INSTITUTO FEDERAL
Rio Grande do Norte
Campus Natal-Central

Trabalho de Algoritmos

Definição e implementação de um vetor dinâmico

Curso: Análise e Desenvolvimento de Sistemas - 2º período
Aluno: Luiz Fernando Gama Nery
Professor: Jorgiano Vidal

IFRN, Natal, Rio Grande do Norte

SUMÁRIO

---

01. Introdução

Nesse relatório será apresentado as definições e implementações de um vetor dinâmico utilizando a linguagem C++, onde serão mostrados dois métodos diferentes sendo eles: **Alocação dinâmica de arrays** e o outro utilizando **Lista Duplamente Ligada.** Este trabalho tem como objetivo explicar mais sobre a gerência de memória que é uma característica da matéria de Algoritmos no qual estamos cursando no presente momento.

02. Vetor Dinâmico

Na alocação dinâmica podemos alocar espaços durante a execução de um programa, ou seja, a alocação dinâmica é feita em tempo de execução, o que é bem interessante do ponto de vista do programador, porque permite que o espaço em memória seja alocado apenas quando necessário. Além do mais, a alocação dinâmica permite aumentar ou até diminuir a quantidade de memória alocada.

03. Implementação

03.01. Organização de arquivos fontes

Na produção deste trabalho foram utilizados alguns códigos feitos na linguagem C++ incluindo:

- array_list.hpp (Neste arquivo está contido o código principal onde está contido a classe que contém todos os métodos solicitados no trabalho, incluindo o construtor que define as informações necessárias como tamanho do array).

- array_list.cpp (Este arquivo contém apenas alguns testes de funções do arquivo .hpp, entre eles alguns métodos e leituras de lista).

- linked_list.hpp (Neste arquivo está contido o código principal onde está contido a classe que contém todos os métodos solicitados no trabalho, incluindo o construtor que define as informações necessárias como tamanho do array(Na lista ligada não há necessidade já que o seu valor só aumenta à medida que é adicionado valores ao programa)).

- linked_list.cpp (Este arquivo contém apenas alguns testes de funções do arquivo .hpp, entre eles alguns métodos e leituras de lista).

- test-pushfront-array-list-01.cpp (Esse arquivo de teste foi fornecido pelo orientador do trabalho e contém um código para testar o método **push front** do arquivo .hpp com diferentes valores de teste que serão mostrados a seguir).

- test-pushfront-linked-list-01.cpp (Esse arquivo de teste foi fornecido pelo orientador do trabalho e contém um código para testar o método **push front** do arquivo .hpp com diferentes valores de teste que serão mostrados a seguir).

- test-removeat-array-list-01.cpp (Esse arquivo de teste foi fornecido pelo orientador do trabalho e contém um código para testar o método **remove at** do arquivo .hpp com diferentes valores de teste que serão mostrados a seguir).

- test-removeat-linked-list-01.cpp test-removeat-array-list-01.cpp (Esse arquivo de teste foi fornecido pelo orientador do trabalho e contém um código para testar o método **remove at** do arquivo .hpp com diferentes valores de teste que serão mostrados a seguir).

Os mesmos citados acima estarão anexados juntamente a esse relatório no **GitHub** e poderão ser acessados e testados caso seja necessário.

03.02. Arrays com alocação dinâmica

Arrays com alocação dinâmica consiste basicamente em uma classe que que possui métodos e atributos assim como a lista duplamente ligada, porém sua capacidade aumenta à medida que o tamanho do array está cheio.

A classe **`array_list`** utiliza um ponteiro para um array estático e realoca a memória conforme necessário. Três estratégias de aumento de capacidade são implementadas:

1. Capacidade inicial de 100, aumentando em 100 cada vez que precisar de mais.
2. Capacidade inicial de 1000, aumentando em 1000 cada vez que precisar de mais.
3. Capacidade inicial de 8, duplicando cada vez que precisar de mais.

