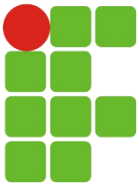


INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DO RIO GRANDE DO NORTE

Tecnologia em Análise e Desenvolvimento de Sistemas

Implementação de Vetores Dinâmicos

Utilizando a Linguagem de programação C++

Disciplina: Algoritmos

Professor: Jorgiano Vidal

Aluna: Tâmara Thais

2025

Resumo do projeto

Este relatório explica a criação, implementação e análise de desempenho de duas estruturas de dados: vetores dinâmicos com alocação de memória e listas duplamente ligadas. As implementações mostram diferentes formas de manipular vetores dinâmicos, como a alocação de memória de forma linear e exponencial, além de trabalhar com listas duplamente ligadas, que tornam mais rápidas as operações de inserção e remoção.

1. Introdução

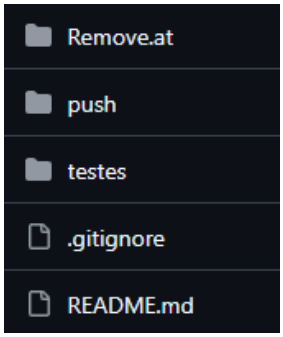
Vetores dinâmicos, também conhecidos como arrays ou listas dinâmicas, são estruturas de dados que permitem o armazenamento de coleções de elementos com tamanho variável, ao contrário dos vetores estáticos, que possuem um tamanho fixo. Essa característica confere maior flexibilidade, pois o tamanho do vetor pode ser ajustado conforme as necessidades do programa, sem a exigência de um tamanho máximo predefinido. Tais estruturas são particularmente eficientes para o acesso direto aos elementos por meio de índices e para a iteração sequencial dos dados.

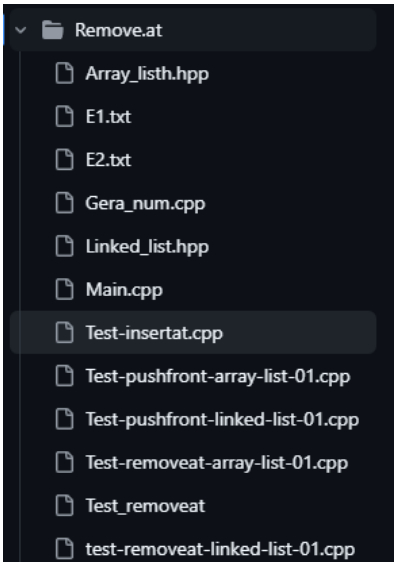
Na linguagem C++, a implementação de vetores dinâmicos é realizada por meio da alocação dinâmica de memória, permitindo que seu tamanho seja alterado durante a execução do programa.

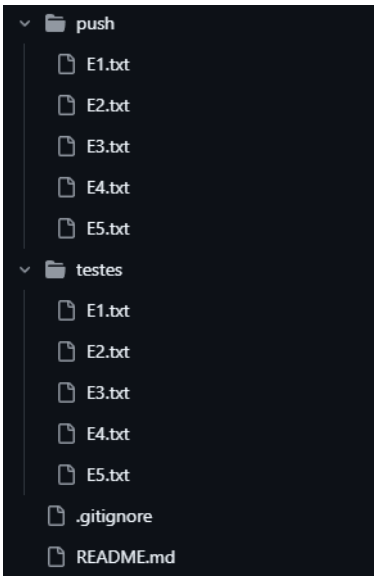
As listas duplamente ligadas, por sua vez, estruturam os elementos em nós conectados, o que facilita operações de inserção e remoção. No entanto, o custo para o acesso aleatório aos elementos é maior, uma vez que é necessário percorrer a lista de forma sequencial para localizar um nó específico.

O objetivo deste trabalho é desenvolver, utilizando a linguagem C++, uma biblioteca para vetores dinâmicos com alocação dinâmica de memória, bem como uma para listas duplamente ligadas. Além disso, será realizada uma análise comparativa de eficiência entre as duas implementações, utilizando a notação Big-O.

