Uma análise comparativa de três estruturas de dados primordiais: vetor, lista e tabela hash

Flavio L. Fernandes[1], **Eduardo H. M. da Cruz**[1]

[1]Instituto Federal do Paraná – campus Paranavaí – Brasil

flaviolukfer@gmail.com, eduardo.cruz@ifpr.edu.br

Abstract. *Data structuring is fundamental for software as its routines and functionalities. Therefore, comprehending the different approaches to structuring data and their differences is pretty essential. So, our work studies three of the main data structures: vector, list and hash table; bringing basic operations' implementations and doing experiments to compare the efficiency of these data structures and collaborate to comprehend the appropriate scenarios of each approach.*

Resumo. *Estruturar dados corretamente é parte fundamental para um programa, tanto quanto suas rotinas e funcionamento. Portanto, é essencial compreender as diferentes abordagens existentes para se estruturar dados e as diferenças entre elas. Sendo assim, este trabalho conduz um estudo de três das principais estruturas de dados: vetor, lista e tabela hash; trazendo implementações de suas operações básicas e realizando experimentos com objetivo de comparar a eficiência destas estruturas de dados e colaborar com a compreensão dos cenários apropriados para cada abordagem.*

1. Introdução

"Conjuntos são tão fundamentais para a Ciência da Computação quanto para a Matemática." [Cormen et al. 2022]. Todo programa gera, armazena ou processa dados de alguma forma, fazendo importante entender as possibilidades existentes e mais usadas para estruturar os dados de um sistema de computação. Programas são constituídos por algoritmos e estrutura de dados, portanto, um bom programa é uma combinação de ambos [Schildt 2021].

A escolha e implementação corretas de uma estrutura de dados são de suma importância para um programa, tanto quanto suas rotinas e funcionamento. As formas como os dados são armazenados e organizados dependem da natureza do problema a ser solucionado [Schildt 2021]. Existem várias abordagens conhecidas e bem consolidadas para estruturar e organizar dados. Algumas das mais populares são: ***vetores***, ***listas***, ***árvores***, entre outras.

Deste modo, este trabalho objetiva conduzir um estudo de três estruturas de dados básicas: vetor, lista e tabela hash. Além disso, este trabalho também objetiva analisar e comparar duas destas estruturas: vetor e lista, visando compreender as diferenças entre elas e colaborar na compreensão de quando escolher uma em detrimento de outra. O presente trabalho está estruturado conforme a seguir.

Nas Seções 2 , 3 e 4, são conduzidos estudos das estruturas de dados em questão. Para cada estrutura, são apresentados algoritmos para as operações de *inserção*, *busca* e

deleção, em forma de pseudocódigos. Para todas as operações são explanadas suas respectivas complexidades de tempo. Na Seção 5, são dados detalhes de como os estudos e experimentos foram conduzidos e como as análises foram realizadas. Na Seção 6, são explanados e discutidos os resultados experimentais, utilizando representações gráficas para expor os resultados de forma clara e objetiva. Por fim, na Seção 7, é exposto um diálogo a respeito da importância deste trabalho para as áreas relacionadas e quais próximos passos devem ser tomados para incrementar a contribuição da presente pesquisa.

2. Vetor

Vetor é uma estrutura de dados que representa um arranjo que pode mudar de ***tamanho*** e de ***capacidade*** (entende-se por tamanho a quantidade real de elementos contidos no arranjo e entende-se por capacidade a quantidade de espaços em memória alocados para o arranjo) [cplusplus.com 2022]. Esta é a grande diferença entre vetor e arranjo, pois enquanto o arranjo é uma estrutura estática, o vetor é uma estrutura que pode ser dinâmica.

Internamente o vetor usa um arranjo dinamicamente alocado para armazenar seus elementos. Este arranjo pode precisar ser realocado para que possa mudar dinamicamente sua capacidade, o que implica em alocar um novo arranjo e mover todos os elementos para ele, desalocando o arranjo original após o processo [cplusplus.com 2022]. A realocação é a operação mais custosa da estrutura de dados vetor e é por meio da realocação que se dá a dinamicidade da estrutura.