Abaixo serão listados os métodos presentes nessa classe:

O método **increase_capacity** adiciona espaço de memória ao array quando o mesmo está cheio.

```
void increase_capacity() {

    capacity_ *= 2;

    int *new_data = new int[capacity_];

    for (unsigned int i = 0; i < size_; ++i) {

      new_data[i] = data[i];

    }

    delete[] data;

    data = new_data;

  }
```

O método **percent_occupied** retorna o percentual de memória ocupado.

```
double percent_occupied() {

    return (static_cast<double>(this->size_) / this->capacity_) * 100;

  }
```

O método **`insert_at`** insere um novo elemento em uma posição específica dentro do vetor dinâmico. Ele possui verificações de validação do index, caso a capacidade esteja esgotada, ele aloca mais memória de acordo com as estratégias de aumento citadas mais acima.

Desempenho: O(n)

```
bool insert_at(unsigned int index, int value) {

    if (index > this->size_) {

      return false;

    }

    if (this->size_ == this->capacity_) {

      increase_capacity();

    }

    for (unsigned int i = this->size_; i > index; --i) {

      data[i] = data[i - 1];

    }

    data[index] = value;

    ++this->size_;

    return true; }
```

O método **remove_at** tem a mesma proposta do insert, porém removendo um elemento de alguma posição dentro do array. Possui os mesmos métodos de verificação, move os elementos subsequentes para a esquerda para preencher o espaço vazio deixado pelo elemento removido e atualiza o tamanho do array.

Desempenho: O(n)

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

O método **get_at** retorna o elemento da posição desejada do array. Caso não seja passado nenhum parâmetro dentro do método, se for utilizado um laço da forma correta, ele retorna os valores presentes dentro do vetor.

Desempenho: O(1)

```
int get_at(unsigned int index) {
    return this->data[index];
  }
```

O método **clear** limpa e reseta todos os atributos utilizados na classe.

```
void clear() {
    this->size_ = 0;
    this->capacity_ = 0;
    delete[] data;
    data = new int[capacity_];
  }
```

O método **push_back** adiciona um valor no final do array.
Desempenho: O(1)

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

O método **push_front** adiciona um valor ao início do array e desloca todos os outros elementos uma posição a mais.
Desempenho: O(n)

```
void push_front(int value) {
    if (this->size_ == this->capacity_)
      increase_capacity();
    for (int i = this->size_; i > 0; --i) {
      data[i] = data[i - 1];
    }
    data[0] = value;
    ++this->size_;
  }
```

O método **pop_back** remove o último valor do array.
Desempenho: O(1)

```
bool pop_back() {
    if (this->size_ == 0) {
      return false; // Lista vazia
```

```
    }
    --this->size_;
    return true;
  }
```

O método **pop_front** remove o primeiro valor do array.
Desempenho: O(n)

```
bool pop_front() {
    if (this->size_ == 0) {
      return false; // Lista vazia
    }
    for (unsigned int i = 1; i < size_; ++i) {
      data[i - 1] = data[i];
    }
    --this->size_;
    return true;
  }
```

Os métodos **front** e **back** retornam consecutivamente o primeiro e o último elemento do vetor.

```
int front() {
    if (this->size_ > 0)
      return this->data[0];
    return -1;
  }

  int back() {
    if (this->size_ > 0)
      return this->data[this->size_ - 1];
    return -1;
  }
```

O método **remove** retira um elemento do vetor por meio do índice.
Desempenho: O(n)

```
bool remove(int value) {
    int index = find(value);
    if (index != -1) {
      remove_at(index);
      return true;
    }
    return false;
  }
```

O método **find** retorna o índice do primeiro elemento com o valor sugerido.
Desempenho: O(n)

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

O método **count** retorna quantas vezes um valor está presente dentro do vetor.

```
int count(int value) {
    int conta = 0;
    for (unsigned int i = 0; i < this->size_; ++i) {
      if (this->data[i] == value) {
        conta++;
      }
    }
    return conta;
  }
```

