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

Algoritmos – 2024.1

Fábio Alexandre de Souza Lucena Filho

RELATÓRIO SOBRE A IMPLEMENTAÇÃO DE UM VETOR DINÂMICO

Natal, Rio Grande do Norte
2024

SUMÁRIO

01. INTRODUÇÃO

Este trabalho da matéria de algoritmos tem como objetivo implementar uma biblioteca de classes de lista dinâmica e outra lista duplamente ligada, usufruindo de alocação de memória dinamicamente Tudo isso será realizado por meio de um arquivo .hpp. A linguagem de programação escolhida para realização deste trabalho será C++, visto que esta linguagem é uma das mais eficientes quando se trata de análise de desempenho, o que também é um dos focos que serão abordados ao decorrer deste relatório.

02. VETORES DINÂMICOS

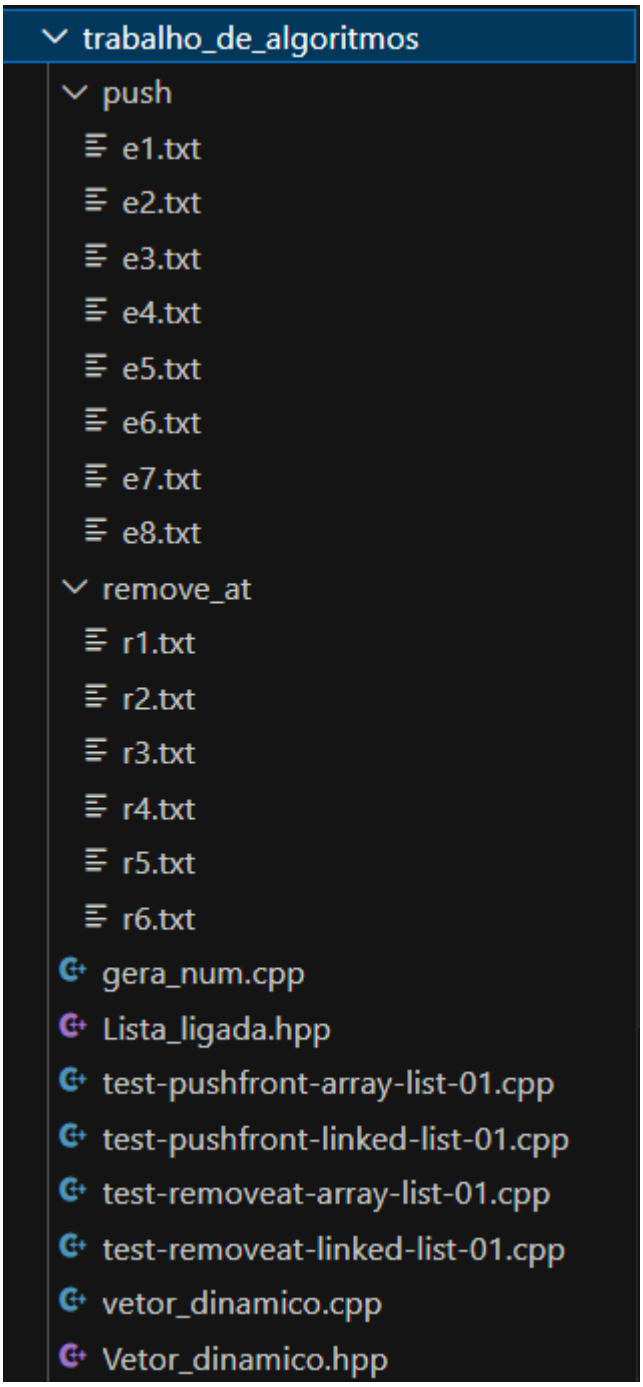
Vetor Dinâmico, mais conhecido na linguagem de programação Python como método list é basicamente um ponteiro que aumenta a capacidade de armazenamento conforme a necessidade do usuário, permitindo que elementos sejam, principalmente inseridos e removidos dinamicamente. Como por exemplo se o seu vetor tem apenas 100 de tamanho,você poderá inserir apenas 100 elementos, mas caso você esteja utilizando vetores dinâmicos existe a possibilidade de inserir mais elementos utilizando um método chamado increase_capacity(), método o qual será explicado mais adiante neste relatório.