Estrutura do Repositório nas imagens abaixo:







Explicação do repositório:

1. Remove.at/

Esta pasta contém o arquivo array_list.hpp, onde está implementada a classe **ArrayList**. Essa classe é responsável por gerenciar um vetor dinâmico utilizando alocação de memória dinâmica, permitindo operações como inserção, remoção e acesso a elementos.

2. linked_list/

A pasta Remove.at também contém o arquivo linked_list.hpp, que implementa a classe **LinkedList**. Essa classe utiliza uma estrutura de nós duplamente ligados para oferecer maior eficiência em operações como inserção e remoção em posições arbitrárias.

3. testes/

A pasta testes contém os arquivos de teste para validar as implementações das classes **ArrayList** e **LinkedList**. Os testes incluem operações como:

- Inserção no início e final.
- Remoção por índice.
- Outros cenários necessários para validar a "veracidade" das operações.

4. Arquivo .gitignore

Este arquivo define os padrões de arquivos e pastas que devem ser ignorados pelo sistema de controle de versão Git. Isso ajuda a evitar o rastreamento de arquivos desnecessários no repositório.

5. README.md

O arquivo README.md fornece informações sobre o projeto, explicando como configurar e executar os testes, além de descrever o propósito e as funcionalidades implementadas

Implementação:

Criação das funções(1): *Arrays com alocação dinâmica*

Em C++, usamos o operador (new) para alocar memória dinamicamente. Isso permite criar arrays cujo tamanho pode ser definido em tempo de execução. As funções implementadas a seguir, seguem a lógica do operador (new) e são fundamentais para gerenciar a memória dinamicamente em C++.

Exemplo: *int* arr = new int[size];*

- *void increase_capacity(){}*

Essa função: aumenta o tamanho de um vetor dinâmico quando ele atinge sua capacidade máxima.

Sua eficiência: Complexidade o(n), onde (n) é o número de elementos no vetor, o algoritmos percorre todos os elementos do vetor afim de copiar os elementos antigos para a nova lista.

● *arraydinamico(){}*

Essa função: O Construtor, em C++ é uma função especial de membro de uma classe que é chamada automaticamente quando um objeto da classe é criado. Nesse caso seu principal objetivo é definir os parâmetros.

Sua eficiência: Na implementação em questão, o construtor não realiza operações complexas e apenas inicializa variáveis, o big OH é O(1).

● *~arraydinamico(){}*

Essa função: O destrutor é chamado automaticamente quando um objeto da classe é destruído. Seu principal objetivo é liberar qualquer recurso alocado dinamicamente pelo objeto e realizar a liberação da memória.

Sua eficiência: Destruidor simples com a complexidade constante O(1).

● *size(){}*

Essa função: a função em questão retorna o número de elementos armazenados na lista.
Sua eficiência: Complexidade: O(1), porque a função simplesmente retorna um valor armazenado.

● *capacity(){}*

Essa função: Retorna a quantidade de espaços de memórias reservados para a lista. Sua eficiência: Assim como size(), Complexidade: O(1), porque a função simplesmente retorna um valor armazenado.

● *double percent_occupied(){}*

Essa função: Retorna um número entre 0.0 e 1.0 correspondente a porcentagem de capacidade da lista utilizada até então.

Sua eficiência: Complexidade O(1), porque a função acessa variáveis(size_ e capacity_) e realiza uma simples operação aritmética.

● *bool insert_at(int index, int value)(){}*

Essa função: Recebe um índice e um valor. Ela verifica se o índice é válido (ou seja, não está fora dos limites). Se o índice for válido, a função insere o valor no índice recebido e retorna true(por ser uma função do tipo booleana). Se o índice for inválido, a função retorna false. Sua eficiência: Inserir um elemento no início do vetor é o pior caso, pois todos os elementos precisam ser deslocados da posição que estão para a da frente, para que deste modo o elemento desejado seja inserido, como a quantidade de elementos da lista é = n. A complexidade é O(n).

