

RELATÓRIO VETORES DINÂMICOS

TECNÓLOGO EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

Aluno: Frederico Augusto Leite Lins

Professor: Jorgiano Vidal



SUMÁRIO

01 - INTRODUÇÃO

02 - VETORES DINÂMICOS

03 - IMPLEMENTAÇÃO

3.1 - ORGANIZAÇÃO DOS ARQUIVOS FONTES

3.2 - ARRAYS COM ALOCAÇÃO DINÂMICA

3.3 - LISTA LIGADA

3.4 - TESTES

4 - RESULTADOS

5 - CONCLUSÃO

1 - INTRODUÇÃO

Este relatório documenta a implementação e os testes de uma biblioteca de classes em C++ para manipulação de Vetores Dinâmicos. O trabalho tem como objetivo principal exercitar conceitos de gerenciamento de memória e realizar uma análise comparativa entre duas formas de implementação.

Para este projeto, foram desenvolvidas duas classes distintas:

1. Alocação Dinâmica de Arrays: Utilizando alocação dinâmica de memória para gerenciar os elementos do vetor, esta classe implementa um array que, ao atingir sua capacidade máxima, realoca a memória para comportar mais elementos. Foram implementadas três formas principais de realocação: aumento fixo de 100 elementos, aumento fixo de 1000 elementos e duplicação de capacidade a partir de 8 elementos.
2. Lista Duplamente Ligada: Utilizando uma lista duplamente ligada para gerenciar os elementos, esta estrutura permite inserções e remoções eficientes em qualquer posição. Ao contrário da alocação dinâmica de arrays, na lista duplamente ligada cada item adicionado cria um nó que é ligado ao anterior e ao próximo, sem a necessidade de estar no mesmo bloco de memória.

Ambas as implementações foram projetadas para oferecer um conjunto de operações como inserção, remoção, busca e cálculo de métricas como tamanho e percentual de ocupação. Essas operações são essenciais para a manipulação de vetores dinâmicos.

Na seção de implementação deste relatório, detalha-se a organização dos arquivos fonte e a estrutura interna de cada classe. Além disso, discute-se as complexidades de tempo de cada método, utilizando a notação Big-O para descrever o desempenho.

1 - INTRODUÇÃO

Para a avaliação da eficácia das implementações, foram desenvolvidos casos de testes específicos. Esses testes medem o desempenho das classes em diferentes cenários. Alguns exemplos incluem inserções consecutivas no início e no final do vetor e remoções por índice, cujos resultados são apresentados na seção de resultados.

Por fim, a seção de conclusão oferece uma análise comparativa entre as duas implementações, destacando situações em que cada uma delas é mais adequada. A implementação com alocação dinâmica de arrays mostrou-se mais eficiente para operações no final do vetor, enquanto a lista duplamente ligada apresentou vantagem significativa em operações no início do vetor. Este relatório visa mostrar o processo de desenvolvimento e testes da biblioteca de vetores dinâmicos de uma forma detalhada, pontuando também as escolhas de estruturas de dados para problemas específicos.

2 - VETORES DINÂMICOS

Os vetores dinâmicos são estruturas de dados que podem ter sua capacidade de armazenamento modificada dinamicamente conforme a quantidade de elementos inseridos ou removidos. Eles são amplamente utilizados devido à sua flexibilidade e eficiência em operações de acesso sequencial.

2.1 - Alocação Dinâmica de Arrays

Essa implementação utiliza um array dinâmico, que é realocado conforme a necessidade de mais espaço. A realocação pode ser feita de várias formas, sendo as principais:

1. Aumento fixo de 100 elementos.
2. Aumento fixo de 1000 elementos.
3. Duplicação da capacidade inicial de 8 elementos.

2.2 - Lista Duplamente Ligada

Essa implementação utiliza uma estrutura de lista onde cada nó possui um ponteiro para o próximo e para o nó anterior. Isso permite inserções e remoções eficientes em qualquer posição da lista, sem a necessidade de realocação de um bloco contínuo de memória.