O método **sum** soma todos os valores presentes no array e retorna o total.

```
int sum() {
```

```
    int soma = 0;
    for (unsigned int i = 0; i < this->size_; ++i) {
      soma += this->data[i];
    }
    return soma;
  }
};
```

03.03. Lista Ligada

A lista ligada também consiste em uma classe com métodos e funções, porém diferentemente da lista de Arrays com alocação dinâmica ela não vem com memória pré-alocada, à medida que ela é criada, há um ponteiro que é basicamente um nó onde possui o index anterior e posterior já preparados para receber valores, dessa forma, ele é essencial para tarefas que pequenas que utilizem pouca memória.

Abaixo serão listados os métodos presentes nessa classe:

O método **percent_occupied** retorna o percentual de memória ocupado. No caso da Lista Ligada, sempre será 100% pois ela só aumenta à medida que os valores são adicionados.

```
double percent_occupied() {

        return 100.0;

    }
```

O método **`insert_at`** insere um novo elemento em uma posição específica dentro do vetor dinâmico. Ele possui verificações de validação do index, caso a capacidade esteja esgotada, ele aloca mais memória de acordo com as estratégias de aumento citadas mais acima.

Desempenho: O(n)

```
bool insert_at(unsigned int index, int value) {

        if (index > this->size_) {

            return false; // Índice fora do intervalo

        }



        int_node* new_node = new int_node;

        new_node->value = value;



        if (index == 0) {

            // Inserir no início

            new_node->next = this->head;

            new_node->prev = nullptr;

            if (this->head != nullptr) {
```

```
            this->head->prev = new_node;
        }
        this->head = new_node;
        if (this->tail == nullptr) {
            this->tail = new_node;
        }
    } else if (index == this->size_) {
        // Inserir no final
        new_node->next = nullptr;
        new_node->prev = this->tail;
        if (this->tail != nullptr) {
            this->tail->next = new_node;
        }
        this->tail = new_node;
        if (this->head == nullptr) {
            this->head = new_node;
        }
    } else {
        // Inserir no meio
        int_node* current = this->head;
        for (unsigned int i = 0; i < index; ++i) {
            current = current->next;
        }
```

```
            new_node->next = current;

            new_node->prev = current->prev;

            current->prev->next = new_node;

            current->prev = new_node;

        }



        this->size_++;

        return true;

    }
```

O método **remove_at** tem a mesma proposta do insert, porém removendo um elemento de alguma posição dentro do array. Possui os mesmos métodos de verificação, move os elementos subsequentes para a esquerda para preencher o espaço vazio deixado pelo elemento removido e atualiza o tamanho do array.

Desempenho: O(n)

```
bool remove_at(unsigned int index) {
        if (index >= this->size_)
            return false; // Não removeu
        int_node* to_remove = this->head;
        for (unsigned int i = 0; i < index; ++i)
            to_remove = to_remove->next;
        if (to_remove->prev != nullptr)
            to_remove->prev->next = to_remove->next;
        if (to_remove->next != nullptr)
            to_remove->next->prev = to_remove->prev;
        delete to_remove;
        return true; // Removeu
    }
```

O método **get_at** retorna o elemento da posição desejada do array. Caso não seja passado nenhum parâmetro dentro do método, se for utilizado um laço da forma correta, ele retorna os valores presentes dentro do vetor.