03. IMPLEMENTAÇÃO

Nessa parte iremos implementar o Arraylist e as Lista Duplamente Ligadas, analisar a taxa de desempenho, e também explicar o que cada método irá realizar.

03. 01 ORGANIZAÇÃO DOS ARQUIVOS FONTES

Esse espaço será dedicado exclusivamente para separação dos arquivos fontes usados na implementação do vetor dinâmico e da lista duplamente ligada.



03. 02 ARRAY COM ALOCAÇÃO DINÂMICA

Increase_capacity_()
Taxa de desempenho do método: o(n).

Esse método tem como função aumentar dinamicamente a capacidade do ponteiro, permitindo que você insira mais elementos no seu array.

```
   int* array;
    unsigned int size_, capacity_;
    void increase_capacity_() { // Aumenta capacidade do seu ponteiro
(array);
        int *new_array = new int[capacity_ + 100]; // Cria um novo ponteiro
e reserva na memória tamanho capacity_ + 100;
        for (unsigned int i=0; i < this->size_; i++) new_array[i] =
array[i]; // O novo ponteiro (new_array) receberá os mesmos valores que o
outro ponteiro (array);
    }
        delete [] array; // Libere a memória utilizada pelo ponteiro
antigo(array);
        array =  new_array; // O ponteiro antigo (array) apontará para o
ponteiro novo (new_array);
        capacity_ = capacity_ + 100; // Some a capacidade antiga + 100;
```

Vetor_dinamico()
Taxa de desempenho do método o(1).

Esse construtor irá atribuir os valores iniciais dos atributos.

```
    Vetor_dinamico() { // Construtor
        this->size_ = 0; // Definindo tamanho para 0;
        this->capacity_ = 100; // Definindo a capacidade para 100;
```

```
        this->array = new int[capacity_]; // Criando novo ponteiro de
tamanho capacidade.
    }
```

~Vetor_dinâmico()
Taxa de desempenho do método o(1)

Esse método terá como função liberar a memória alocada por array.

```
    ~Vetor_dinamico() { // Destrutor
        delete [] this->array; // Libere a memória alocada por array.
    }
```

unsigned int size()
Taxa de desempenho do método o(1)

Esse método terá como função retornar o tamanho do array.

```
unsigned int size() { // Retorna a quantidade de elementos armazenados
        return this->size_;
    }
```

unsigned int capacity()
Taxa de desempenho do método o(1)

Esse método terá como função retornar a capacidade atual do array.

```
    unsigned int capacity() { // Retorna o espaço reservado para
armazenar os elementos
        return this->capacity_;
    }
```

double percent_occupied()
Taxa de desempenho do método o(1)

Esse método terá como função retornar o percentual ocupado da memória pelo array

---

```
    double percent_occupied() { // Retorna um valor entre 0.0 a 1.0 com
o percentual da memória usada.
        double retorno = 0; // Supondo que o array está vazio ele não
possui nada ocupado então retorno será 0.0;
        if (this->size_ == this->capacity_ ) retorno = 1.0; // Se o
tamanho for igual capacidade quer dizer que tudo foi ocupado então
retorno será 1.0;
        else retorno = ((double)this->size_ / (double)this->capacity_);
// Caso contrário o percentual ocupado será a divisão de tamanho por
capacidade;
        return retorno; // Retorna o valor de retorno.
    }
```

bool insert_at(unsigned int index, int value)
Taxa de desempenho do método o(n)

Esse método terá como função inserir um elemento qualquer no índice desejado.

```
bool insert_at(unsigned int index, int value) { // Insere elemento no
índice index
        int retorno = false;
        if (index < this->size_) retorno = true;
        if (this->capacity_ == this->size_) increase_capacity_();
        int atual = this->array[index], proximo = this->array[index +
1], ultimo = this->array[this->size_-1];
        this->array[index] = value;
        for (unsigned int i = index + 1; i < this->size_;i++){
                proximo = this->array[i];
                this->array[i] = atual;
                atual = proximo;
        }
        this->array[this->size_] = ultimo;
        this->size_++;
        return retorno;
    }
```

bool remove_at(unsingned int index)
Taxa de desempenho do método o(n)