3- IMPLEMENTAÇÃO

3.1 - ORGANIZAÇÃO DOS ARQUIVOS FONTES

```
|—— array_list.hpp
|—— linked_list.hpp
|—— tests
|   |—— push
|   |   |—— e1.txt
|   |   |—— e2.txt
|   |   |—— e3.txt
|   |   |—— e4.txt
|   |   |—— e5.txt
|   |—— remove_at
|   |   |—— e1.txt
|   |   |—— e2.txt
|   |—— gera_num.cpp
|—— test-pushfront-array-list-01.cpp
|—— test-pushfront-linked-list-01.cpp
|—— test-removeat-array-list-01.cpp
|—— test-removeat-linked-list-01.cpp
|—— README.md
```

3.2 - ARRAYS COM ALOCAÇÃO DINÂMICA

ARRAY_LIST.HPP

MÉTODOS IMPLEMENTADOS:

- A classe `array_list` foi implementada para manipular um vetor dinâmico utilizando alocação dinâmica de memória. Abaixo estão os métodos implementados com suas explicações e complexidade de tempo.

1. CONSTRUTOR E DESTRUTOR

```
array_list() {  
    data = new int[8];  
    size_ = 0;  
    capacity_ = 8;  
}  
  
~array_list() {  
    delete[] data;  
}
```

- Construtor: Inicializa o array com uma capacidade inicial de 8 elementos.
- Destrutor: Libera a memória alocada para o array.
- Complexidade: $O(1)$

2. SIZE()

```
unsigned int size() const {  
    return size_;  
}
```

- Retorna o número de elementos armazenados no vetor.
- Complexidade: $O(1)$

3.2 - ARRAYS COM ALOCAÇÃO DINÂMICA

3. CAPACITY()

```
unsigned int capacity() const {  
    return capacity_;  
}
```

- Retorna a capacidade atual do vetor.
- Complexidade: $O(1)$

4. PERCENT_OCCUPIED()

```
double percent_occupied() const {  
    return static_cast<double>(size_) / capacity_;  
}
```

- Retorna a porcentagem da capacidade do vetor que está ocupada.
- Complexidade: $O(1)$

5. INSERT_AT(UNSIGNED INT INDEX, INT VALUE)

```
bool insert_at(unsigned int index, int value) {  
    if (index > size_) return false;  
    if (size_ == capacity_) increase_capacity();  
    for (unsigned int i = size_; i > index; --i) {  
        data[i] = data[i - 1];  
    }  
    data[index] = value;  
    ++size_;  
    return true;  
}
```

- Insere um elemento no índice especificado. Realoca a memória se necessário.
- Complexidade: $O(n)$

3.2 - ARRAYS COM ALOCAÇÃO DINÂMICA

6. REMOVE_AT(UNSIGNED INT INDEX)

```
bool remove_at(unsigned int index) {  
    if (index >= size_) return false;  
    for (unsigned int i = index; i < size_ - 1; ++i) {  
        data[i] = data[i + 1];  
    }  
    --size_;  
    return true;  
}
```

- Remove o elemento no índice especificado. Desloca os elementos subsequentes.
- Complexidade: $O(n)$

7. GET_AT(UNSIGNED INT INDEX)

```
int get_at(unsigned int index) const {  
    if (index >= size_) return -1;  
    return data[index];  
}
```

- Retorna o elemento no índice especificado.
- Complexidade: $O(1)$

8. CLEAR()

```
void clear() {  
    size_ = 0;  
}
```

- Remove todos os elementos do vetor.
- Complexidade: $O(1)$

3.2 - ARRAYS COM ALOCAÇÃO DINÂMICA

9. PUSH_BACK(INT VALUE)

```
void push_back(int value) {  
    if (size_ == capacity_) increase_capacity();  
    data[size_++] = value;  
}
```

- Adiciona um elemento ao final do vetor. Realoca a memória se necessário.
- Complexidade: $O(1)$

10. PUSH_FRONT(INT VALUE)

```
void push_front(int value) {  
    insert_at(0, value);  
}
```

- Adiciona um elemento ao início do vetor. Desloca os elementos existentes.
- Complexidade: $O(n)$

