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 insetionsort.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 (μs)

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**

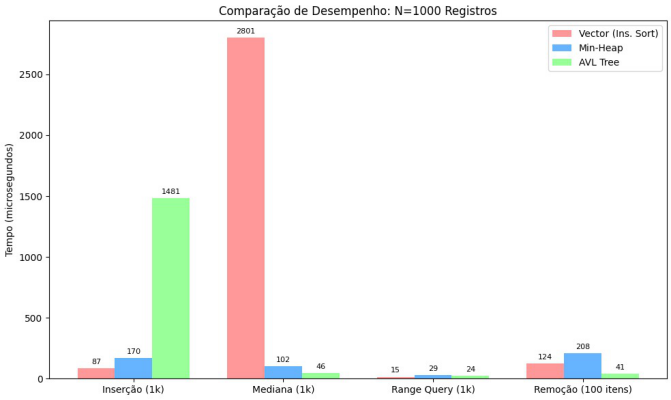

Figura 1. Comparativo de desempenho (em μs) para $N = 1000$ registros. Nota-se a disparidade no cálculo da mediana pelo Vetor.

```
=== BENCHMARK: MIN HEAP ===
[INSERT] 1000 registros | Tempo: 0.76160 ms
[MEDIAN] Resultado: 17.45500 | Tempo: 0.06110 ms
[MIN(3)] -9.96000 -9.94000 -9.92000 | Tempo: 0.06710 ms
[MAX(3)] 44.81000 44.81000 44.78000 | Tempo: 0.07010 ms
[RANGE(20, 25)] Encontrados: 90 itens | Tempo: 0.02110 ms
[REMOVE(22.5)] Tempo: 0.00520 ms
[PRINTSORTED] Gerada lista de 1000 itens | Tempo: 0.07130 ms
```

Figura 2. Dados Gerados pelo Programa sobre o Heap.

```
=== BENCHMARK: VECTOR INSERTION SORT (O(N^2)) ===
[INSERT] 1000 registros | Tempo: 0.68350 ms
[MEDIAN] Resultado: 17.45500 | Tempo (inclui Sort): 2.51440 ms
[MIN(3)] -9.96000 -9.94000 -9.92000 | Tempo: 0.00470 ms
[MAX(3)] 44.81000 44.81000 44.78000 | Tempo: 0.00470 ms
[RANGE(20, 25)] Encontrados: 90 itens | Tempo: 0.01300 ms
[REMOVE(22.5)] Tempo: 0.00480 ms
[PRINTSORTED] Lista de 1000 itens | Tempo: 0.00530 ms
```

Figura 3. Dados Gerados pelo Programa sobre o Insertion Sort.

```
=== BENCHMARK: AVL TREE ===
[INSERT] 1000 registros | Tempo: 1.78210 ms
[MEDIAN] Resultado: 17.45500 | Tempo: 0.04580 ms
[MIN(3)] -9.96000 -9.94000 -9.92000 | Tempo: 0.03440 ms
[MAX(3)] 44.81000 44.81000 44.78000 | Tempo: 0.03750 ms
[RANGE(20, 25)] Encontrados: 90 itens | Tempo: 0.00860 ms
[REMOVE(22.5)] Tempo: 0.00080 ms
[PRINTSORTED] Gerada lista de 1000 itens | Tempo: 0.02640 ms
```

Figura 4. Dados Gerados pelo Programa sobre a Arvore AVL.

```
=== RESULTADOS DO BENCHMARK (Microsegundos - us) ===
Operacao        HEAP        AVL         VEC(Ins)    Vencedor
-----------------------------------------------------------
Insert (1000x) 77          715         28          VEC
Median Calc    66          27          2288        AVL
Range Query    17          14          13          VEC
Remove (100x)  199         35          113         AVL

Analise Rapida:
- Vector (Ins) vence na insercao pois e O(1) (so adiciona ao fim).
- Vector perde feio na Mediana pois roda Insertion Sort O(N^2).
- AVL continua imbativel em Range e Remove.
```

Figura 5. Conclusões do Programa.

A. *Análise dos Dados*

Os testes foram realizados com um conjunto de $N = 1000$ registros de temperatura. A Tabela resume os tempos médios de execução obtidos em microssegundos (μs).

1) Inserção: O Vetor (Lazy) apresentou o melhor desempenho absoluto ($28\mu s$), sendo aproximadamente 2,7 vezes mais rápido que a Heap ($77\mu s$) e 25 vezes mais rápido que a AVL ($715\mu s$).

- **Análise:** O Vetor beneficia-se da alocação contígua de memória e da complexidade $O(1)$ amortizada do método push_back. A Heap, embora eficiente ($O(\log N)$), exige trocas de memória (swaps durante o sift-up. A AVL é penalizada severamente pelo overhead de alocação dinâmica de nós dispersos na memória e, principalmente, pelo custo computacional das rotações de rebalanceamento a cada inserção.

2) Cálculo da Mediana: Neste cenário, a abordagem do Vetor falhou drasticamente, registrando o pior tempo ($2288\mu s$). A Árvore AVL foi a mais eficiente ($27\mu s$).

- **Análise:** O custo oculto do Vetor é pago aqui. Para calcular a mediana, o vetor precisa ser ordenado. O algoritmo Insertion Sort utilizado tem complexidade $O(N^2)$, o que explica a explosão no tempo de execução. A AVL, por manter os dados estruturalmente ordenados, necessita apenas de um percurso em ordem (in-order traversal) com complexidade $O(N)$, ou $O(1)$ se a contagem de nós for mantida, tornando-a ideal para sistemas de monitoramento em tempo real.

3) Range Query e Cache: Observou-se um empate técnico entre o Vetor ($13\mu s$) e a AVL ($14\mu s$), com leve vantagem para o Vetor.

- **Análise:** Embora a AVL possua complexidade assintótica superior para buscas ($O(\log N + K)$), o Vetor ($O(N)$) compensa com a localidade espacial de referência. A CPU moderna consegue pré-carregar linhas de cache inteiras do vetor contíguo, acelerando a varredura linear. Em contrapartida, a AVL exige o acesso a nós dispersos na memória (ponteiros), resultando em cache misses que igualam o tempo de execução para $N = 1000$.

4) Remoção: A AVL demonstrou superioridade clara ($35\mu s$), sendo cerca de 3 vezes mais rápida que o Vetor e 5 vezes mais rápida que a Heap.

- **Análise:** Para remover um valor específico, tanto a Heap quanto o Vetor precisam primeiro encontrá-lo, o que exige uma busca linear $O(N)$. A AVL utiliza sua propriedade de busca binária para localizar o nó alvo em $O(\log N)$ e realizar a remoção localmente, confirmando sua robustez para manutenção do conjunto de dados.

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).