Esse método terá como função remover um elemento qualquer no índice desejado.

```
    bool remove_at(unsigned int index) {
        int retorno = true; // Retorno é verdade;
        if (index >= size_) { // Se o index for maior ou igual size
retorno passa a ser falso;
            retorno = false;
        }
        for (unsigned int i = index; i < size_ - 1; ++i) {
            array[i] = array[i + 1]; // Comece o loop no index e a
posição que ele começa vai receber o elemento que está na frente;
        }
        size_--; // Diminua o tamanho do array;
        return retorno; // Retorne retorno;
    }
```

int get_at(unsigned int index)
Taxa de desempenho do método o(n)

Esse método terá como função retornar um elemento qualquer baseado no índice desejado.

```
int get_at(unsigned int index) { // Retorna elemento no índice index,
-1 se índice inválido
        if (index >= this->size_){
            this->array[index] = -1;
        }
        return this->array[index];
    }
```

void clear()
Taxa de desempenho do método o(1)

Esse método remove todos os elementos deixando o vetor no estado inicial.

```
void clear() {  // Remove todos os elementos, deixando o vetor no
estado inicial
            size_ = 0; // Tamanho será 0;
            capacity_ = 0; // Capacidade será 0;
            delete [] array; // Libere o espaço de memória usado por
array
            array = new int[capacity_]; // Array recebe novo espaço de
memória de tamanho capacidade;
    }
```

void push_back(int value)
Taxa de desempenho do método o(n)

Esse método tem como função adicionar o número desejado no final do vetor.

```
void push_back(int value) { // Adiciona um elemento no ``final'' do
vetor
        if (this->capacity_==this->size_) increase_capacity_(); // Se
capacidade for igual tamanho chame a função increase_capacity_()
        this->array[this->size_] = value; // A última posição passa a
ser value
        this->size_ +=1;   // Aumente o tamanho do array
    }
```

void push_front(int value)
Taxa de desempenho do método o(n)

Esse método tem como função adicionar o número desejado no início do vetor.

```
void push_front(int value) { // Adiciona um elemento no início do vetor
        if (this->capacity_==this->size_) increase_capacity_();
            int atual = array[0], proximo = array[1];
```

```
            for (unsigned int i = 0; i < this->size_+1;i++){
                proximo = this->array[i];
                this->array[i] = atual;
                atual = proximo;
            }
        this->array[0] = value;
        this->size_+= 1;
        }
```

bool pop_back()
Taxa de desempenho do método o(1)

Esse método tem como função remover o último elemento do vetor.

```
bool pop_back() { // Remove um elemento do ``final'' do vetor
        int retorno = false; // Crie uma variável chamada retorno e ela
é falsa
        if (this->size_ >= 1){ // Se o tamanho do vetor for maior que 1
            this->size_--; // Diminua o tamanho do vetor em 1;
            retorno = true; // E retorno passará a ser verdadeiro;
        }
        return retorno; // Retorna a variável retorno.
   }
```

bool pop_front()
Taxa de desempenho do método o(n)

Esse método tem como função remover o primeiro elemento do vetor.

```
    bool pop_front() {  // Remove um elemento do ``início'' do vetor
        int retorno = false; // Crie uma variável chamada retorno e ela
é falsa;
        if (this->size_ > 1){ // Se o tamanho do vetor for maior que 1;
            retorno = true; // Retorno passa a ser verdade;
            for (unsigned int i = 0; i <this->size_-1;i++){
                array[i] = array[i+1]; // O "atual" que seria o 1
elemento do vetor passa a ser o que seria o "próximo" elemento do vetor
ou seja array[1,2,3] -> array[2,3]
            }
        }
        this->size_--; // Diminuo o tamanho em 1;
        return retorno; // Retorna a variável retorno.
    }
```

int back()
Taxa de desempenho do método o(1)

Esse método tem como função retornar o último elemento do vetor.

```
int back(){ // Retorna o elemento do ``final'' do vetor
        return this->array[this->size_-1]; // Retorne array[size-1],
nesse caso size-1 seria o último elemento.
    }
```

int front()
Taxa de desempenho do método o(1)