Desempenho: O(n)

```
int get_at(unsigned int index) {
    // Verifica se o índice está fora dos limites
    if (index >= this->size_) {
        throw std::out_of_range("error");
    }
    int_node* current = this->head;

    // Percorre a lista até o índice especificado
    for (unsigned int i = 0; i < index; ++i) {
        if (current == nullptr) {
            throw std::out_of_range("error");
        }
        current = current->next;
    }

    // Retorna o valor no índice especificado
    return current->value;
}
```

O método **clear** limpa e reseta todos os atributos utilizados na classe.

```
void clear() {
        int_node* current = this->head;
```

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

    this->capacity_ = 0;
    delete[] data;
    data = new int[capacity_];
  }
```

O método **push_back** adiciona um valor no final do array.
Desempenho: O(1)

```
void push_back(int value) {
        int_node* new_node = new int_node;
        new_node->value = value;
        new_node->next = nullptr;
        new_node->prev = this->tail;

        if (this->tail != nullptr) {
            this->tail->next = new_node;
        }
        this->tail = new_node;

        if (this->head == nullptr) {
            this->head = new_node;
        }

        this->size_++;
    }
```

O método **push_front** adiciona um valor ao início do array e desloca todos os outros elementos uma posição a mais.
Desempenho: O(1)

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

O método **pop_back** remove o último valor do array.
Desempenho: O(1)

```
bool pop_back() {
        if (this->tail == nullptr) {
            return false;
        }

        int_node* to_remove = this->tail;
        if (this->tail->prev != nullptr) {
            this->tail = this->tail->prev;
            this->tail->next = nullptr;
        } else {
            this->head = nullptr;
            this->tail = nullptr;
        }

        delete to_remove;
        this->size_--;
        return true;
    }
```

O método **pop_front** remove o primeiro valor do array.
Desempenho: O(1)

```
bool pop_front() {
        if (this->head == nullptr) {
            return false;
        }

        int_node* to_remove = this->head;
        if (this->head->next != nullptr) {
            this->head = this->head->next;
            this->head->prev = nullptr;
        } else {
            this->head = nullptr;
            this->tail = nullptr;
        }

        delete to_remove;
        this->size_--;
        return true;
    }
```

Os métodos **front** e **back** retornam consecutivamente o primeiro e o último elemento do vetor.

```
int front() {
        if (this->head == nullptr) {
            throw std::out_of_range("A lista está vazia.");
        }
        return this->head->value;
    }
    int back() {
        if (this->tail == nullptr) {
            throw std::out_of_range("A lista está vazia.");
        }
        return this->tail->value;
```

O método **remove** retira um elemento do vetor por meio do índice.
Desempenho: O(n)

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

O método **find** retorna o índice do primeiro elemento com o valor sugerido.
Desempenho: O(n)

```
int find(int value) {
        int_node* current = this->head;
        int index = 0;
        while (current != nullptr) {
            if (current->value == value) {
                return index;
            }
```

```
            current = current->next;
            index++;
        }
        return -1;
    }

  }
```

O método **count** retorna quantas vezes um valor está presente dentro do vetor.

```
int count(int value) {
        int_node* current = this->head;
        int conta = 0;
        while (current != nullptr) {
            if (current->value == value) {
                conta++;
            }
            current = current->next;
        }
        return conta;
    }
```

O método **sum** soma todos os valores presentes no array e retorna o total.

