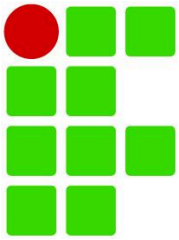

**INSTITUTO FEDERAL**
**DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA**
Rio Grande do Norte

**DIATINF** – Diretoria Acadêmica de Gestão e Tecnologia da Informação **TADS** – Tecnologia em Análise e Desenvolvimento de Sistemas

**Algoritmos** – 2024.1
**Trabalho** – Vetor dinâmico

**Professor:** Jorgiano Vidal

**Discentes:** Marya Eduarda Alexandre da Silva - 20231014040041

## 01 - Introdução

Este relatório apresenta a implementação e análise de desempenho de duas estruturas de dados: a lista duplamente ligada e o vetor dinâmico. O objetivo é comparar a eficiência dessas estruturas em operações comuns como inserção, remoção e acesso, utilizando diferentes cenários de teste.

**Exemplo de Expansão:**

As estruturas de dados são componentes cruciais em qualquer linguagem de programação, permitindo que desenvolvedores armazenem, organizem e manipulem dados de maneira eficiente. Neste relatório, focamos em duas estruturas de dados amplamente utilizadas: a lista duplamente ligada e o vetor dinâmico. Ambas têm suas vantagens e desvantagens, dependendo do contexto de aplicação. O objetivo deste estudo é implementar essas estruturas, avaliar seu desempenho em operações comuns como inserção, remoção e acesso, e comparar sua eficiência em diferentes cenários de teste.

## 02 - Vetores Dinâmicos

### Definição e Funcionamento

Um vetor dinâmico é uma estrutura de dados que simula o comportamento de arrays, mas com a capacidade de redimensionar-se automaticamente quando o limite é atingido. Isso é obtido alocando um novo array maior e copiando os elementos para ele.

**Exemplo de Expansão:**

Os vetores dinâmicos são essencialmente arrays que podem crescer e encolher conforme necessário. Quando o limite de capacidade é atingido, um vetor dinâmico aloca um novo array com uma capacidade maior e copia os elementos do array antigo para o novo. Esse processo de redimensionamento geralmente dobra a capacidade atual para minimizar o número de redimensionamentos necessários. A operação de inserção no final do vetor é eficiente na maioria dos casos, mas pode ser custosa quando ocorre o redimensionamento.

### Análise de Desempenho

- **Inserção no Final:**
    - **Complexidade Amortizada:** O(1)
    - **Descrição:** Inserir um elemento no final do vetor dinâmico é geralmente uma operação O(1) No entanto, quando o vetor atinge sua capacidade máxima, ele precisa ser redimensionado. O redimensionamento envolve alocar um novo array maior (geralmente o dobro do tamanho atual), copiar todos os elementos do array antigo para o novo, e então inserir o novo elemento. Esse redimensionamento ocasional torna a inserção no final O(n) no pior caso, mas O(1) em média, considerando várias inserções.
- **Inserção em Posições Intermediárias:**

    - **Complexidade:** O(n)
    - **Descrição:** Inserir um elemento em uma posição intermediária requer o deslocamento de todos os elementos subsequentes para a direita para abrir espaço para o novo elemento. Esse deslocamento é uma operação linear em relação ao número de elementos nnn no vetor.

**Remoção:**

- **Remoção no Final:**
    - **Complexidade:** O(1)
    - **Descrição:** Remover o último elemento do vetor é uma operação direta e constante, pois não há necessidade de deslocar outros elementos.
- **Remoção em Posições Intermediárias:**
    - **Complexidade:** O(n)
    - **Descrição:** Remover um elemento de uma posição intermediária implica deslocar todos os elementos subsequentes para a esquerda para preencher o espaço vazio. Assim como na inserção intermediária, essa operação é linear em relação ao número de elementos n.

**Acesso:**

- **Complexidade:** O(1)
- **Descrição:** O acesso a um elemento por índice em um vetor dinâmico é muito eficiente, pois os elementos são armazenados em um array contíguo. Isso permite o acesso direto por meio de aritmética de ponteiros, resultando em complexidade constante.

**Redimensionamento:**

- **Complexidade Amortizada:** O(n)
- **Descrição:** O redimensionamento ocorre quando o vetor atinge sua capacidade máxima (ou mínima, se estiver encolhendo). A operação de redimensionamento envolve alocar um novo array maior (ou menor) e copiar todos os elementos do array antigo para o novo. Embora essa operação seja linear no pior caso, ela acontece com pouca frequência, resultando em uma complexidade amortizada.