Esse método tem como função retornar o primeiro elemento do vetor.

```
int front(){ // Retorna o elemento do ``início'' do vetor
        return this->array[0]; // Retorna array[0], 0 seria o primeiro
elemento do vetor.
    }
```

bool remove(int value)
Taxa de desempenho do método o(n)

Esse método tem como função remover o elemento desejado no vetor, diferente do remove_at() que remove por índice, ele irá remover o número desejado, ou seja se o número não estiver presente ele retorna -1.

```
        bool remove(int value) { // Remove value do vetor caso esteja
presente ;
        int retorno = false; // Crie uma variável retorno e ela é
falsa;
        for (unsigned int i = 0 ; i < this->size_; i++){
            if (array[i] == value){ // Percorra a lista em busca de
achar se a condição é verdadeira ou falsa;
                retorno = true; // Se a condição for verdadeira retorno
se torna verdade;
                for (unsigned int j = i; j < this->size_ - 1; j++)
array[j] = array[j + 1]; // O "atual" que seria o valor de i do vetor
passa a ser o que seria o "próximo" elemento do vetor ou seja
array[1,2,3] -> array[2,3];
                break;
            }
        }
        this->size_--; // Diminua o tamanho do vetor em 1;
        return retorno; // Retorna a variável retorno
        }
```

int find(int value)
Taxa de desempenho do método o(n)

Esse método tem como função procurar o elemento desejado no vetor.

```
int find(int value) { // Retorna o índice de value, -1 caso value não esteja presente
        int valor = -1; // Crie uma variável valor e ela é igual a -1
        for (unsigned int i = 0; i < this->size_; i++){
            if (array[i] == value){ // Percorra a lista e se o valor desejado estiver presente, a variável valor passará a ser "i", onde "i" seria o índice onde está o "value" desejado;
                valor = i;
                break;
            }
        return valor; // Retorne a variável retorno.
        }
    }
```

int count(int value)
Taxa de desempenho do método o(n)

Esse método tem como função contar quantas vezes o elemento desejado está no vetor

---

```
    int count(int value) { // Retorna quantas vezes value occorre no vetor
        int cont = 0; // Crie um contador e atribua o valor 0 a ele;
        for (unsigned int i = 0; i < this->size_; i++){
            if (array[i] == value) cont++; // Percorra o vetor e se o valor for igual a "value", o valor da variável cont será aumentado em 1;
        }
        return cont; // Retorna a variável cont.
    }
```

int sum()
Taxa de desempenho do método o(n)

Esse método tem como função realizar a soma total do vetor.

```
 int sum() { // Retorna a soma dos elementos do vetor
        int soma = 0; // Crie um acumulador e atribua o valor 0 a ele;
        for (unsigned int i = 0; i < this->size_; i++){
            soma = soma + array[i]; // Percorra a lista e faça a soma
total do vetor;
        }
        return soma; // Retorna a soma total do vetor;
    }
```

03. 03 Lista Duplamente Ligada

int Lista_ligada()
Taxa de desempenho do método o(1)

Esse construtor irá atribuir os valores iniciais da lista duplamente ligada.

```
Lista_ligada() {
        this->head = nullptr; // O primeiro(head) será nulo;
        this->tail = nullptr; // O último(tail) será nulo;
        this->size_ = 0; // O tamanho será = 0.
    }
```

int -Lista_ligada()
Taxa de desempenho do método o(1)

Isso irá servir como um destrutor da sua lista duplamente ligada

---

```
 ~Lista_ligada() {
        int_node *atual = this->head; // Crie uma nova lista duplamente
ligada chamada atual e ela começa em head;
        int_node *i ;// Crie uma nova lista duplamente ligada i;
```

```
        while (atual != nullptr){ // Enquanto atual for diferente de
nulo;
        i = atual; // i aponta para atual;
        atual = atual->next; // isso serve para percorrer pela lista;
        delete  i; // libere a memória usada por i.
        }
    }
```

int size()
Taxa de desempenho do método o(1)

Esse método irá retornar o tamanho da sua lista duplamente ligada.

```
   unsigned int size() { // Retorna a quantidade de elementos
armazenados.
        return this->size_;
    }
```

bool insert_at(unsigned int index, int value)
Taxa de desempenho do método o(n)

Esse método terá como função inserir um elemento qualquer no índice desejado.