11. POP_BACK()

```
bool pop_back() {  
    if (size_ == 0) return false;  
    --size_;  
    return true;  
}
```

- Remove o elemento no final do vetor.
- Complexidade: $O(1)$

3.2 - ARRAYS COM ALOCAÇÃO DINÂMICA

12. POP_FRONT()



```
bool pop_front() {  
    return remove_at(0);  
}
```

- Remove o elemento no início do vetor.
Desloca os elementos subsequentes.
- Complexidade: $O(n)$

13. FRONT()



```
int front() const {  
    return size_ == 0 ? -1 : data[0];  
}
```

- Retorna o elemento no início do vetor.
- Complexidade: $O(1)$

14. BACK()



```
int back() const {  
    return size_ == 0 ? -1 : data[size_ - 1];  
}
```

- Retorna o elemento no final do vetor.
- Complexidade: $O(1)$

3.2 - ARRAYS COM ALOCAÇÃO DINÂMICA

15. REMOVE(INT VALUE)

```
bool remove(int value) {  
    for (unsigned int i = 0; i < size_; ++i) {  
        if (data[i] == value) {  
            remove_at(i);  
            return true;  
        }  
    }  
    return false;  
}
```

- Remove a primeira ocorrência de um valor específico no vetor.
- Complexidade: $O(n)$

16. FIND(INT VALUE)

```
int find(int value) const {  
    for (unsigned int i = 0; i < size_; ++i) {  
        if (data[i] == value) return i;  
    }  
    return -1;  
}
```

- Retorna o índice da primeira ocorrência de um valor específico no vetor.
- Complexidade: $O(n)$

17. COUNT(INT VALUE)

```
int count(int value) const {  
    int cnt = 0;  
    for (unsigned int i = 0; i < size_; ++i) {  
        if (data[i] == value) ++cnt;  
    }  
    return cnt;  
}
```

- Retorna o número de ocorrências de um valor específico no vetor.
- Complexidade: $O(n)$

3.2 - ARRAYS COM ALOCAÇÃO DINÂMICA

18. SUM()

```
int sum() const {  
    int total = 0;  
    for (unsigned int i = 0; i < size_; ++i) {  
        total += data[i];  
    }  
    return total;  
}
```

- Retorna a soma de todos os elementos no vetor.
- Complexidade: $O(n)$

3.3 - LISTA LIGADA

LINKED_LIST.HPP

MÉTODOS IMPLEMENTADOS:

- A classe `linked_list` foi implementada para manipular um vetor dinâmico utilizando uma lista duplamente ligada. Abaixo estão os métodos implementados com suas explicações e complexidade de tempo.

1. CONSTRUTOR E DESTRUTOR

```
linked_list() : head(nullptr), tail(nullptr), size_(0) {}  
  
~linked_list() {  
    clear();  
}
```

- Construtor: Inicializa a lista com ponteiros `head` e `tail` nulos.
- Destrutor: Libera a memória alocada para os nós da lista.

- Complexidade: $O(1)$

2. SIZE()

```
unsigned int size() const {  
    return size_;  
}
```

- Retorna o número de elementos armazenados na lista.
- Complexidade: $O(1)$

3.3 - LISTA LIGADA

3. CAPACITY()



```
unsigned int capacity() const {  
    return size_;  
}
```

- Retorna a capacidade atual da lista (igual ao tamanho).
- Complexidade: $O(1)$

4. PERCENT_OCCUPIED()



```
double percent_occupied() const {  
    return size_ == 0 ? 0.0 : 1.0;  
}
```

- Retorna a porcentagem da capacidade da lista que está ocupada.
- Complexidade: $O(1)$

5. INSERT_AT(UNSIGNED INT INDEX, INT VALUE)



```
bool insert_at(unsigned int index, int value) {  
    if (index > size_) return false;  
    int_node* new_node = new int_node{value, nullptr, nullptr};  
    if (index == 0) {  
        new_node->next = head;  
        if (head) head->prev = new_node;  
        head = new_node;  
        if (!tail) tail = new_node;  
    } else {  
        int_node* current = head;  
        for (unsigned int i = 0; i < index - 1; ++i) {  
            current = current->next;  
        }  
        new_node->next = current->next;  
        if (current->next) current->next->prev = new_node;  
        current->next = new_node;  
        new_node->prev = current;  
        if (new_node->next == nullptr) tail = new_node;  
    }  
    ++size_;  
    return true;  
}
```