2.1. Definição da classe

A classe *Vector* utiliza 3 atributos (variáveis) internos. O atributo *arr* é um ponteiro que guarda o endereço de memória do arranjo interno do vetor, portanto, seu valor inicial é *NIL*. Os atributos *capacidade* e *tamanho* são utilizados para representar o tamanho em memória do arranjo interno e a quantidade de dados armazenados no arranjo, respectivamente. O tamanho do vetor é inicializado em 0 e a capacidade é inicializada no construtor da classe, recebida por parâmetro. O construtor também é responsável por alocar memória para o arranjo interno. O Algoritmo 1 implementa o cabeçalho da classe *Vector*.

```
1  Vector
2      arr ⟵ NIL
3      capacidade
4      tamanho ⟵ 0
5
6      Vector (capacidade)
7          this.capacidade ⟵ capacidade
8          this.arr ⟵ aloca [capacidade]
9      ...
```

Algoritmo 1: Atributos e construtor da classe Vector.

2.2. Inserção

Assim como na estrutura de dados *pilha*, em um vetor a operação de inserção é frequentemente chamada de *push* (empurrar, do inglês), isto porque, ao inserir um dado em um vetor, este dado é inserido no final da estrutura [Cormen et al. 2022]. O Algoritmo 2 implementa a operação de inserção de dados no vetor.

A implementação da operação de inserção de dados no vetor apresentada neste trabalho, baseia-se na implementação da biblioteca padrão da linguagem de programação

```
1  Vector
2      ...
3      Push (dado)
4          se this.tamanho < this.capacidade então
5              this.arr[this.tamanho] ⟵ dado
6          senão
7              this.capacidade ⟵ this.capacidade + 1
8              temp ⟵ this.arr
9              desaloca this.arr
10             this.arr ⟵ aloca [this.capacidade]
11             para i ⟵ 0 até this.tamanho − 1 faça
12                 this.arr[i] ⟵ temp[i]
13             desaloca temp
14             this.arr[this.tamanho] ⟵ dado
15         this.tamanho ⟵ this.tamanho + 1
16     ...
```

Algoritmo 2: Operação de inserção de dados no vetor.

C++[1]. A operação de inserção de dados no vetor, em geral, possui complexidade constante ($\theta(1)$). No caso onde há a necessidade de realocar o arranjo interno para aumentar a capacidade do vetor, a complexidade é linear ($\theta(N)$) [cplusplus.com 2022]. Alocar e desalocar dados são operações extremamente custosas para o processamento, portanto, isto também deve ser levado em consideração.

2.3. Busca

Assim como em arranjos, os dados de um vetor são acessados através de índices, como explicado no início da seção. Porém, para buscar um dado é necessário percorrer o vetor até que o dado seja encontrado. Portanto, o método *Get* recebe um dado por parâmetro e retorna o dado caso o encontre, caso contrário, retorna um erro de dado não encontrado. O Algoritmo 3 implementa a operação de busca de dados do vetor.

```
1  Vector
2      ...
3      Get (dado)
4          para i ⟵ 0 até this.tamanho faça
5              se dado = this.arr[i] então
6                  retorna this.arr[i]
7          erro "dado não encontrado"
8      ...
```

Algoritmo 3: Operação de busca de dados do vetor.

A complexidade do método *Get* é linear ($\theta(N)$).

2.4. Deleção

Há duas operações de deleção frequentemente utilizadas em vetores: deletar um dado no final do vetor e deletar um dado em uma posição específica. Este trabalho implementa somente a deleção em uma posição específica, para qual o nome *erase* (apagar, do inglês) é comumente utilizado. Apesar de a deleção ser baseada no índice, contudo, é necessário fazer uma busca linear similar a busca implementada no método *Get*, porém para encontrar o índice. O Algoritmo 4 implementa a operação de deleção de dados do vetor.

