Henrique Coutinho Layber

Análise de Estruturas de Dados em C

Vitória
2019

Henrique Coutinho Layber

Análise de Estruturas de Dados em C

Relatório apresentado como resultado da lista 10 de exercício da disciplina de Estruturas de Dados 1, pela Universidade Federal do Espírito Santo.

Universidade Federal do Espírito Santo
Departamento de Informática

Vitória
2019

Sumário

	INTRODUÇÃO	3
1	MÉTODO UTILIZADO	4
2	RESULTADOS OBTIDOS	5
2.1	Dados Sequenciais	5
2.2	Dados Randômicos	6
3	ANÁLISE	7
3.1	Sequencial	7
3.1.1	Lista	7
3.1.2	Árvore Binária	7
3.1.3	Árvore AVL	7
3.2	Randômica	8
3.2.1	Lista	8
3.2.2	Árvore Binária	8
3.2.3	Árvore AVL	8
	CONCLUSÃO	9
	REFERÊNCIAS BIBLIOGRÁFICAS	10

Lista de ilustrações

Figura 1 – Inserções Sequenciai 5
Figura 2 – Busca (1) Sequencial 5
Figura 3 – Inserções Randômicas 6
Figura 4 – Busca (1) Randômica 6

INTRODUÇÃO

Este exercício tem por objetivo a análise das estruturas de dados Lista, Árvore Binária e Árvore *AVL* (Árvore Binária Balanceada).

Os arquivos de código fonte de todos os programas utilizados podem ser encontrados anexo a este relatório. As especificações da máquina[1] pode ser encontrado anexo a este documento. Os dados obtidos podem ser encontrados no meu Google Docs[2].

O programa carrega valores inteiros na estrutura, randomiza um valor dentre os inseridos, e busca ele na estrutura. Como o valor de uma busca é muito pequeno, torna-se difícil a mensuração dele. Para contornar isso, o programa busca um valor aleatório n vezes, e retorna dois valores: O tempo levado pra carregar todos os dados de entrada na estrutura e o tempo levado pra buscar o valor, n vezes.

Com base nesses dados capturados, podemos decidir melhor qual a estrutura mais eficiente em qual divisão de inserção de dados, busca de dados e em quais faixas de valores.

[1] Saída do comando `sudo lshw`
[2] Link para a planilha

1 MÉTODO UTILIZADO

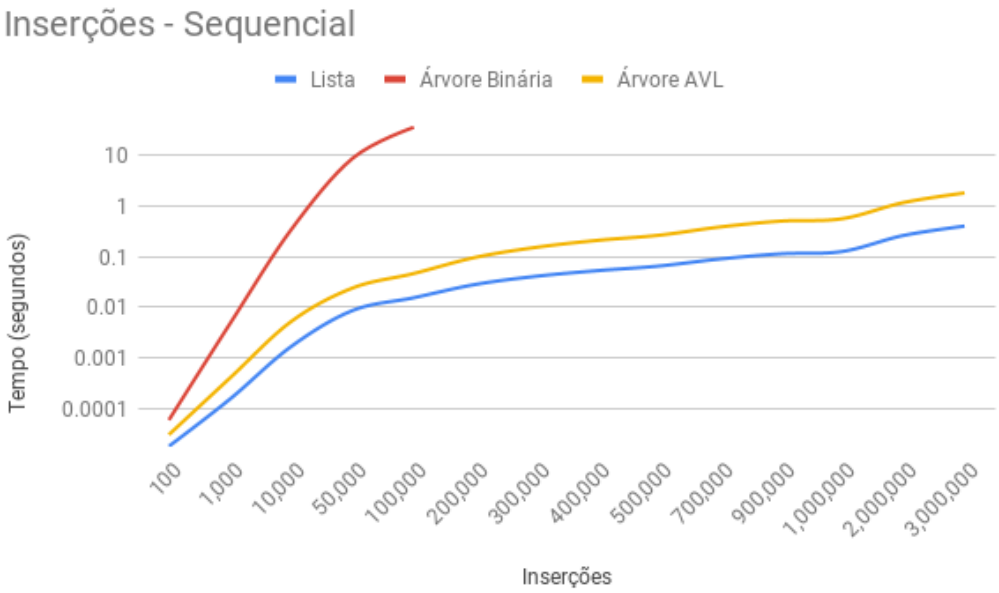
Como os olhos atentos podem já ter percebido, o programa basea-se em um valor aleatório, e isso significa, em geral, que será utilizada a função `rand()`. Como essa função é um tanto "diferente", foi resolvido utilizar um script *BASH* para fazer a execução do programa várias vezes. Com isso, obtemos resultado mais precisos.
Então, esses resultados foram colocados em uma planilha para mais fácil manipulação e visualização. Para obter os valores de busca unitária, após calcular a média de busca de n elementos, simplesmente dividiu-se essa média por n.