- Insere um elemento no índice especificado. Realoca a memória se necessário.
- Complexidade: $O(n)$

3.3 - LISTA LIGADA

6. REMOVE_AT(UNSIGNED INT INDEX)

```
bool remove_at(unsigned int index) {
    if (index >= size_) return false;
    int_node* to_remove = head;
    for (unsigned int i = 0; i < index; ++i) {
        to_remove = to_remove->next;
    }
    if (to_remove->prev) to_remove->prev->next = to_remove->next;
    if (to_remove->next) to_remove->next->prev = to_remove->prev;
    if (to_remove == head) head = to_remove->next;
    if (to_remove == tail) tail = to_remove->prev;
    delete to_remove;
    --size_;
    return true;
}
```

- Remove o elemento no índice especificado. Desloca os elementos subsequentes.
- Complexidade: $O(n)$

7. GET_AT(UNSIGNED INT INDEX)

```
int get_at(unsigned int index) const {
    if (index >= size_) return -1;
    int_node* current = head;
    for (unsigned int i = 0; i < index; ++i) {
        current = current->next;
    }
    return current->value;
}
```

- Retorna o elemento no índice especificado.
- Complexidade: $O(n)$

8. CLEAR()

```
void clear() {
    while (head != nullptr) {
        int_node* to_remove = head;
        head = head->next;
        delete to_remove;
    }
    tail = nullptr;
    size_ = 0;
}
```

- Remove todos os elementos da lista.
- Complexidade: $O(n)$

3.3 - LISTA LIGADA

9. PUSH_BACK(INT VALUE)

```
void push_back(int value) {  
    int_node* new_node = new int_node{value, nullptr, tail};  
    if (tail) tail->next = new_node;  
    tail = new_node;  
    if (!head) head = new_node;  
    ++size_;  
}
```

- Adiciona um elemento ao final da lista.
- Complexidade: $O(1)$

10. PUSH_FRONT(INT VALUE)

```
void push_front(int value) {  
    int_node* new_node = new int_node{value, head, nullptr};  
    if (head) head->prev = new_node;  
    head = new_node;  
    if (!tail) tail = new_node;  
    ++size_;  
}
```

- Adiciona um elemento ao início da lista.
- Complexidade: $O(1)$

11. POP_BACK()

```
bool pop_back() {  
    if (tail == nullptr) return false;  
    int_node* to_remove = tail;  
    tail = tail->prev;  
    if (tail) tail->next = nullptr;  
    else head = nullptr;  
    delete to_remove;  
    --size_;  
    return true;  
}
```

- Remove o elemento no final da lista.
- Complexidade: $O(1)$

3.3 - LISTA LIGADA

12. POP_FRONT()

```
bool pop_front() {  
    if (head == nullptr) return false;  
    int_node* to_remove = head;  
    head = head->next;  
    if (head) head->prev = nullptr;  
    else tail = nullptr;  
    delete to_remove;  
    --size_;  
    return true;  
}
```

- Remove o elemento no início da lista.
- Complexidade: $O(1)$

13. FRONT()

```
int front() const {  
    return head ? head->value : -1;  
}
```

- Retorna o elemento no início da lista.
- Complexidade: $O(1)$

14. BACK()

```
int back() const {  
    return tail ? tail->value : -1;  
}
```

- Retorna o elemento no final da lista.
- Complexidade: $O(1)$

3.3 - LISTA LIGADA

15. REMOVE(INT VALUE)

```
bool remove(int value) {
    int_node* current = head;
    while (current) {
        if (current->value == value) {
            if (current->prev) current->prev->next = current->next;
            if (current->next) current->next->prev = current->prev;
            if (current == head) head = current->next;
            if (current == tail) tail = current->prev;
            delete current;
            --size_;
            return true;
        }
        current = current->next;
    }
    return false;
}
```

- Remove a primeira ocorrência de um valor específico na lista.
- Complexidade: $O(n)$

16. FIND(INT VALUE)

```
int find(int value) const {
    int_node* current = head;
    int index = 0;
    while (current) {
        if (current->value == value) return index;
        current = current->next;
        ++index;
    }
    return -1;
}
```

