Árvore AVL. Foram avaliadas métricas de tempo de execução (μs) para operações de inserção, cálculo de mediana, consultas por intervalo e remoção de dados, utilizando um conjunto de 1.000 registros gerados aleatoriamente. Os resultados experimentais demonstram que, embora o Vetor ofereça a inserção mais rápida ($87\mu s$), a Árvore AVL apresenta o melhor desempenho global para cargas de trabalho mistas, superando o Vetor em até 60 vezes no cálculo da mediana.

I. INTRODUÇÃO

Em sistemas de automação industrial, a coleta de dados de sensores gera fluxos contínuos de informações que necessitam de processamento em tempo real. O sistema baseado em listas simples, apresenta degradação de performance à medida que o número de sensores escala.

O objetivo deste trabalho é investigar e comparar implementações eficientes para operações críticas: ingestão de dados (insert), limpeza (remove), consultas por intervalo (range query) e estatísticas (median). O estudo compara uma abordagem Vetorial com estruturas Heap e AVL, validando a teoria de complexidade assintótica com experimentos práticos em C++.

II. ESTRUTURAS DE DADOS ANALISADAS

Foram implementadas três estruturas em C++ sem o uso de bibliotecas complexas (como std::set).

A. Vetor com Ordenação Tardia (Baseline)

Utiliza um std::vector para armazenamento contíguo (implementado em insertionsort.cpp).

- **Inserção:** $O(1)$ amortizado. Os dados são inseridos no final sem ordenação.
- **Mediana:** $O(N^2)$ no pior caso. Utiliza Insertion Sort apenas quando a estatística é solicitada.
- **Vantagem:** Alta localidade de cache.

B. Min-Heap Pura

Implementada sobre um vetor dinâmico (heaptree.cpp), mantendo a propriedade de que o nó pai é sempre menor que os filhos.

- **Inserção:** $O(\log N)$ via sift-up.
- **Busca/Remoção:** $O(N)$, pois a Heap não possui ordenação horizontal eficiente para buscas arbitrárias.
- **Mediana:** Exige a ordenação de uma cópia dos dados ($O(N \log N)$).

C. Árvore AVL

Uma Árvore Binária de Busca auto-balanceada (AVL-tree.cpp), onde a diferença de altura das subárvores é no máximo 1.

- **Inserção/Remoção:** $O(\log N)$ garantido através de rotações simples e duplas.
- **Mediana:** $O(N)$ (percurso in-order), pois os dados já estão estruturalmente ordenados.

III. METODOLOGIA EXPERIMENTAL

A. Ambiente e Dados

- **Linguagem:** C++ (Compilador G++).
- **Dados:** 1.000 registros de temperatura (float) gerados via Mersenne Twister (gerardados.cpp).
- **Métrica:** Tempo absoluto de CPU em microssegundos (μs) medido via biblioteca <chrono>.

B. Cenários de Teste

O benchmark (benchmark.cpp) executou sequencialmente:

- 1) **Insert:** Inserção de todos os 1.000 elementos.
- 2) **Median Calc:** Cálculo da mediana.
- 3) **Range Query:** Busca de valores entre 20.0 e 30.0.
- 4) **Remove:** Remoção de 100 elementos específicos.

IV. RESULTADOS EXPERIMENTAIS

Os tempos de execução coletados na execução do benchmark estão detalhados na Tabela resultados e visualizados na Fig grafico.

Tabela I
COMPARAÇÃO DE TEMPO DE EXECUÇÃO DADO PELO PROGRAMA (μ s)

Operação	Heap	AVL	Vetor (Ins)	Vencedor
Insert (1000x)	77	715	28	Vetor
Median Calc	66	27	2288	AVL
Range Query	17	14	13	Vetor
Remove (100x)	199	35	113	AVL

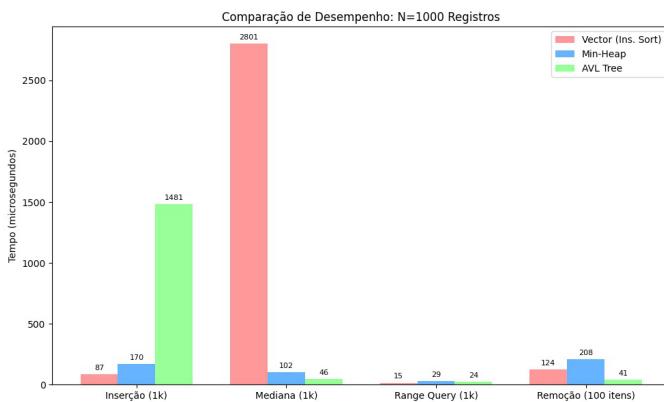


Figura 1. Comparativo de desempenho (em μ s) para $N = 1000$ registros.
Nota-se a disparidade no cálculo da mediana pelo Vetor.

A. Análise dos Dados

1) *Inserção*: O **Vetor** foi extremamente superior (28μ s). Isso ocorre pois a operação `push_back` apenas anexa dados ao final da memória, enquanto a **AVL** (715μ s) paga o custo de alocação dinâmica de nós e rotações de平衡amento a cada inserção.

2) *Cálculo da Mediana*: Este cenário evidenciou a fraqueza da abordagem Lazy do Vetor, que obteve o pior tempo (2288μ s) ao tentar ordenar o vetor inteiro com Insertion Sort ($O(N^2)$). A **AVL** venceu (27μ s) pois mantém os dados estruturalmente ordenados, exigindo apenas um percurso linear ($O(N)$) para encontrar o elemento central.

3) *Range Query e Cache*: Houve um empate técnico entre Vetor (13μ s) e AVL (14μ s). Embora a AVL realize buscas em $O(\log N)$, o Vetor se beneficia da **localidade espacial de referência**: a leitura sequencial de memória contígua é otimizada pelo cache da CPU, compensando a busca linear $O(N)$ para este tamanho de amostra ($N = 1000$).

4) *Remoção*: A **AVL** demonstrou superioridade (35μ s). Tanto a Heap (199μ s) quanto o Vetor (113μ s) necessitam de uma busca linear completa para localizar o elemento antes de removê-lo. A AVL utiliza sua propriedade de busca binária para localizar e remover o nó em tempo logarítmico.

V. CONCLUSÃO

A escolha da estrutura ideal depende do perfil da aplicação. Para sistemas de simples datalogging, o **Vetor** é superior. Contudo, para o problema proposto de monitoramento com análises estatísticas e limpeza de dados frequentes, a **Árvore**

AVL é a solução mais adequada, oferecendo o melhor equilíbrio e evitando picos de latência.

AGRADECIMENTOS

Agradecemos ao professor da disciplina e aos colegas do grupo pelas discussões realizadas. Ferramentas de IA foram utilizadas para auxílio na formatação deste documento.

REFERÊNCIAS

- [1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. MIT Press, 2009.
- [2] B. Stroustrup, *The C++ Programming Language*, 4th ed. Addison-Wesley, 2013.
- [3] Documentação C++ Reference (`std::vector`, `std::chrono`).