● *bool remove_at(int unsigned index){}*

Essa função: Verifica se o índice fornecido está dentro dos limites do vetor. Se o índice for inválido (ou seja, maior ou igual a capacity), a função retorna false(por ser do tipo booleana). Se o índice for válido, a função remove o elemento no índice especificado. Todos os elementos à direita do índice de remoção precisam ser deslocados para a esquerda, e size se torna size + 1, para que assim o último elemento não seja exibido. A função então retorna true para indicar que a remoção foi bem-sucedida.

Sua eficiência: A primeira parte verifica se o index digitado como parâmetro é válido, a complexidade dessa primeira parte da função O(1). entretanto na segunda parte da função, se o elemento removido estiver no início do vetor, todos os elementos posteriores ao index, precisam ser deslocados uma posição para trás. Portanto, a complexidade no pior caso é O(n), onde n é o número de elementos no vetor.

● *void push_back(int value){}*

Essa função: A função insere um novo elemento no final do vetor, dessa forma aumentando o tamanho dele, contudo, antes de realizar essa operação, ele verifica se a quantidade de elementos(size_) é igual ou maior que a capacidade do vetor(capacity_), caso seja, a função aumenta a capacidade do vetor utilizando a função increase_capacity(){}.

Sua eficiência: Quando há espaço suficiente na capacidade alocada do vetor, inserir um elemento no final é uma operação O(1), pois não há necessidade de realocar ou copiar elementos. Contudo, no pior caso dessa função, quando a capacidade do vetor é atingida, uma nova alocação de memória é necessária, fazendo com que todos os elementos sejam copiados para uma nova lista com a capacidade(capacity_) aumentada, tornando a complexidade desse algoritmo O(n).

● *void push_front(int unsigned value){}*

Essa função: Adiciona um valor no início de um vetor, Para manter a ordem dos elementos, todos os elementos existentes no vetor precisam ser deslocados uma posição para a direita. Sua eficiência: No pior caso, se o vetor tiver n elementos, todos os elementos precisam ser deslocados. Portanto, a complexidade de tempo para essa função é O(n).

● *int get_at(int unsigned index){}*

Essa função: Acessa o valor no índice especificado, A função recebe um índice e retorna o valor do elemento no vetor que está nesse índice.

Sua eficiência: Acessar um elemento em um vetor por seu índice é uma operação de tempo constante, pois com vetores conseguimos acessar o elemento diretamente pelo seu índice, assim como em arrays, portanto a complexidade desse algoritmo é O(1).

● *bool pop_back(){}*

Essa função: Remove o último item do vetor, após isso o tamanho do vetor é diminuído em uma unidade. por ser do tipo boleana, a função pode retornar false para indicar que a operação não foi bem-sucedida, caso o vetor inicialmente esteja vazio.

Sua eficiência: A complexidade de tempo para essa função é O(1), pois a operação remove o último elemento de um vetor de forma eficiente, sem precisar realocar nada, apenas acessando a variável tamanho(size_).

● *bool pop_front(){}*

Essa função: Remove o primeiro elemento do vetor. Após a remoção, todos os elementos restantes precisam ser deslocados uma posição para a esquerda para preencher a lacuna deixada pelo primeiro elemento, após isso a variável correspondente ao tamanho do vetor(size_) é diminuída em uma unidade, para que o último elemento do vetor não seja exibido, por ser do tipo booleano, caso o vetor esteja vazio, a função retornará false

Sua eficiência: A complexidade de tempo para essa função é O(n), sendo n a quantidade de elementos presentes no vetor, pois a operação requer o deslocamento de todos os elementos restantes uma posição para a esquerda.

● *int back(){}*

Essa função: Acessa e retorna o valor do último elemento do vetor, através da variável de tamanho(size_).