Como dados de entrada, foram usados dados sequenciais e randômicos, de:
100 (cem)
1000 (mil)
10000 (dez mil)
50000 (cinquenta mil)
100000 (cem mil)
200000 (duzentos mil)
300000 (trezentos mil)
400000 (quatrocentos mil)
500000 (quinhentos mil)
700000 (setecentos mil)
900000 (novecentos mil)
1000000 (um milhão)
2000000 (dois milhões)
3000000 (três milhões)

2 RESULTADOS OBTIDOS

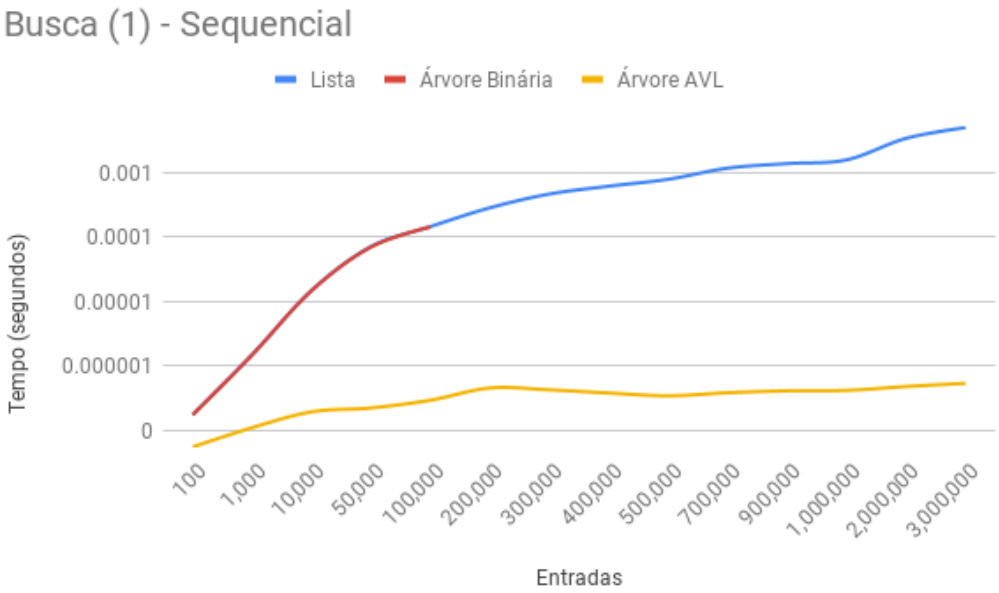
2.1 Dados Sequenciais

Figura 1 – Inserções Sequenciais



Fonte: elaborado pelo autor.