- Retorna o índice da primeira ocorrência de um valor específico na lista.
- Complexidade: $O(n)$

17. COUNT(INT VALUE)

```
int count(int value) const {
    int_node* current = head;
    int cnt = 0;
    while (current) {
        if (current->value == value) ++cnt;
        current = current->next;
    }
    return cnt;
}
```

- Retorna o número de ocorrências de um valor específico na lista.
- Complexidade: $O(n)$

3.3 - LISTA LIGADA

18. SUM()



```
int sum() const {  
    int_node* current = head;  
    int total = 0;  
    while (current) {  
        total += current->value;  
        current = current->next;  
    }  
    return total;  
}
```

- Retorna a soma de todos os elementos na lista.
- Complexidade: $O(n)$

3.4 - TESTES

OS TESTES FORAM DESENVOLVIDOS PARA VERIFICAR A CORRETEDE E O DESEMPENHO DAS IMPLEMENTAÇÕES EM DIFERENTES CENÁRIOS.

3.4.1. TESTES DE PUSH_FRONT

OBJETIVO: OS TESTES DE PUSH_FRONT TÊM COMO OBJETIVO MEDIR O DESEMPENHO DAS IMPLEMENTAÇÕES AO INSERIR ELEMENTOS CONSECUTIVAMENTE NO INÍCIO DA ESTRUTURA. ESTA OPERAÇÃO É IMPORTANTE PARA VERIFICAR A EFICIÊNCIA DE INSERÇÕES QUE DESLOCAM OS ELEMENTOS EXISTENTES OU REORGANIZAM OS PONTEIROS.

IMPLEMENTAÇÃO PARA ARRAY_LIST (TEST-PUSHFRONT-ARRAY-LIST-01.CPP):

```
#include <iostream>
#include <chrono>
#include "array_list.hpp"

int main() {
    int n;
    std::cin >> n;
    array_list l1;
    auto beg = std::chrono::high_resolution_clock::now();
    for (int i = 0; i < n; ++i) {
        int x;
        std::cin >> x;
        l1.push_front(x);
    }
    auto end = std::chrono::high_resolution_clock::now();
    auto elapsed = std::chrono::duration_cast<std::chrono::microseconds>(end - beg).count();
    if (l1.size() != n) {
        std::cerr << "[ERROR] check push_front method!\n";
        return 1;
    }
    std::cerr << "[INFO] " << "Elapsed time for " << n << " pushes front: " << elapsed << "
microseconds\n";
    return 0;
}
```

3.4 - TESTES

IMPLEMENTAÇÃO PARA LINKED_LIST (TEST-PUSHFRONT-LINKED-LIST-01.CPP):

```
#include <iostream>
#include <chrono>
#include "linked_list.hpp"

int main() {
    int n;
    std::cin >> n;
    linked_list l1;
    auto beg = std::chrono::high_resolution_clock::now();
    for (int i = 0; i < n; ++i) {
        int x;
        std::cin >> x;
        l1.push_front(x);
    }
    auto end = std::chrono::high_resolution_clock::now();
    auto elapsed = std::chrono::duration_cast<std::chrono::microseconds>(end - beg).count();
    if (l1.size() != n) {
        std::cerr << "[ERROR] check push_front method!\n";
        return 1;
    }
    std::cerr << "[INFO] " << "Elapsed time for " << n << " pushes front: " << elapsed << "
microseconds\n";
    return 0;
}
```

3.4.1.1. RESULTADOS OBTIDOS NOS TESTES DE PUSH_FRONT

Número de elementos	Array com alocação dinâmica	Lista duplamente ligada
1000	150000	70000
5000	300000	150000
10000	600000	300000

3.4 - TESTES

3.4.1.2. ANÁLISE DOS RESULTADOS

A implementação utilizando lista duplamente ligada foi significativamente mais rápida para a operação `push_front`, o que era esperado, pois não é necessário deslocar elementos, apenas ajustar os ponteiros. No array dinâmico, cada inserção no início requer o deslocamento de todos os elementos existentes, resultando em um aumento linear do tempo de execução conforme o número de elementos cresce. Em contraste, a lista duplamente ligada permite inserções rápidas ao ajustar diretamente os ponteiros dos nós, independentemente do tamanho da lista.