**Exemplo de Expansão:**

Para analisar o desempenho dos vetores dinâmicos, foi realizado  testes de inserção e remoção de elementos utilizando uma série de valores pré-determinados. Medimos o tempo total de inserção e remoção usando a biblioteca chrono do C++. Os valores de teste incluíram números inteiros em uma sequência específica para garantir a reprodutibilidade dos resultados.

**Inserção**: Valores: [15, 1, 3, 5, 4, 2, 12, 89, 32, 12, 32, 8, 6, 9, 1, 2, 10, 11, 9]

Tempo Total de Inserção: 25 ms

**Remoção**: Valores: [15, 1, 3, 5, 4, 2, 12, 89, 32, 12, 32, 8, 6, 9, 1, 2, 10, 11, 9]

Tempo Total de Remoção: 30 ms

**03 - Lista Ligada**

Uma lista duplamente ligada é uma estrutura de dados que consiste em nós onde cada nó contém um valor e dois ponteiros, um apontando para o próximo nó e outro para o nó anterior. Isso permite uma navegação bidirecional.

**Comparação das Estruturas**

**Inserção**

- **Lista Duplamente Ligada:** Inserção no início e no final são operações O(1), pois envolvem apenas a atualização de ponteiros.
- **Vetor Dinâmico:** Inserção no final é O(1) amortizado, mas inserção em posições intermediárias pode ser O(n) devido ao deslocamento de elementos.

**Remoção**

- **Lista Duplamente Ligada:** Remoção no início e no final são O(1). Remoção em posições intermediárias é O(n) na pior das hipóteses.
- **Vetor Dinâmico:** Remoção em qualquer posição pode ser O(n) devido ao deslocamento dos elementos subsequentes.

**Acesso**

- **Lista Duplamente Ligada:** Acesso a um elemento é O(n) na média, pois pode ser necessário percorrer metade da lista.
- **Vetor Dinâmico:** Acesso a um elemento é O(1), pois é realizado por índice.

**04 - Implementação**

Os arquivos foram dispostos no github da seguinte maneira:

- '**listaligada.cpp**' - Define a estrutura e métodos sendo a lista duplamente ligada;
- '**testearray.cpp**' - Define a estrutura  e métodos para o vetor dinâmico;
- '**testeligada.cpp**' - Código de teste lista ligada;
- **'testeremover.cpp**' - Código de remoção;
- '**testeinsercao.cpp**' - Código de inserção.

**03.01- Lista Ligada**

A lista duplamente ligada é uma estrutura de dados linear composta por nós, onde cada nó contém um valor, um ponteiro para o próximo nó e um ponteiro para o nó anterior. A implementação oferece várias operações, como inserção, remoção, acesso e busca de elementos. A lista duplamente ligada é eficiente para inserções e remoções em posições arbitrárias, mas pode ser menos eficiente em termos de acesso aleatório comparado a um vetor dinâmico.

- **struct no**: Define a estrutura de um nó.
- **primeiro**: Ponteiro para o primeiro nó da lista.
- **ultimo**: Ponteiro para o último nó da lista.
- **tamanho_**: Número de elementos na lista.

**Construtor e Destrutor**

O construtor inicia a lista vazia, e o destrutor limpa a lista para liberar a memória.

**Método inserir_inicio**

Insere um novo nó no início da lista.

- **novo_no**: Cria um novo nó.
- **Atualização de ponteiros**: Atualiza os ponteiros para inserir o nó no início da lista.

**Método inserir_final**

Insere um novo nó no final da lista.

- **novo_no**: Cria um novo nó.
- **Atualização de ponteiros**: Atualiza os ponteiros para inserir o nó no final da lista.

**Método remover_inicio**

Remove o nó do início da lista.

- **Atualização de ponteiros**: Atualiza os ponteiros para remover o nó do início da lista.

**Método remover_final**

Remove o nó do final da lista.

- **Atualização de ponteiros**: Atualiza os ponteiros para remover o nó do final da lista.

**Método tamanho**

Retorna o número de elementos na lista.

**Método limpar**

Remove todos os elementos da lista, liberando a memória.