```
int sum() {
        int_node* current = this->head;
        int soma = 0;
        while (current != nullptr) {
            soma += current->value;
            current = current->next;
        }
        return soma;
    }
```

04. Testes

Nessa parte do projeto serão mostrados alguns testes realizados utilizando as classes feitas e mostradas no tópico anterior. Os testes serão apresentados e neles irão conter a quantidade de números e o tempo em milissegundos (ms) no qual o programa demorou pra ser executado.

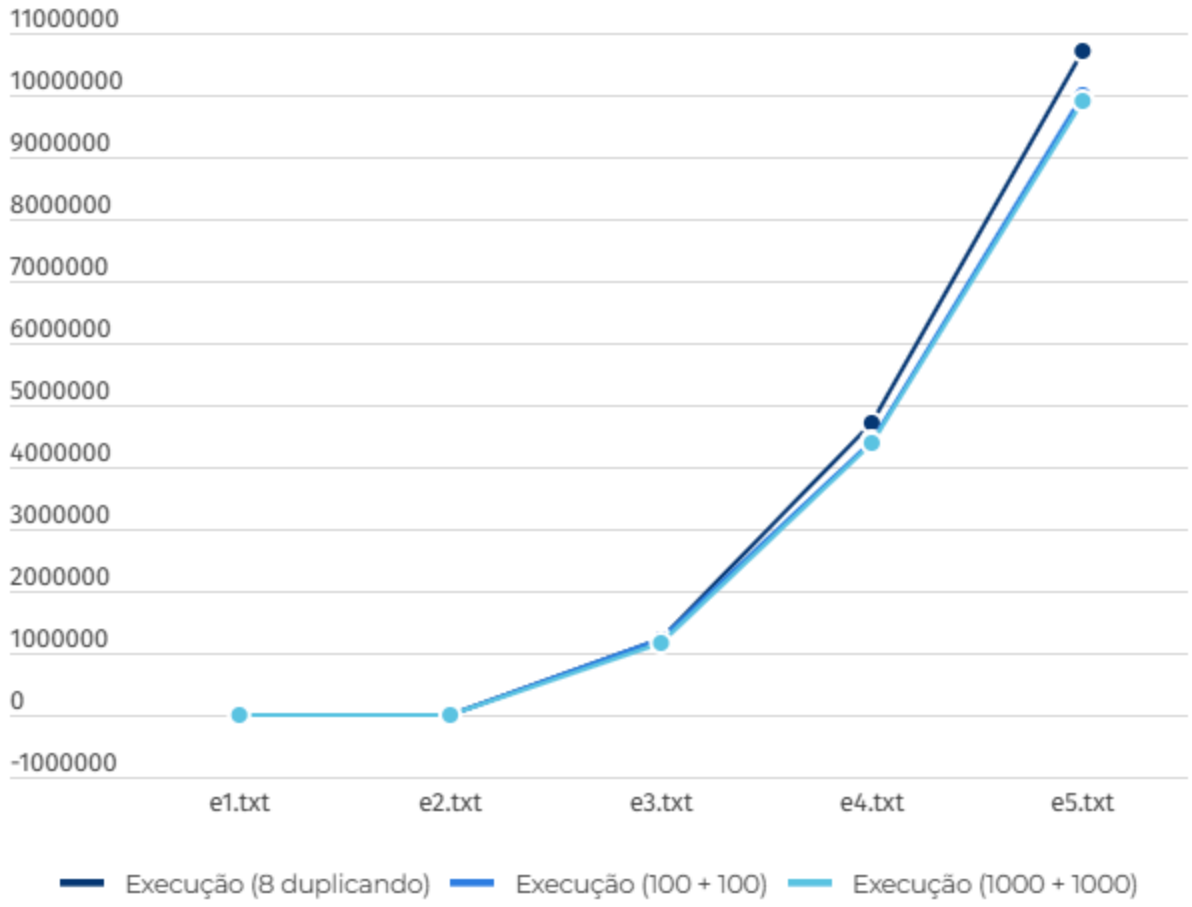
04.01. Testes feitos

O primeiro teste será feito utilizando o método **push_front**.
Os arquivos utilizados para teste contém cada um deles uma quantidade de números que serão definidos abaixo:
e1.txt - 5 números
e2.txt - 10 números
e3.txt - 1000 números
e4.txt - 2000 números
e5.txt - 3000 números