---

bool remove_at(unsingned int index)
Taxa de desempenho do método o(n)

Esse método terá como função remover um elemento qualquer no índice desejado.

---

int get_at(unsigned int index)
Taxa de desempenho do método o(n)

Esse método terá como função retornar um elemento qualquer baseado no índice desejado.

```
int get_at(unsigned int index) { // Retorna elemento no índice index,
-1 se índice inválido
        int retorno;//  Crie uma variável retorno;
        if (index >= size_) retorno =  -1; // Se o index for maior ou
igual ao tamanho retorno será -1;
        else {
        int_node *atual = this->head; //  Crie uma nova lista;
duplamente ligada chamada atual e ela aponta para head
        for (int i = 0; i < index ;i++) atual = atual->next;// Isso
serve para percorrer pela lista;
retorno =  atual->value; // retorna o atual valor ;
        }
        return retorno; // retorne retorno.
    }
```

void clear(unsigned int index)
Taxa de desempenho do método o(n)

Esse método remove todos os elementos deixando o vetor no estado inicial.

```
   void clear() { // Remove todos os elementos, deixando o vetor no
estado inicial
        int_node *atual = this->head; // Crie uma nova lista duplamente
ligada chamada atual e ela começa em head;
        int_node *i;// Crie uma nova lista duplamente ligada chamada i;
        while (atual != nullptr){ // Enquanto atual for diferente de
nulo
        i = atual; // i aponta para atual;
        atual = atual->next; // Percorra pela lista;
        delete  i; // Apague a memória usada por i.
        }
    }
```

void push_front(int value)
Taxa de desempenho do método o(1)

Esse método tem como função adicionar o número desejado no início do vetor.

```
    void push_front(int value) { // Adiciona um elemento no ``início''
do vetor
        int_node *novo_no = new int_node; // Crie uma nova lista
duplamente ligada chamada novo_no;
        novo_no->value = value; // O valor de novo_no aponta para value
        novo_no->next = head; // O próximo em novo_no apontará para
cabeça;
        novo_no->prev = nullptr; // previous(prev) de novo_no é nulo
        if (this->head == nullptr) this->tail = novo_no; // Se o head
de novo_no for nulo, o rabo de novo_no aponta para novo_no;
```

```
        else this->head->prev = novo_no; // O que vem antes de head
aponta para o novo_no
        this->head = novo_no; // Head aponta para novo no;
        size_++; // Aumente o tamanho em 1.
    }
```

void push_back(int value)
Taxa de desempenho do método o(1)

Esse método tem como função adicionar o número desejado no final do vetor.

```
void push_front(int value) { // Adiciona um elemento no ``início'' do
vetor
        int_node *novo_no = new int_node;
        novo_no->value = value;
        novo_no->next = head;
        novo_no->prev = nullptr;
        if (this->head == nullptr) this->tail = novo_no;
        else this->head->prev = novo_no;
        this->head = novo_no;
        size_++;
    }
```

bool pop_back()
Taxa de desempenho do método o(1)

Esse método tem como função remover o último elemento do vetor.

```
   bool pop_back() { // Remove um elemento do ``final'' do vetor
     int retorno = true; // Crie uma variável retorno e ela é
verdadeira;
if (this->tail == nullptr) retorno = false; //se a cauda for nula
retorno será falso
  int_node *atual = this->tail; // Atual aponta para tail;

 this->tail = this->tail->prev; // Cauda apontará para o que vem antes
da cauda (prev);
        atual = atual->prev; // Isso serve para andar pela lista
        size_--; // Diminua o tamanho em 1;
        return retorno;
    }
```

bool pop_front()
Taxa de desempenho do método o(1)

Esse método tem como função remover o primeiro elemento do vetor.

```
bool pop_front() { // Remove um elemento do ``início'' do vetor
       int retorno = true; // Cria uma variável retorno e ela é
verdadeira;
        if (this->tail == nullptr) retorno = false; // Se a cauda for
nula retorno é falsa;
        int_node *atual = this->head; // Atual aponta para cabeça
        this->head = this->head->next;// Cabeça aponta para próxima
cabeça
        atual = atual->next; // Andando pela lista
        size_--;
        return retorno;
    }
```

int back()
Taxa de desempenho do método o(1)

