INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
DO RIO GRANDE DO NORTE
Tecnologia em Análise e Desenvolvimento de Sistemas

Jesrriel Moura Lopes

Implementação de Vetores Dinâmicos
Utilizando a Linguagem C++

NATAL
07/2024

Jesrriel Moura Lopes

Implementação de Vetores Dinâmicos
Utilizando a Linguagem C++

Trabalho apresentado à disciplina de algoritmos, correspondente a avaliação do 2º período da Graduação em Tecnologia em Análise e Desenvolvimento de Sistemas do Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Norte com a finalidade de desenvolver um relatório de Implementação de Vetores Dinâmicos em C + +.

Docente responsável: Jorgiano Márcio Bruno Vidal

Maio, 2023

1. Introdução

Vetores dinâmicos, arrays dinâmicos ou listas dinâmicas, São estruturas de dados que possibilitam o armazenamento de uma coleção de elementos com tamanho flexível. Essas estruturas podem diminuir ou se expandir conforme a necessidade durante a execução do programa, diferentemente dos vetores estáticos, que contém um tamanho fixo.

Essa característica proporciona uma flexibilidade adicional, já que não se faz necessário especificar um tamanho máximo para o vetor de forma prévia, possibilitando uma adaptação mais ágil de acordo com as necessidades do programa. Os vetores dinâmicos permitem acessar elementos diretamente usando índices e são particularmente eficientes quando é necessário acessar dados rapidamente e percorrê-los de forma sequencial.

Nesse contexto, a criação de vetores dinâmicos em C++ pode ser obtida através do uso de alocação dinâmica de memória, permitindo que o tamanho dos vetores seja ajustado conforme necessário durante a execução do programa.

As listas duplamente ligadas (ou encadeadas) oferecem uma abordagem diferente, onde os elementos são estruturados em nós conectados, permitindo inserções e remoções mais eficientes. No entanto, esse método resulta em um custo mais elevado para o acesso aleatório aos elementos, pois nesse método, cada elemento(ponteiros) aponta para o próximo e para o anterior, sendo necessário percorrer toda a lista para por exemplo chegar ao elemento do meio

As listas duplamente ligadas (ou encadeadas) oferecem uma abordagem diferente, onde os elementos são estruturados em nós conectados, permitindo inserções e remoções mais eficientes. No entanto, esse método resulta em um custo mais elevado para o acesso aleatório aos elementos, pois, nesse método, cada nó (ponteiro) aponta para o próximo e para o anterior, sendo necessário percorrer toda a lista para, por exemplo, chegar ao elemento do meio.

Com base no que foi apresentado, o objetivo deste trabalho é desenvolver, utilizando a linguagem de programação C, uma biblioteca para vetores dinâmicos com alocação dinâmica e outra para listas duplamente ligadas. Além disso, pretende-se analisar a eficiência de cada implementação, utilizando a notação Big-O como referência.

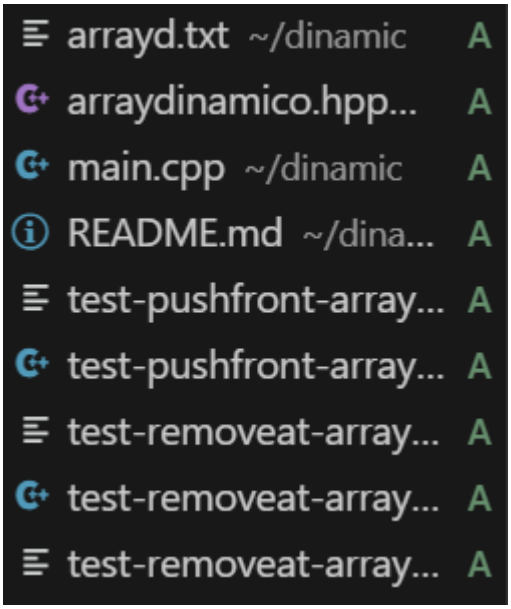
2. Implementação

2.1 Organização dos arquivos fontes

No projeto em questão, Existem 3 tipos de arquivos, o primeiro tipo são os arquivos onde as funções são definidas (main.cpp e linkedlist1.cpp); o segundo tipo de arquivos, são os de cabeçalho onde as funções dos vetores dinâmicos são declaradas, esse arquivo fica no mesmo diretório que os arquivos.cpp, (arraydinamico.hpp e linkedlist2.hpp); o último tipo de arquivos, são os de teste, onde serão testadas a eficiência das funções antes feitas.

Imagem 1.

Pastas e arquivos.



2.2 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(){}*

Essa função: Acessa e retorna o valor do primeiro elemento do vetor

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

- *bool remove(int value)(){}*

Essa função: Procura pelo valor especificado e remove o mesmo do vetor, caso esteja presente. Por ser uma função do tipo booleana, caso o valor não esteja presente, ele retorna False.