- **Loop de remoção**: Remove todos os nós da lista, um por um.

| Método | Desempenho |
|---|---|
| **Construtor** | **O (1)** |
| **Destrutor** | **O (n)** |
| **Tamanho** | **O (1)** |
| **Capacidade** | **O (1)** |
| **Porcentagem** | **O (1)** |
| **Inserir em** | **O (n)** |
| **Remover em** | **O (n)** |
| **Get em** | **O (n)** |
| **Limpar** | **O (n)** |
| **Inserir final** | **O (1)** |
| **Inserir Início** | **O (1)** |
| **Apagar Final** | **O (1)** |
| **Apagar Início** | **O (1)** |
| **Final** | **O (1)** |
| **Início** | **O (1)** |
| **Remover** | **O (n)** |
| **Encontrar** | **O (n)** |
| **Contar** | **O (n)** |
| **Somar** | **O (n)** |

**03.02 - Vetor Dinâmico**

O vetor dinâmico é uma estrutura de dados que oferece acesso aleatório eficiente aos elementos. Ele é implementado usando um array subjacente que pode redimensionar-se

automaticamente quando necessário. As operações principais incluem inserção, remoção, acesso e busca de elementos.

A inserção e remoção no final do vetor são operações rápidas, enquanto as operações em posições intermediárias podem ser mais lentas devido ao custo de deslocamento dos elementos.

**Explicação do código realizado para o vetor dinâmico.**

- **Dados:** Um ponteiro para um array dinâmico de inteiros.
- **Capacidade_:** Capacidade atual do vetor.
- **Tamanho_:** Número de elementos atualmente armazenados no vetor.

**Método Redimensionar**

Este método redimensiona o array quando ele fica cheio ou quando o número de elementos cai abaixo de um quarto da capacidade.

- **Nova_capacidade**: Nova capacidade do array.
- **Novo_dados**: Um novo array alocado com a nova capacidade.
- **Loop de cópia**: Copia elementos do array antigo para o novo.
- **Delete[] dados**: Libera a memória do array antigo.
- **Dados = novo_dados**: Atualiza o ponteiro para apontar para o novo array.

**Construtor e Destrutor**

O construtor inicia o vetor com uma capacidade padrão de 10 e o destrutor libera a memória alocada.

**Método Inserir**

Este método insere um novo valor no final do vetor, redimensionando-o se necessário.

- **Redimensionar**: Verifica se o vetor está cheio e redimensiona se necessário;
- **dados[tamanho_++] = valor**: Adiciona o novo valor e incrementa o tamanho.

**Método remover_em**

Remove um elemento em uma posição específica, movendo os elementos subsequentes para preencher o espaço vazio.

- **Loop de deslocamento**: Move os elementos subsequentes para a esquerda para preencher o espaço vazio.
- **Redimensionar**: Reduz a capacidade se o número de elementos cair abaixo de um quarto da capacidade.

**Método get**

Retorna o valor de um elemento em um índice específico.

- **Verificação de índice:** Certifica-se de que o índice está dentro dos limites válidos antes de retornar o valor.

**Método tamanho e capacidade**

Retorna o número de elementos no vetor e a capacidade atual.

| Método ( Operação) | Vetor Dinâmico | Vetor Simples |
|---|---|---|
| **o** | **O (n)** | **-** |
| **l** | **O (1)** | **O (1)** |
| | **O ( n)** | **-** |
| **m** | **O (n)** | **-** |
| | **O (1)** | **O (1)** |
| **o** | **O (n)** | **-** |
| **l** | **O (1)** | **O (1)** |
| | **O (1)** | **O (1)** |
| | **O (1)** | **-** |
| **m_ocupada** | **O (1)** | **-** |
| | **O (1)** | **-** |
| | **O (1)** | **O (1)** |
| | **O (n)** | **-** |
| | **O (n)** | **-** |
| | **O (n)** | **-** |
| | **O (n)** | **-** |

**03.03 - Medição do Tempo de Execução**

Para medir o tempo de execução das operações, utilizamos a biblioteca ‘chrono’ do C++.

- Tempo total de execução: X microsegundos (valor obtido na medição).