O segundo teste será feito utilizando o método **remove_at**.
Os arquivos utilizados para teste contém cada um deles uma quantidade de números que serão definidos abaixo:
e1.txt - 15 números
e2.txt - 15 números

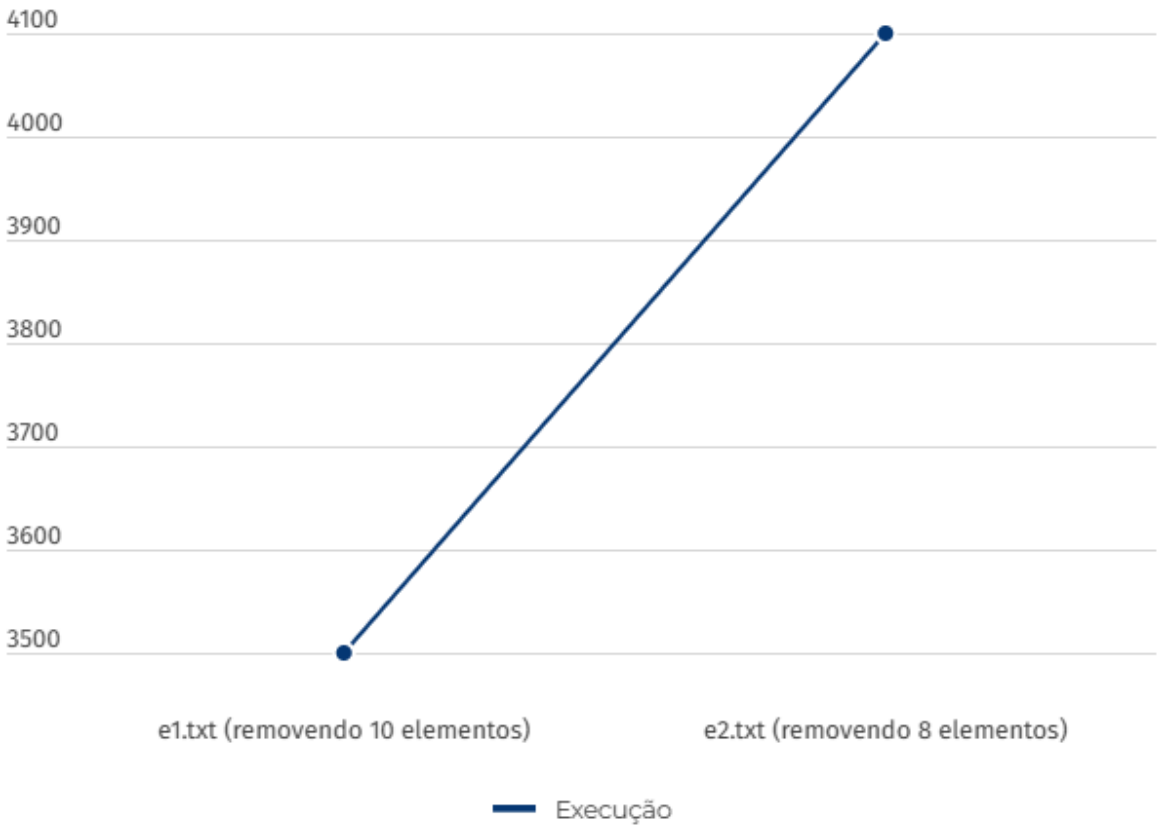
PUSH FRONT - ARRAY LIST

Arquivo utilizado	**Tempo de execução (8 e vai duplicando)**	**Tempo de execução (100 em 100)**	**Tempo de execução (1000 em 1000)**
e1.txt	1400 ms	1200 ms	1000 ms
e2.txt	4600 ms	3100 ms	2900 ms
e3.txt	1237300 ms	1213100 ms	1171400 ms
e4.txt	4703801 ms	4431001 ms	4382300 ms
e5.txt	10705403 ms	9984000 ms	9919100 ms



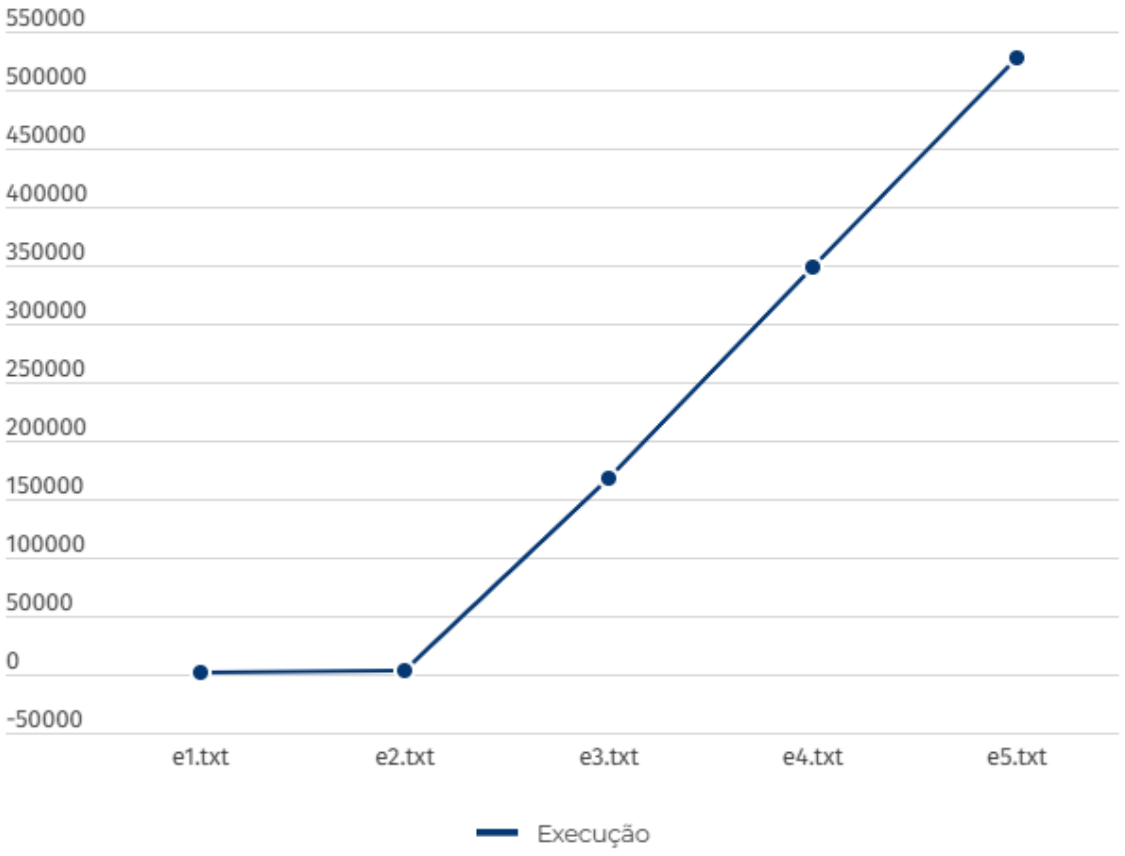
REMOVE AT - ARRAY LIST

Arquivo utilizado	**Tempo de execução**
e1.txt (removendo 10 elementos)	3500 ms
e2.txt (removendo 8 elementos)	4100 ms



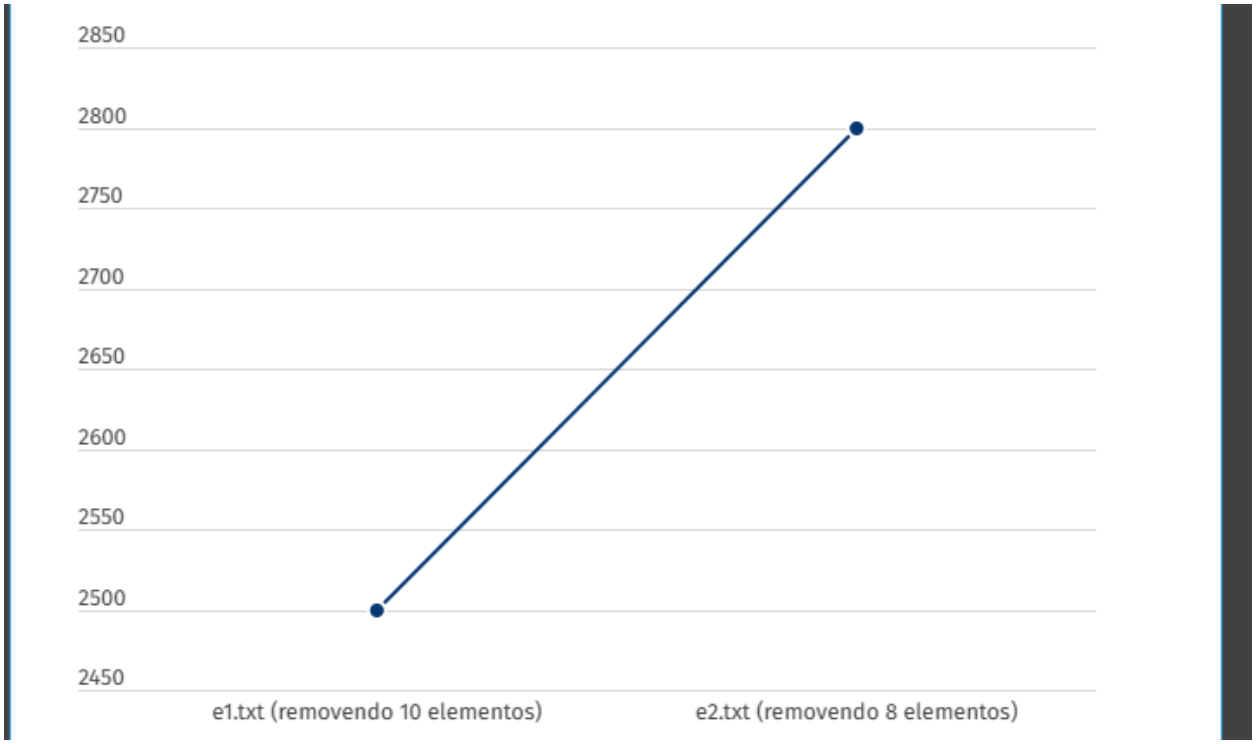
PUSH FRONT - LINKED LIST