[1] cplusplus.com

```
1  Vector
2      ...
3      Erase (dado)
4          índice ⟵ −1
5          para i ⟵ 0 até this.tamanho faça
6              se dado = this.arr[i] então
7                  índice ⟵ i
8                  quebrar laço
9          se = −1 então
10             retorna −1
11         para i ⟵ 0 até this.tamanho faça
12             this.arr[i] ⟵ this.arr[i+1]
13         this.tamanho ⟵ this.tamanho − 1
14         se índice − 1 = this.tamanho − 1 então
15             retorna índice − 1
16         retorna índice
```

Algoritmo 4: Operação de deleção de dados do vetor.

O método *Erase* possui complexidade linear ($\theta(N)$) [cplusplus.com 2022].

3. Lista duplamente ligada

A lista ligada, ou encadeada, é uma estrutura de dados dinâmica onde os dados são organizados em ordem linear. Diferente de um vetor, a ordem linear de uma lista ligada é determinada por ponteiros e não por índices [Cormen et al. 2022]. Enquanto no vetor os elementos ocupam endereços contíguos de memória, em uma lista ligada os elementos ocupam endereços arbitrários de memória, como ilustrado pela Figura 1.

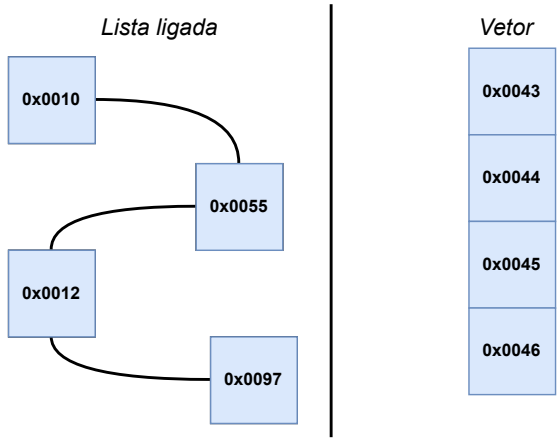

Figura 1. Endereçamento de memória dos elementos de uma lista ligada versus endereçamento de memória de um vetor.

A lista ligada implementada neste trabalho é a lista duplamente ligada. A lista duplamente ligada é um tipo de lista ligada onde cada nó aponta para o próximo nó da lista e também para o nó anterior [Cormen et al. 2022], como é possível ver na Figura 2.

Nós são pequenas estruturas de dados, que neste trabalho também serão representados como objetos, que armazenam um dado e dois ponteiros, um para o próximo nó e outro para o nó anterior, como demonstra o Algoritmo 5.

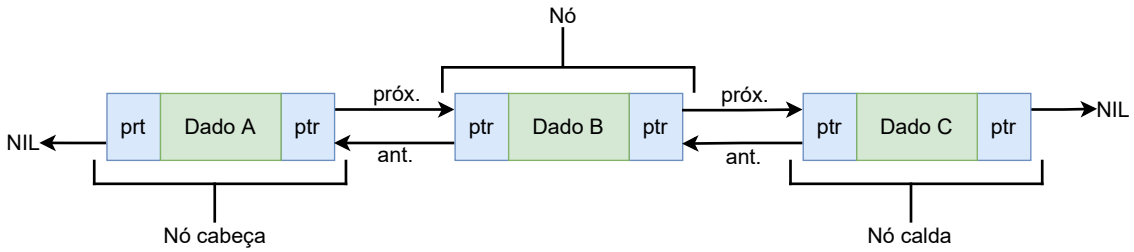

Figura 2. Representação de uma lista duplamente ligada.

```
1 Node
2     dado
3     próximo ⟵ NIL
4     anterior ⟵ NIL
5
6     Node (dado)
7         this.dado ⟵ dado
```

Algoritmo 5: Nó de uma lista duplamente ligada.