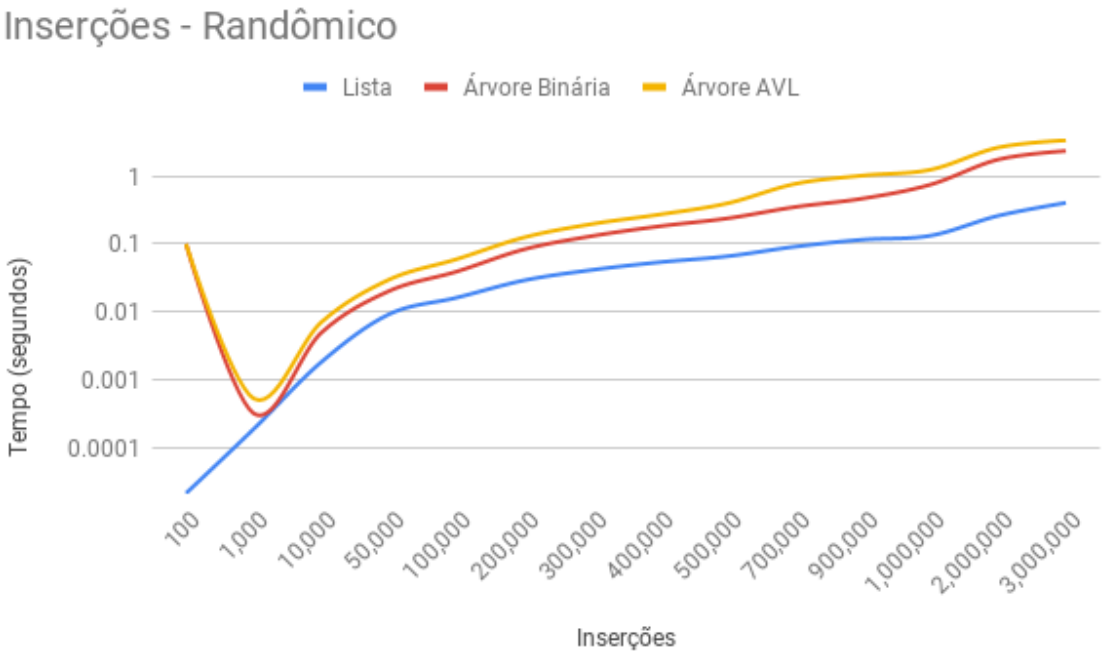
Figura 2 – Busca (1) Sequencial



Fonte: elaborado pelo autor.

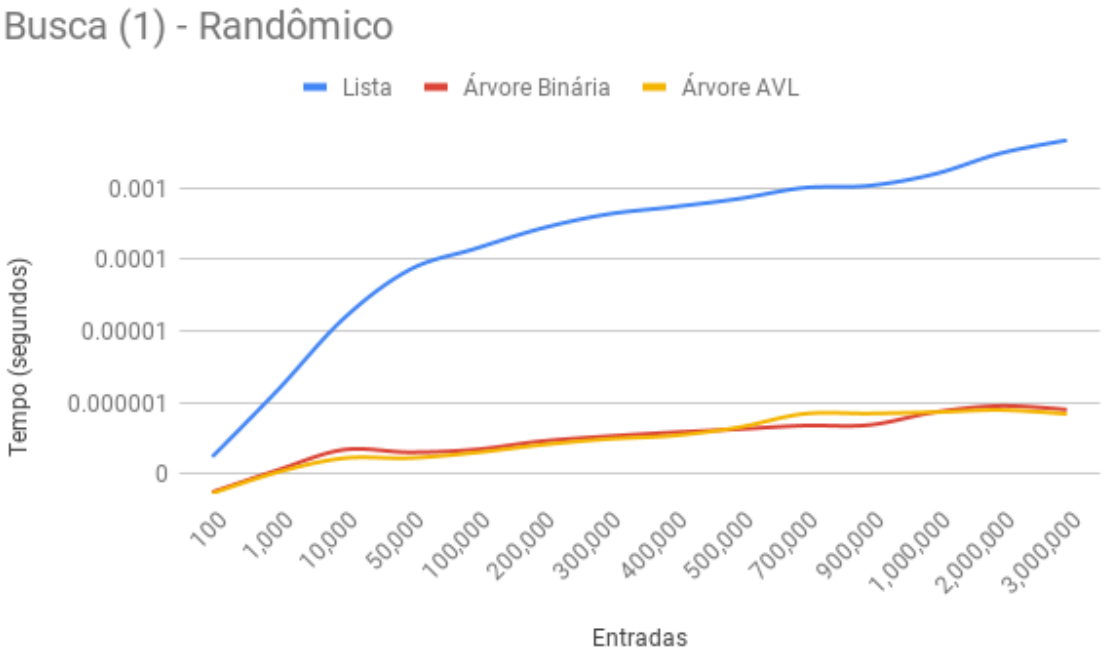
2.2 Dados Randômicos

Figura 3 – Inserções Randômicas



Fonte: elaborado pelo autor.

Figura 4 – Busca (1) Randômica



Fonte: elaborado pelo autor.

3 Análise

Podemos então analisar as informações contidas no gráfico e chegar a conclusões medidas. Devemos primeiro conhecer nossos dados de entrada, pois como vemos isso é crucial para o desemprenho do código, para implementarmos a estrutura mais eficiente.

3.1 Sequencial

3.1.1 Lista

A lista claramente se beneficia na inserção (de todo o tipo de dado), e é de fácil implementação, mas é seriamente prejudicada por percorrer somente em uma direção. Enquanto ela sempre supera na inserção, a busca é difícil. É uma boa opção pra quando temos casos de muitas, muitas inserções e pouca busca.