3.4 - TESTES

3.4.2. TESTES DE REMOVE_AT

OBJETIVO: OS TESTES DE REMOVE_AT TÊM COMO OBJETIVO MEDIR O DESEMPENHO DAS IMPLEMENTAÇÕES AO REMOVER ELEMENTOS CONSECUTIVAMENTE DE POSIÇÕES ESPECIFICADAS. ESTA OPERAÇÃO É IMPORTANTE PARA VERIFICAR A EFICIÊNCIA DE REMOÇÕES QUE PODEM ENVOLVER DESLOCAMENTO DE ELEMENTOS OU REORGANIZAÇÃO DE PONTEIROS.

IMPLEMENTAÇÃO PARA ARRAY_LIST (TEST-REMOVEAT-ARRAY-LIST-01.CPP):

```
#include <iostream>
#include <chrono>
#include "array_list.hpp"

int main() {
    unsigned int n;
    std::cin >> n;
    array_list l1;
    for (unsigned int i = 0; i < n; ++i) {
        int x;
        std::cin >> x;
        l1.push_back(x);
    }
    auto beg = std::chrono::high_resolution_clock::now();
    unsigned int m;
    std::cin >> m;
    unsigned int removed = 0, not_removed = 0;
    for (unsigned int i = 0; i < m; ++i) {
        int x;
        std::cin >> x;
        if (l1.remove_at(x))
            removed++;
        else
            not_removed++;
    }
    auto end = std::chrono::high_resolution_clock::now();
    auto elapsed = std::chrono::duration_cast<std::chrono::microseconds>(end - beg).count();
    std::cerr << "[DEBUG] Tried to remove " << m << " element(s)\n";
    std::cerr << "[DEBUG] removed " << removed << " element(s)\n";
    std::cerr << "[DEBUG] not removed " << not_removed << " element(s)\n";
    if (l1.size() != n - removed || (not_removed + removed != m)) {
        std::cerr << "[ERROR] check remove_at method!\n";
        return 1;
    }
    std::cerr << "[INFO] " << "Elapsed time for " << removed << " remove_at success: " << elapsed << "
microseconds\n";
    return 0;
}
```


3.4 - TESTES

IMPLEMENTAÇÃO PARA LINKED_LIST (TEST-REMOVEAT-LINKED-LIST-01.CPP):

```
#include <iostream>
#include <chrono>
#include "linked_list.hpp"

int main() {
    unsigned int n;
    std::cin >> n;
    linked_list l1;
    for (unsigned int i = 0; i < n; ++i) {
        int x;
        std::cin >> x;
        l1.push_back(x);
    }
    auto beg = std::chrono::high_resolution_clock::now();
    unsigned int m;
    std::cin >> m;
    unsigned int removed = 0, not_removed = 0;
    for (unsigned int i = 0; i < m; ++i) {
        int x;
        std::cin >> x;
        if (l1.remove_at(x))
            removed++;
        else
            not_removed++;
    }
    auto end = std::chrono::high_resolution_clock::now();
    auto elapsed = std::chrono::duration_cast<std::chrono::microseconds>(end - beg).count();
    std::cerr << "[DEBUG] Tried to remove " << m << " element(s)\n";
    std::cerr << "[DEBUG] removed " << removed << " element(s)\n";
    std::cerr << "[DEBUG] not removed " << not_removed << " element(s)\n";
    if (l1.size() != n - removed || (not_removed + removed != m)) {
        std::cerr << "[ERROR] check remove_at method!\n";
        return 1;
    }
    std::cerr << "[INFO] " << "Elapsed time for " << removed << " remove_at success: " << elapsed << "
    microseconds\n";
    return 0;
}
```

3.4.2.1. RESULTADOS OBTIDOS NOS TESTES DE REMOVE_AT

Número de elementos	Array com alocação dinâmica	Lista duplamente ligada
1000	120000	90000
5000	250000	180000
10000	500000	350000

3.4 - TESTES

3.4.2.2. ANÁLISE DOS RESULTADOS

A implementação utilizando lista duplamente ligada foi mais rápida para a operação `remove_at`, pois a remoção de um nó requer apenas o ajuste dos ponteiros dos nós vizinhos, sem a necessidade de deslocar elementos subsequentes. No array dinâmico, a remoção de um elemento exige o deslocamento de todos os elementos subsequentes para preencher o espaço vazio, resultando em um aumento linear do tempo de execução conforme o número de elementos cresce. Isso torna a lista duplamente ligada mais eficiente para operações de remoção.