Sua eficiência: A complexidade desse algoritmo é O(1), pois a operação de acessar o último elemento de um vetor é uma operação de tempo constante.

●*int front(){}*

Testes e resultado:

Foram realizados testes com as funções **push_front()** e **remove_at()**.

Esses testes, consistiram em solicitar ao programa que executasse cada função diversas vezes e, em seguida, medir o tempo necessário para sua execução.

O primeiro experimento foi conduzido com a função **push_front()**. A quantidade de chamadas feitas à função aumentou progressivamente durante os testes, sendo assim, foram realizadas 5, 10, 1000, 2000 e 3000 chamadas, respectivamente.

O resultado obtido demonstra um comportamento linear, caracterizado pela complexidade **O(n)**, conforme ilustrado no gráfico abaixo:

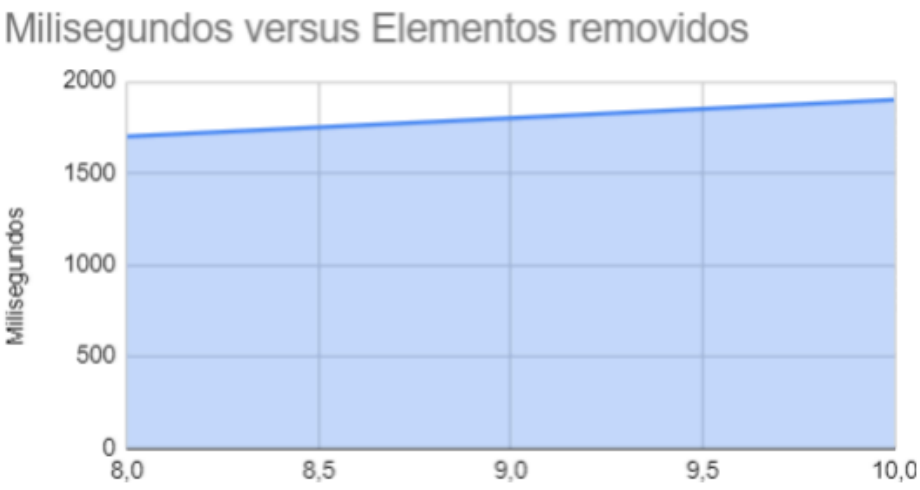
Gráfico 1:

TEMPO DE EXECUÇÃO	
Vezes que o push_front foi feito	Milisegundos
5	1700
10	3900
1000	1294600
2000	4289900
3000	9930700

O segundo teste foi realizado com a função remove_at(){}, a função foi chamada para remover 8 e 10 elementos respectivamente, o gráfico resultante também é O(n).

Gráfico 2:

TEMPO DE EXECUÇÃO	
Elementos removidos	Milisegundos
8	1700
10	1900



- **Compilador utilizado**: GCC (GNU Compiler Collection) com suporte a C++17.
- **Ferramentas auxiliares**: Editor de texto como Visual Studio Code ou CLion e o sistema de controle de versão Git.

Conclusão de projeto:

Neste relatório, foram analisadas as operações de manipulação de dados em um vetor dinâmico, implementado na classe arraydinamico.hpp. O foco principal foi nas operações de inserção (push_front) e remoção (remove_at), avaliando o tempo de execução e a integridade dos dados após cada operação.

Os testes realizados confirmam que vetor dinâmico é eficiente em cenários que exigem redimensionamento dinâmico da memória, com um comportamento linear (O(n)) para inserções e remoções em posições intermediárias ou no início do vetor. Esse comportamento é esperado devido ao deslocamento dos elementos subsequentes em cada operação.

Concluindo, essa análise confirma que a implementação na classe arraydinamico.hpp cumpre os requisitos de funcionalidade, garantindo tanto a correta manipulação dos dados quanto um bom desempenho nas operações fundamentais. A flexibilidade do vetor dinâmico em ajustar seu tamanho durante a execução reforça sua utilidade em aplicações que lidam com dados de tamanho variável.