3.1. Definição da classe

A lista duplamente ligada possui dois tipos especiais de nós, comumente chamados de *nó cabeça* e *nó calda*. O nó cabeça armazena o dado que está no início da lista e como não há anterior a ele, logo, o ponteiro do nó anterior recebe o valor *NIL*. O nó calda, por sua vez, armazena o dado que está no final da lista, logo, o ponteiro do próximo nó recebe o valor *NIL* como a Figura 2 ilustra. Além destes dois ponteiros, a implementação de lista abordada neste trabalho adiciona um atributo *tamanho*, responsável por armazenar a quantidade de dados contidos na lista. O Algoritmo 6 implementa o cabeçalho da classe *Doubly-Linked-List*.

```
1 Doubly-Linked-List
2     cabeça ⟵ NIL
3     calda ⟵ NIL
4     tamanho ⟵ 0
```

Algoritmo 6: Atributos da classe Doubly-Linked-List.

3.2. Inserção

Similar ao vetor, a operação de inserção na lista é frequentemente chamada de *push*, contudo, a lista duplamente ligada possibilita também a inserção de dados no início da estrutura. Deste modo, é comum ter dois métodos *push*, frequentemente chamados de *push front* e *push back* para adicionar dados no começo e no final da lista, respectivamente. Este trabalho implementa somente a inserção no final da lista.

Os 2 métodos de inserção de dados na lista duplamente ligada são muito parecidos, o que reflete em suas complexidades já que ambos possuem complexidade constante ($\theta(1)$) [cplusplus.com 2022]. O Algoritmo 7 implementa a operação de inserção de dados na lista duplamente ligada.

3.3. Busca

A operação de busca implementada no método *Get* é uma busca linear simples que no fim retorna um ponteiro para o nó da lista cujo dado é igual ao dado recebido por parâmetro.

```
1  Doubly-Linked-List
2      ...
3      Push-Back (dado)
4          novo-nó ⟵ aloca Node(data)
5          this.tamanho ⟵ this.tamanho + 1
6          se this.cabeça ≠ NIL então
7              this.cabeça ⟵ novo-nó
8              this.calda ⟵ novo-nó
9              retorna
10         this.calda.próximo ⟵ novo-nó
11         novo-nó.anterior ⟵ this.calda
12         this.calda ⟵ novo-nó
13     ...
```

Algoritmo 7: Operação de inserção de dados na lista duplamente ligada.

Caso o dado não esteja armazenado na lista, o método retorna *NIL* [Cormen et al. 2022]. O Algoritmo 8 implementa a operação de busca de dados da lista duplamente ligada.

```
1  Doubly-Linked-List
2      ...
3      Get (dado)
4          nó ⟵ this.cabeça
5          enquanto nó ≠ NIL e nó.dado ≠ dado faça
6              nó ⟵ nó.próximo
7          retorna nó
8      ...
```

Algoritmo 8: Operação de busca de dados na lista duplamente ligada.

A complexidade do método *Get* é linear ($\theta(N)$) [Cormen et al. 2022].

3.4. Deleção

A operação de deleção implementada no método *Erase* remove da lista um nó cujo ponteiro é recebido por parâmetro. Caso seja necessário remover um elemento com determinado dado, deve-se chamar primeiro o método *Get* [Cormen et al. 2022]. O Algoritmo 9 implementa a operação de deleção de dados da lista duplamente ligada.

A complexidade do método *Erase* é constante ($\theta(1)$). Porém, caso seja necessário encontrar primeiro o nó para depois removê-lo utilizando o método *Get*, então a complexidade de *Erase* será a mesma de *Get*, ou seja, linear ($\theta(N)$) [Cormen et al. 2022].

4. Tabela hash

Também chamada de tabela de espalhamento, a tabela hash é uma estrutura de dados de dados eficaz para implementar dicionários, isto porquê, em média, as operações básicas de inserção, busca e deleção (operações básicas de um dicionário) possuem complexidade constante ($O(1)$) [Cormen et al. 2022].

Uma tabela de espalhamento pode ser implementada tanto como uma estrutura de dados estática, quanto como dinâmica. A tabela hash implementada neste trabalho é dinâmica e utiliza o *encadeamento* como estratégia para tratar *colisões*.