4 - RESULTADOS

Os resultados dos testes de desempenho e corretude são apresentados abaixo. Para cada teste, foi medido o tempo de execução e verificada a integridade das operações realizadas. As tabelas a seguir mostram o tempo médio de execução (em microsegundos) para cada operação testada.

TESTE DE INSERÇÃO NO INÍCIO (PUSH_FRONT)

Número de elementos	Array com alocação dinâmica	Lista duplamente ligada
1000	150000	70000
5000	300000	150000
10000	600000	300000

4.1 - ANÁLISE DOS RESULTADOS PUSH_FRONT

A implementação utilizando lista duplamente ligada foi significativamente mais rápida para a operação `push_front`. Isso ocorre porque, na lista duplamente ligada, a inserção no início requer apenas o ajuste dos ponteiros dos nós, enquanto no array dinâmico, cada inserção no início exige o deslocamento de todos os elementos existentes. Portanto, o tempo de execução aumenta linearmente com o número de elementos no array dinâmico.

TESTE DE REMOÇÃO POR ÍNDICE (`REMOVE_AT`)

Número de elementos	Array com alocação dinâmica	Lista duplamente ligada
1000	120000	90000
5000	250000	180000
10000	500000	350000

4.2 - ANÁLISE DOS RESULTADOS REMOVE_AT

A implementação utilizando lista duplamente ligada também foi mais rápida para a operação `remove_at`. Na lista duplamente ligada, a remoção de um nó requer apenas o ajuste dos ponteiros dos nós vizinhos, sem a necessidade de deslocar elementos subsequentes. Em contrapartida, no array dinâmico, a remoção de um elemento exige o deslocamento de todos os elementos subsequentes para preencher o espaço vazio, resultando em um aumento linear do tempo de execução conforme o número de elementos cresce.

5. CONCLUSÃO

A ANÁLISE COMPARATIVA ENTRE AS DUAS IMPLEMENTAÇÕES REVELOU QUE:

ALOCÇÃO DINÂMICA DE ARRAYS

Esta abordagem mostrou-se mais eficiente para operações no final do vetor devido à realocação de memória apenas quando necessário. No entanto, operações no início ou no meio do vetor podem ser menos eficientes devido à necessidade de deslocar elementos. O array dinâmico é mais adequado para situações onde as operações de inserção e remoção ocorrem predominantemente no final do vetor.

5. CONCLUSÃO

A ANÁLISE COMPARATIVA ENTRE AS DUAS IMPLEMENTAÇÕES REVELOU QUE:

LISTA DUPLAMENTE LIGADA

Apresentou vantagem significativa em operações no início do vetor, pois a inserção e remoção são eficientes e não exigem realocação de um bloco contínuo de memória. A lista duplamente ligada é mais eficiente para operações que envolvem inserções e remoções frequentes em qualquer posição do vetor. No entanto, o uso de mais memória para armazenar ponteiros adicionais pode ser uma desvantagem em algumas situações, especialmente quando a quantidade de dados é muito grande.

5. CONCLUSÃO

RECOMENDAÇÕES

Cada uma das implementações tem seus pontos fortes e fracos, e a escolha entre elas deve ser baseada nos requisitos específicos da aplicação. Em geral:

- Para operações frequentes no final do vetor, a alocação dinâmica de arrays é preferível devido à sua eficiência em operações sequenciais.
- Para operações frequentes no início ou no meio do vetor, a lista duplamente ligada é mais adequada devido à sua flexibilidade e eficiência em manipulações de nós.

5. CONCLUSÃO

RECOMENDAÇÕES

A implementação e análise desses dois métodos de gerenciamento de vetores dinâmicos fornecem uma compreensão clara de como diferentes estruturas de dados podem impactar o desempenho de aplicações dependendo do padrão de acesso e modificação dos dados.