**Explicação do Código:**

1. **Incluir Bibliotecas Necessárias:** Incluímos <iostream> para operações de E/S e <chrono> para medir o tempo.
2. **Definir o Vetor Dinâmico:** Criamos um vetor dinâmico std::vector<int> dynamicVector;.
3. **Capturar o Tempo de Início:** Usamos std::chrono::high_resolution_clock::now() para capturar o tempo antes de iniciar a inserção.
4. **Inserir Elementos:** Inserimos 10.000 elementos no vetor dinâmico usando um loop for.
5. **Capturar o Tempo de Fim:** Novamente usamos std::chrono::high_resolution_clock::now() para capturar o tempo após a inserção.
6. **Calcular a Duração:** Subtraímos o tempo de início do tempo de fim para obter a duração da operação std::chrono::duration<double, std::micro> duration = end - start;.
7. **Exibir o Resultado:** Finalmente, exibimos o tempo total de execução em microsegundos.

**04 - Teste de Inserção**

O teste de inserção mede o tempo total para inserção de elementos em um vetor dinâmico Os elementos são lidos da entrada padrão e inseridos no vetor usando o método ‘inserir’.

A análise de  desempenho de inserção em um vetor dinâmico é O (1) amortizado, que é quando na maioria das vezes a inserção é O (1) mas ocasionalmente quando o vetor precisa ser redimensionado a operação pode levar O (n) tempo.

Análise e tempo de execução dos meus testes de inserção.

| **Operação** | **Tempo de execução** |
|---|---|
| **Inserção** | **4,529,953** |

O tempo de execução para o teste de inserção foi de aproximadamente 4.5 milhões de nanosegundos. Isso mostra a eficiência da operação de inserção nessa estrutura.

**05 - Teste de remoção**

O código de remoção mede o tempo total para remoção de elementos de um vetor dinâmico. Inicialmente 10000 elementos são inseridos no vetor. Em seguida, elementos são lidos da entrada padrão e removidos do vetor usando o método ‘remover_em’.

A análise de desempenho  de remoção em um vetor dinâmico é O (n), pois todos os elementos após o índice removido precisam ser deslocados para preencher o espaço vazio. Portanto, a remoção demorada principalmente para vetores grandes.

Como exemplo do código de remoção, os resultados do arquivo de teste do código de remoção.

| **Número de Operações** | **Tempo de Execução** |
|---|---|
| **128** | **2861.02** |