3.1.2 Árvore Binária

A árvore binária, se beneficia na busca dos dados, ou se beneficiaria se a entrada não fosse sequencial. Ela é então neste caso uma lista complicada[1], e de difícil abstração, e, usando a maneira recursiva, nem conseguiu chegar a mais de 100000 (cem mil) elementos inseridos, sofrendo de *Stack Overflow*[2] e sofre de perda de performance seríssima antes disso. Caso seja modificada para funcionar de maneira iterativa, ainda teria performance parecida (se não pior) que a da lista.

3.1.3 Árvore AVL

A árvore AVL é uma árvore binária que se balanceia automaticamente enquanto insere dados, ou seja, a inserção dela é a mais cara, porém:

1. Ela cai em menos casos de *Stack Overflow*, por não gerar muitos níveis de recursão em sua execução

2. Ela organiza os dados, mesmo que sequenciais de maneira efetiva a formar uma árvore, então a busca dela se torna `O(n)`.

Então ela resolve o problema de degeneração de árvore de dados encontrado pela árvore binária.

[1] Como a entrada é sequencial, todos os dados acabam sendo alocados em somente um lado da árvore, e ela se torna uma árvore degenerada em 100% dos casos.

[2] `https://pt.wikipedia.org/wiki/Stack_Overflow`

3.2 Randômica

3.2.1 Lista

A inserção da lista é igual independente de seus dados, então isso é inalterado. A busca também continua inalterada pelo sorteio de um valor para busca.

3.2.2 Árvore Binária

É com entradas randômicas que as estruturas de árvore brilham. Aqui, mesmo a árvore binária não tem problema com *Stack Overflow*, pois como os dados não ficam todos indo para apenas um galho, ela fica naturalmente distribuída, por pura chance. Assim, a altura dela fica menor e menos chamadas recursivas são necessárias para chegar até o alvo, reduzindo drasticamente o tempo de busca.

3.2.3 Árvore AVL

A árvore AVL também naturalmente ganha uma vantagem. A busca dela é em teoria mais rápida e consistente que a da árvore binária convencional, pois ela procura manter uma constância de altura entre seus nós, mas na prática, como os dados são aleatoriamente gerados (e não orgânicos) isso acaba não impactando tanto no resultado.

CONCLUSÃO

É certo então que, se possível, devemos conhecer bem nosso caso de entrada para escolher o algoritmo mais eficiente e fazer um programa robusto e melhor otimizado.

Recomendo usar árvore AVL quando a situação encontrada se aproxima de entradas sequenciais, por ter a inserção somente um pouco mais cara que a lista, mas a busca é muito mais rápida. E jamais usar árvores binárias convencionais.

Recomendo usar árvore binária quando a situação encontrada se aproxima de entrada randômicas, a própria aleatoriedade dos dados se encarregará de distribuída e a busca tornará-se tão rápida quando a árvore AVL.

Recomendo usar lista quando a maior preocupação da situação encontrada é a inserção. Ela tem a inserção sempre mais rápida, e tem uma simples implementação.

De qualquer forma, a escolha de qual estrutura deve ser feita ultimamente por quem implemente, e é muito difícil dizer com certeza a partir de que ponto o "correto" seria usar essa ou aquela, pois dos problemas que se encontram, são vários espectros que leva-se em conta: a entrada normalmente não é 100% randômica, ou 100% sequencial, é sempre algo no meio. E como se mensura a "necessidade" de inserir pela necessidade de buscar? Se você nunca vai buscar, tudo bem, mas se for este o caso, qual o sentido em guardar o dado em primeiro momento?

Com o conhecimento das implementações dos TADs, e o conhecimento de seus pontos fortes e fracos, podemos tomar decisões concisas sobre o uso das mesmas, e não apenas reprodutores de conteúdo, cuspindo apenas o que é pedido, como cachorros treinados. Essas decisões é o que diferencia o programador de código do arquiteto de código, que cada programa é uma obra crítica e bem pensada.

REFERÊNCIAS BIBLIOGRÁFICAS

MOTA, Vinícius Fernandes Soares. *Notas de aula: Estrutura de Dados I. 10 de junho de 2019, 12 de junho de 2019.*

ZIVIANI, Nivio. *Projetos de Algoritmos com implementações em Pascal e C.*

D. E. Knuth. *The Art of Computer Programming.*

`https://pt.wikipedia.org/wiki/Lista`

`https://pt.wikipedia.org/wiki/%C3%81rvore_bin%C3%A1ria_de_busca`

`https://pt.wikipedia.org/wiki/%C3%81rvore_AVL`

`https://pt.wikipedia.org/wiki/Stack_Overflow`