Uma tabela de espalhamento utiliza uma combinação chave-valor para mapear os dados no arranjo interno. Neste trabalho a micro-estrutura que representa a combinação chave-valor é chamada de *Pair* (par, do Inglês) e é o elemento base da tabela hash. O

```
1  Doubly-Linked-List
2      ...
3      Erase (nó)
4          this.tamanho ⟵ this.tamanho − 1
5          se nó = this.cabeça então
6              se nó = this.calda então
7                  this.cabeça ⟵ NIL
8                  this.calda ⟵ NIL
9                  this.tamanho ⟵ 0
10             senão
11                 this.cabeça ⟵ nó.próximo
12                 this.cabeça.anterior ⟵ NIL
13         senão se nó = this.calda então
14             this.calda ⟵ nó.anterior
15             this.calda.próximo ⟵ NIL
16         senão
17             nó.próximo.anterior ⟵ nó.anterior
18             nó.anterior.próximo ⟵ nó.próximo
19         desaloca nó
```

Algoritmo 9: Operação de deleção de dados da lista duplamente ligada.

Algoritmo 10 implementa a estrutura *Pair*. Instâncias de um par chave-valor também serão tratadas como objetos no presente trabalho.

```
1  Pair
2      chave
3      valor
4
5      Pair (chave, valor)
6          this.chave ⟵ chave
7          this.valor ⟵ valor
```

Algoritmo 10: Par chave-valor.

4.1. Função de hash

A função de hash é extremamente importante para a tabela de espalhamento. Através da chave e da função de hash é possível encontrar diretamente a posição do arranjo que o elemento ocupa, tornando as operações de inserção, busca e deleção expressamente rápidas e eficientes. Existem várias implementações diferentes de função de hash, este trabalho aborda uma das mais utilizadas, conhecida como **método de divisão**.

O método de divisão mapeia uma determinada chave k para uma de n posições do arranjo através do resto da divisão de k por n, dada pela Equação 1 [Cormen et al. 2022]. O Algoritmo 11 implementa a função de hash utilizando método de divisão.

$$h(k) = k \ mod \ n \tag{1}$$

```
1  Hash (chave, n)
2      retorna chave mod n
```

Algoritmo 11: Função de hash.

4.2. Colisões e tratamento de colisões por encadeamento

É possível que a função de hash mapeie um elemento para uma posição do arranjo que outro elemento já ocupa, a isso é dado o nome *colisão*. Uma técnica bastante utilizada para tratar colisões em uma tabela hash é chamada *encadeamento*.

Nesta abordagem, em cada posição do arranjo é armazenada uma *lista ligada* e cada novo elemento inserido na tabela é mapeado para uma posição do arranjo e inserido na lista dessa posição, deste modo, quando há colisões, os elementos que colidem simplesmente são inseridos na lista. O encadeamento é a técnica de tratamento de colisões abordada por este trabalho. A lista ligada usada para o encadeamento é a mesma lista duplamente ligada implementada na Seção 3.

4.3. Definição da classe

Como demonstra o Algoritmo 12, a classe *Hash-Table* possui os mesmos atributos que a classe *Vector*, contudo, o atributo *tamanho* da tabela hash representa todos os dados inseridos nas listas. No construtor da classe, para cada posição do arranjo é inicializada uma lista duplamente ligada (linhas 9 e 10 do Algoritmo 12), isto é necessário pois, como mencionado anteriormente, a técnica de tratamento de colisões implementada neste trabalho é a abordagem de *encadeamento*.

```
1  Hash-Table
2      arr ⟵ NIL
3      capacidade
4      tamanho ⟵ 0
5
6      HashTable (capacidade)
7          this.capacidade ⟵ capacidade
8          this.arr ⟵ aloca [capacidade]
9          para i ⟵ 0 até capacidade faça
10             this.arr[i] ⟵ Doubly-Linked-List()
11     ...
```

Algoritmo 12: Atributos e construtor da classe Hash-Table.