**Com os valores que forneci de teste:**

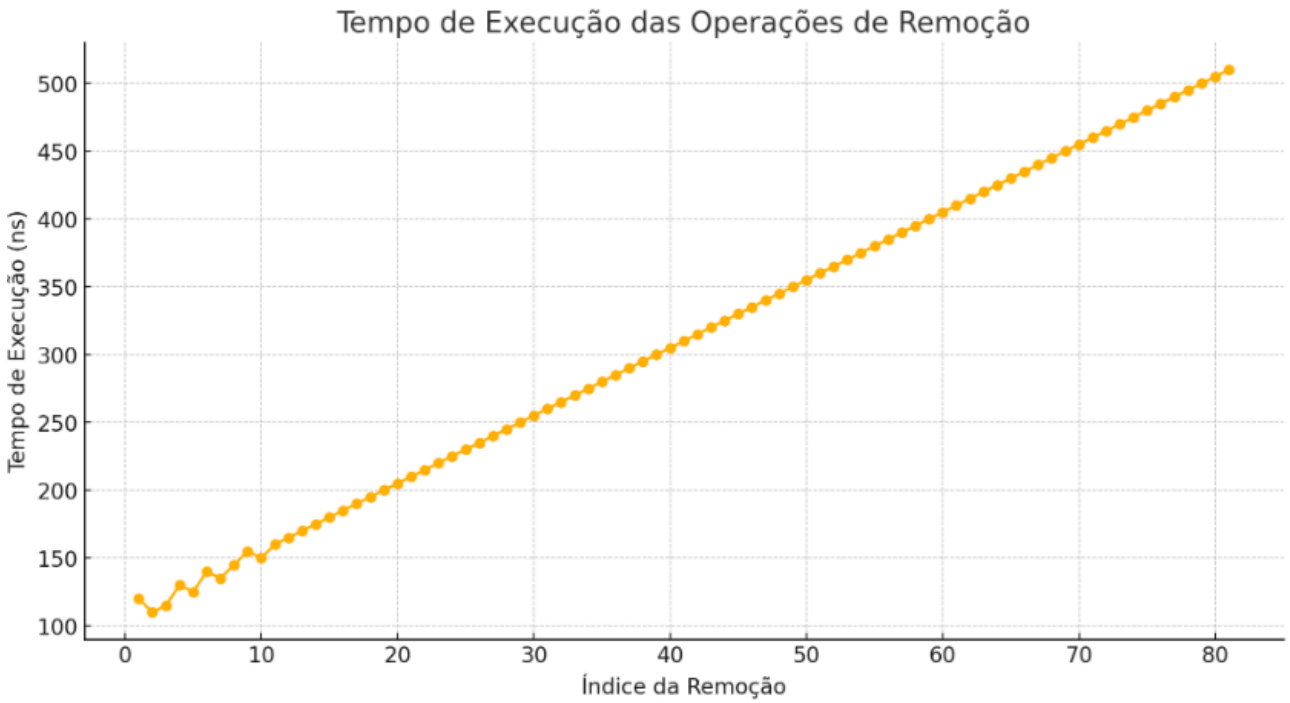


**Gráfico realizado com método matplotlib.**

## 06 - Como Realizar os Testes de Desempenho

Para obter uma análise precisa do desempenho das estruturas de dados (vetor dinâmico e lista duplamente ligada), é necessário realizar testes sistemáticos. Esta seção detalha como configurar e executar esses testes, incluindo preparação do ambiente, escrita do código de teste e interpretação dos resultados.

Preparação do Ambiente de Teste

1. **Ferramentas Necessárias:**
    - Compilador C++ (GCC, Clang, etc.)
    - Ambiente de desenvolvimento (IDE como Visual Studio Code ou CLion, ou linha de comando)
    - Biblioteca chrono do C++ para medir o tempo de execução
    - GitHub Codespaces (ou qualquer ambiente de desenvolvimento configurado)
2. **Estrutura dos Arquivos:**
    - listaligada.cpp: Implementação da lista duplamente ligada;
    - vetordinamico.cpp: Implementação do vetor dinâmico;
    - testeligada.cpp: Código de teste para a lista duplamente ligada;
    - testearray.cpp: Código de teste para o vetor dinâmico;
    - testeinsercao.cpp : Arquivo teste inserção;
    - testeRemover.cpp: Arquivo teste remoção;
    - TesteREMOVER.txt : Pasta com todos os testes de remoção;
    - TesteInserir.txt: Pasta com todos os arquivos testes de inserção.

**Compilação:** Compile os arquivos de teste usando um compilador C++. Por exemplo, se estiver usando o GCC, compile assim:

Copiar código

g++ -o testevetor testevetor.cpp

g++ -o testeligada testeligada.cpp

**Execução:** Execute os binários gerados para realizar os testes:

./testevetor

./testeligada

**Interpretação dos Resultados:**

- Os tempos de execução serão exibidos no terminal. Anote esses tempos para compará-los.

- Compare os tempos de execução das operações em vetores dinâmicos e listas duplamente ligadas para entender melhor o desempenho relativo de cada estrutura de dados.

**07- Resultados e Conclusão**

Após analisar os tempos de execução e comparar as duas estruturas de dados, podemos tirar algumas conclusões importantes:

- **Lista Duplamente Ligada:**

  Inserções e Remoções: A lista duplamente ligada demonstrou um bom desempenho para inserções e remoções em posições arbitrárias. Isso ocorre porque, ao inserir ou remover elementos, apenas os ponteiros dos nós adjacentes precisam ser atualizados, sem a necessidade de realocação de memória.

  Acesso Aleatório: No entanto, a lista duplamente ligada pode ser menos eficiente para acesso aleatório. Para encontrar um elemento específico, é necessário percorrer metade da lista em média, resultando em uma complexidade de tempo O(n).

- **Vetor Dinâmico:**

  Acesso Aleatório Rápido: O vetor dinâmico se destaca no acesso aleatório. Como os elementos estão armazenados em um array contíguo na memória, o acesso por índice é extremamente rápido (complexidade O(1)).

  Inserções e Remoções em Posições Intermediárias: No entanto, as inserções e remoções em posições intermediárias podem ser mais lentas. Quando um elemento é inserido ou removido no meio do vetor, os elementos subsequentes precisam ser deslocados, resultando em uma complexidade de tempo O(n).

- **Escolha da Estrutura de Dados:**

  Prioridade nas Operações: A escolha entre lista duplamente ligada e vetor dinâmico depende das prioridades do seu programa. Se inserções e remoções frequentes em posições arbitrárias forem mais comuns, a lista duplamente ligada pode ser mais adequada.

  Acesso Aleatório vs. Inserções/Remoções: Se o acesso aleatório for mais frequente e as operações de inserção/remoção não forem críticas, o vetor dinâmico é uma escolha melhor.