Sua eficiência: O algoritmo precisa percorrer todos os elementos do vetor afim de comparar todos os índices do vetor com o valor especificado, portanto sua complexidade O(n), onde *n* é o número de elementos no vetor.

- *int find(int value)() {}*

Essa função: Procura pelo valor especificado como parâmetro no algoritimo e retorna o índice (ou posição) onde o valor é encontrado.

Sua eficiência: Como ele percorre todos os elementos da lista, a complexidade O(n), onde nnn é o número de elementos no vetor.

- *int count(int value)(){}*

Essa função: Percorre o vetor comparando a lista na posição de todos os índices com o valor atribuído como parâmetro, e conta quantas vezes o valor especificado aparece.

Sua eficiência: realiza uma busca linear por todas as ocorrências do valor no vetor. Isso tem complexidade O(n), onde n é o número de elementos no vetor.

- *int sum()(){}*

Essa função: Percorre todos os elementos da estrutura de dados e calcula a soma total.

Sua eficiência: realiza uma busca linear através de todos os elementos do vetor para calcular a soma. Isso tem complexidade O(n), onde n é o número de elementos no vetor.

2.3 Testes e resultado

Os testes foram realizados em um notebook com as especificações:

- Sistema operacional: Windows 11 Home
- Processador: Intel Core i5-11300H
- Memória: 8,00 GB

Foram realizados testes com as funções: *push_front(); remove_at()*. os testes consistiam em pedir ao programa que executasse a função uma quantidade de vezes, e medir o tempo de execução da mesma.

O primeiro teste foi feito com a função *push_front(){},* a quantidade de vezes que a função é chamada foi crescendo à medida que os testes eram feitos, a função foi chamada, 5, 10, 1000, 2000, 3000 respectivamente, o gráfico desse resultado é linear O(n)

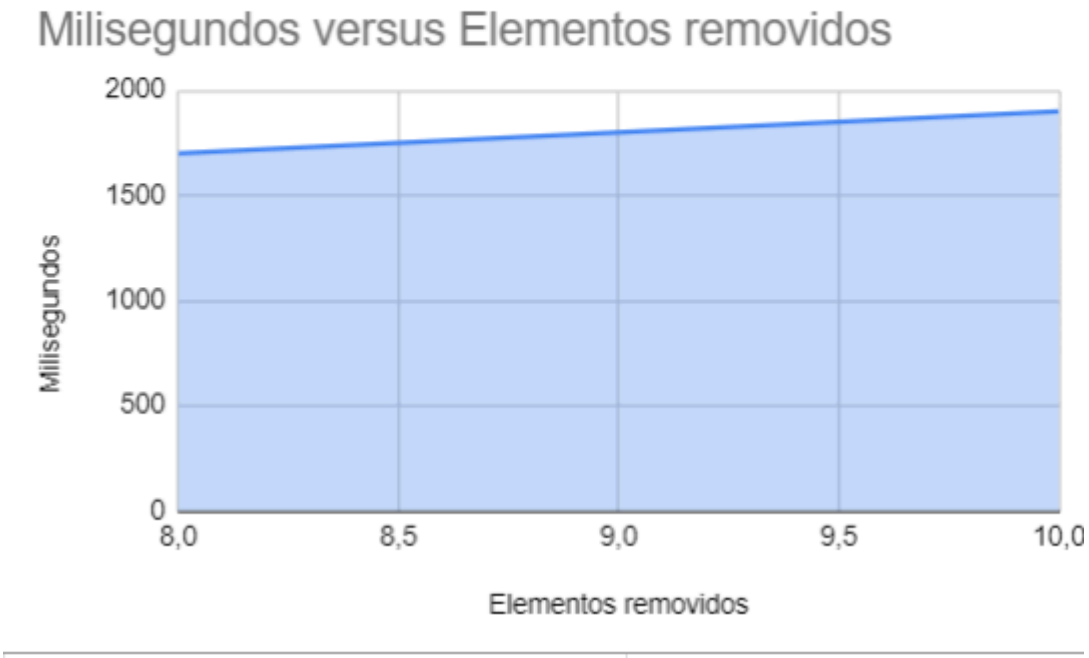
Gráfico 1

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



3. Conclusão

Neste relatório, analisamos a eficiência das operações de manipulação de dados em um vetor dinâmico implementado na classe *arraydinamico.hpp* Especificamente, focamos nas operações de inserção (*push_front*) e remoção (*remove_at*), medindo o tempo de execução e verificando a integridade das operações realizadas.

Em resumo, a avaliação realizada confirma a funcionalidade e eficiência das operações fundamentais na classe *arraydinamico*.