4.4. Inserção

Assim como na árvore binária, a operação de inserção em uma tabela hash é comumente chamada de *add*. O método *Add* da tabela hash recebe por parâmetro um objeto de *Pair* (Algoritmo 10), contendo a chave e valor a serem inseridos na tabela. A posição no arranjo é mapeada utilizando a função de hash (Seção 4.1) aplicada na chave do par, encontrando-se a lista duplamente ligada que ocupa a posição mapeada do arranjo e inserindo o par no fim da lista. O Algoritmo 13 implementa a operação de inserção na tabela hash.

```
1  Hash-Table
2      ...
3      Add (par)
4          lista ⟵ this.arr[Hash(par.chave)]
5          lista.Push-Back(par)
6          this.tamanho ⟵ this.tamanho + 1
7      ...
```

Algoritmo 13: Operação de inserção de dados na tabela hash.

A complexidade do método *Add* é constante ($\theta(1)$), assim como o método *Push-Back* da lista duplamente ligada [Cormen et al. 2022].

4.5. Busca

Similar a operação de inserção, a operação de busca, implementada na função *Get* do Algoritmo 14, encontra a posição da lista em que o dado está inserido aplicando a função de hash na chave que é recebida por parâmetro. Então, a lista duplamente ligada é percorrida até encontrar a primeira ocorrência de um par com a chave recebida.

```
1  Hash-Table
2      ...
3      Get (chave)
4          lista ⟵ this.arr[Hash(chave)]
5          nó ⟵ lista.cabeça
6          enquanto nó ≠ NIL e nó.dado.chave ≠ chave faça
7              nó ⟵ nó.próximo
8          retorna nó.dado
9      ...
```

Algoritmo 14: Operação de busca de dados na tabela hash.

A complexidade do método *Get* é linear ($\theta(N)$) [Cormen et al. 2022]. É importante ressaltar que a complexidade é linear devido a estratégia de tratamento de colisões utilizando a lista duplamente ligada. Devido a grande diminuição no espaço de busca, a busca da tabela hash é muito mais rápida que a busca da lista, apesar de ambas complexidades serem $\theta(N)$. Usando estratégias diferentes de tratamento de colisão seria possível alcançar complexidade logarítmica ($O(lg(N))$) ou até mesmo complexidade constante.

4.6. Deleção

Assim como as operação de busca, a operação de deleção, implementada no método *Erase* do Algoritmo 15, também aplica a função de hash na chave recebida por parâmetro para encontrar a posição da lista duplamente ligada onde o par está armazenado. A lista também é percorrida até encontrar a primeira ocorrência de um par com a chave recebida e então o nó da lista é removido utilizando o método *Erase*.

```
1  Hash-Table
2      ...
3      Erase (chave)
4          lista ⟵ this.arr[Hash(chave)]
5          nó ⟵ lista.cabeça
6          enquanto nó ≠ NIL faça
7              se nó.dado.chave = chave então
8                  lista.Erase(nó)
9                  retorna
10             nó ⟵ nó.próximo
11     ...
```

Algoritmo 15: Operação de deleção de dados na tabela hash.

A complexidade do método *Erase* é linear ($\theta(N)$) [Cormen et al. 2022]. O mesmo ressaltado para o método *Get* se aplica para o método *Erase*, a complexidade é linear devido a complexidade da deleção da lista, mas poderia ser diferente com diferentes abordagens de tratamento de colisão.

5. Metodologia de avaliação

Para realização dos experimentos, foi construída uma biblioteca, na linguagem de programação C++, contendo todas as estruturas de dados abordadas nas seções anteriores. O código fonte da biblioteca está disponível em repositório online no site GitHub[2] e pode ser acessado através do link github.com/ifpr-paranavai/yet-another-data-structure-library.