Arquivo utilizado	**Tempo de execução**
e1.txt	2100 ms
e2.txt	2600 ms
e3.txt	168300 ms
e4.txt	348500 ms
e5.txt	527300 ms



REMOVE AT - LINKED LIST

Arquivo utilizado	**Tempo de execução**
e1.txt (removendo 10 elementos)	2500 ms
e2.txt (removendo 8 elementos)	2800 ms



05. Resultados

A partir dos testes realizados, é possível concluir que o método **push_front** quando utilizado com valores baixos (entre 1 e 5) é mais eficiente no Array List, porém quando são valores maiores, o Linked List passa a se tornar a opção mais viável a ser utilizada devido o seu tempo de execução que acaba ficando bem menor comparado a Array List com valores altos (acima de 1000).

No método **remove_at**, fica visível que no Linked List o tempo de execução é menor que utilizando a classe Array List, e que o tempo de execução quando se trata da remoção de menos valores pode variar e se tornar mais alta do que removendo quantidades maiores no vetor.

Outro fator que foi mostrado pelos testes é que o tempo de execução no método **push_front** quando a capacidade está em 100 ou 1000 é menor, ou seja, ele se torna mais eficiente quando os valores pré definidos de capacidade são maiores e garantem uma folga para que o vetor trabalhe de forma que não precise alocar mais espaço, e consequentemente não use tanta memória do programa.

06. Conclusão

Este trabalho ofereceu uma visão detalhada sobre as diferenças entre as implementações de vetores dinâmicos utilizando arrays com alocação dinâmica e listas duplamente ligadas. Cada abordagem apresenta seus próprios benefícios e limitações, tornando a escolha da estrutura mais adequada dependente das operações que serão executadas com maior frequência.