As operações de inserção, busca e deleção das estruturas de dados abordadas foram testadas da seguinte forma:

1. Foram gerados 30 arquivos contendo 2000000 (dois milhões) de números inteiros gerados aleatoriamente entre 0 e 1999999 (um milhão novecentos e noventa e nove mil novecentos e noventa e nove);
2. Para cada operação de cada estrutura de dados foram realizados 30 testes. Em cada um dos testes, o conteúdo de um dos arquivos gerados foi utilizado como entrada para os algoritmos;
3. Após testadas as operações de determinada estrutura de dados, foi gerado um arquivo CSV contendo todos os 30 resultados de cada operação;
4. Os arquivos CSV gerados foram utilizados para análise dos resultados e produção de gráficos. Os gráficos expostos na Seção 6 foram produzidos na plataforma Colaboratory[3], utilizando as bibliotecas pandas[4], NumPy[5], plotly[6] e SciPy[7] na linguagem de programação Python. Os *scripts* desenvolvidos para análise dos resultados e produção dos gráficos pode ser acessado pelo link colab.research.google.com/drive/1H8PKd6zBct8QwM4bcc8jGl6X0TLki3qm?usp=sharing.

Os testes realizados foram executados no sistema operacional Ubuntu, versão 22.04, em uma máquina virtual. Mais detalhes sobre a configuração do ambiente em que os experimentos foram realizados estão presentes na Tabela 1. Todos os códigos executados foram compilados com parâmetro *-02* de otimização.

Sistema	Parâmetro	Valor
Máquina física	Processador	1x AMD Ryzen 5 3600X, 6 núcleos, 12 threads, 3.8GHz clock base, 4.4GHz clock máximo
	Caches/proc.	12x 32KB L1, 6x 512KB L2, 2x 16MB L3
	Memória principal	2x 8GB DDR4-3200
	Ambiente	Windows 11
Máquina virtual	Processadores	6
	Memória principal	8GB
	Ambiente	Ubuntu 22.04, Linux kernel 5.15.0, GCC 11.2.0

Tabela 1. Configuração dos experimentos.

[2]github.com
[3]colab.research.google.com
[4]pandas.pydata.org
[5]numpy.org
[6]plotly.com
[7]scipy.org

6. Resultados experimentais

O gráfico da Figura 3 compara as médias dos tempos de execução, coletados no experimento, das operações de inserção, busca e deleção. O gráfico de barras conta com as operações no *eixo x* e o tempo médio de execução no *eixo y*, em segundos. Cada barra contém o valor real da média do tempo de execução da operação. Além disso, cada média conta com intervalo de confiança de 95%, utilizando distribuição T de Student, em forma de barra de erro sobre cada barra do gráfico. É importante ressaltar que, devido ao valor muitíssimo discrepante dos demais, não é possível ver o topo da barra de tempo médio da operação de deleção do vetor, porém, os valores do limite inferior e superior calculados foram aproximadamente 140 e 145, respectivamente.

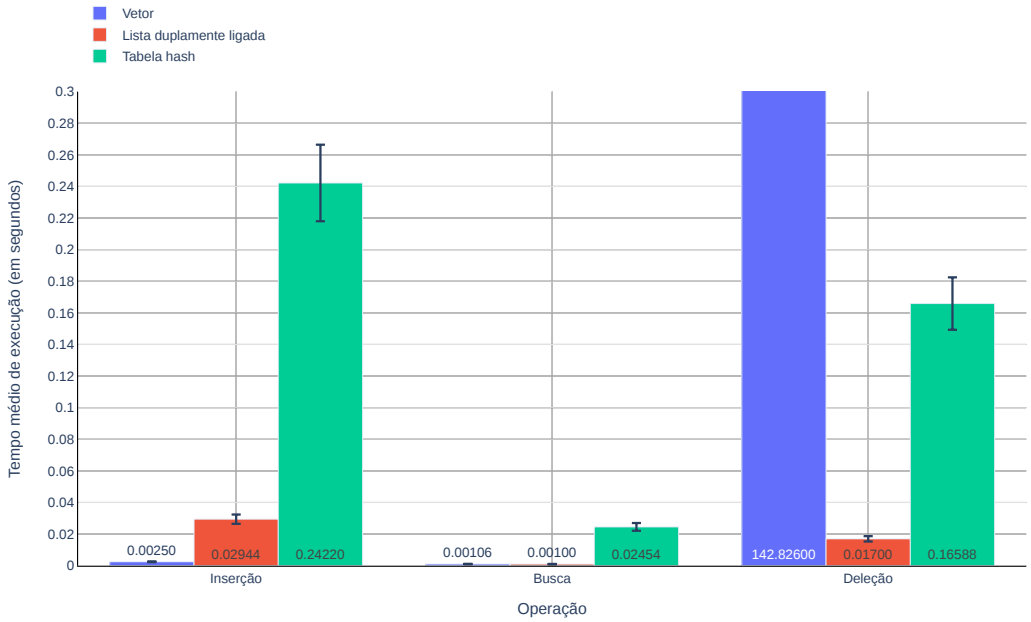

Figura 3. Média do tempo de execução das operações de inserção, busca e deleção das estruturas de dados vetor, lista duplamente ligada e tabela hash em segundos.

É possível notar que, tratando-se de dois milhões de dados do tipo inteiro, tanto vetor quanto lista apresentaram médias bastante satisfatórias nas operações de inserção e busca. Contudo, a inserção no vetor é consideravelmente mais rápida por não haver a necessidade de alocar memória para o dado, operação extremamente custosa para o desempenho e que reflete no gráfico se comparado vetor e lista, apesar de os tempos serem bastante baixos de maneira prática, tratando-se de 2 milhões de elementos. É importante mencionar que, a inserção no vetor não utilizou de realocação visto que isto não é um comportamento de natureza da estrutura e não é uma operação trivial, sendo de grande custo para o desempenho.

A maior diferença apresentou-se na operação de deleção. Enquanto a lista obteve uma média bastante baixa, a média da deleção do vetor foi de aproximadamente 142 segundos. Essa grande diferença pode ser explicada ao observar a implementação da operação de deleção do método *Erase* do vetor, no Algoritmo 4. Quando um dado é

removido do vetor e este dado não está na última posição da estrutura, é necessário movê-lo para a região de lixo e mover uma posição para a esquerda todos os seus sucessores. Mover todos os sucessores uma posição para a esquerda é uma tarefa bastante custosa para o processador, portanto lenta, o que reflete no tempo médio de execução do algoritmo.

É importante mencionar que, para o teste da operação de deleção na lista duplamente ligada, foi utilizada a abordagem em que se é necessário utilizar a operação de busca para encontrar o ponteiro do nó antes de deletar o dado de fato.

Os resultados apresentados para a tabela hash precisam de mais análise e investigação visto que, principalmente a busca deveria apresentar um tempo de execução melhor que o da lista já que, apesar de usar o encadeamento como estratégia de colisão, a tabela hash reduz significantemente o espaço de busca. A inserção também apresenta média controversa, portanto, os testes para esta estrutura precisam ser reanalisados, investigados e corrigidos. Trocar a função de hash para uma abordagem mais eficiente é algo que também deve ser ponderado.

7. Conclusão e trabalhos futuros

É notória a importância de organizar e gerir bem os dados dentro de um programa. Portanto, escolher corretamente qual abordagem utilizar para estruturar os dados de determinado contexto é tão importante quanto as regras que compõe o programa [Schildt 2021]. Com os experimentos conduzidos, foi possível evidenciar a eficiência das estruturas de dados vetor e lista duplamente ligada. Ambas apresentam tempos de execução próximos e bastante satisfatórios, porém, é importante compreender que remover dados de um vetor não é um comportamento trivial e é uma operação bastante custos.

É importante que trabalhos futuros abranjam outras estruturas de dados, como *árvore binária*. Também é importante que mais experimentos sejam realizados, com tamanhos de entradas diferentes e contextos diferentes. Faz-se essencial também que os testes referentes a tabela hash sejam reanalisados, investigados, corrigidos e, se necessário, refeitos, para que contenham dados mais confiáveis e condizentes com a natureza e propósito desta estrutura.

Referências

[Cormen et al. 2022] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2022). *Introduction to algorithms*. MIT press.

[cplusplus.com 2022] cplusplus.com (2022). Standard c++ library reference. `https://cplusplus.com/reference`. Acessado em: 15 jun. 2022.

[Schildt 2021] Schildt, H. (2021). C the complete reference.