Esse método tem como função retornar o último elemento do vetor.

```
    int back(){ // Retorna o elemento do ``final'' do vetor
            if (this->head == nullptr) return -1;
            return this->head->value;
    }
```

int front()
Taxa de desempenho do método o(1)

Esse método tem como função retornar o primeiro elemento do vetor.

```
    int front(){ // Retorna o elemento do ``início'' do vetor
            if (this->tail == nullptr) return -1;
            return this->tail->value;
    }
```

bool remove()
Taxa de desempenho do método o(n)

Esse método tem como função remover o primeiro elemento do vetor.

---

int find(int value)
Taxa de desempenho do método o(n)

Esse método tem como função procurar o elemento desejado no vetor.

```
    int find(int value) { // Retorna o índice de value, -1 caso value não esteja presente
        int encontrou = -1, pos = 0; // Crie uma variável encontrou e ela é -1 e pos e ela é 0;
        int_node *atual =  this->tail; // Crie atual e ele aponta para cauda;
        while (atual != nullptr){ // Enquanto atual não é nulo
            if (atual->value == value) { // Se esse valor for igual valor;
                encontrou = pos;// Encontrou vai ser pos;
                break;
            }
        atual = atual->next; // isso serve para andar pela lista;
        pos++; // Aumente posição;
        }
        return encontrou;
    }
```

int count(int value)
Taxa de desempenho do método o(n)

Esse método tem como função contar quantas vezes o elemento desejado está no vetor

---

```
    int count(int value) { // Retorna quantas vezes value ocorre no vetor
        int cont = 0; // Crie uma variável cont e ela é 0;
        int_node *atual = this->head; // Atual aponta para cabeça;
        while (atual != nullptr){ // Enquanto atual não é nulo;
            if (atual->value == value) cont++;// Se esse valor for igual valor,cont aumenta em 1 ;
        atual = atual->next;  // isso serve para andar pela lista
        }
        return cont;
    }
```

int sum()
Taxa de desempenho do método o(n)

Esse método tem como função realizar a soma total do vetor.

```
 int sum() { // Retorna a soma dos elementos do vetor
        int soma = 0; // Crie uma variável soma e ela é 0;
        int_node *atual = this->head; // Atual aponta para cabeça;
        while(atual != nullptr){ // Enquanto atual não é nulo;
            soma = soma + atual->value; // isso serve para dizer o
valor de atual e somar os elementos da lista;
            atual = atual->next; // isso serve para andar pela lista;
        }
        return soma ;
    }
```

04. Testes

Esse espaço é dedicado à realização dos diversos casos de testes consistentes, os quais serão realizados e analisados devidamente. Segue abaixo o nome dos arquivos teste e a quantidade de números a serem testados, bem como gráficos representativos da sua taxa de desempenho..

04. 01 Testes com Alocação Dinâmica

Testando o push_front()

Push_front()/Push_back()

e1.txt → quantidade de números : 5
e2.txt → quantidade de números : 10
e3.txt → quantidade de números : 1000
e4.txt → quantidade de números : 2000
e5.txt → quantidade de números : 3000
e6.txt → quantidade de números : 10000
e7.txt → quantidade de números : 100000
e8.txt → quantidade de números : 500000

Tempo de execução (em ms) **Casos de teste**	**100 em 100**	**1000 em 1000**	**Começa em 8 e vai dobrando**
e1.txt	**6900**	**3400**	**3700**
e2.txt	**5800**	**4000**	**9100**

e3.txt	**3307100**	**1843000**	**2734800**
e4.txt	**9450900**	**10030800**	**9760000**
e5.txt	**194384000**	**20971100**	**22174900**
e6.txt	**140341800**	**124714000**	**129748400**
e7.txt	**13320417800**	**12945086800**	**1421670070**
e8.txt	**Time Limit Exceeded**	**Time Limit Exceeded**	**Time Limit Exceeded**

Gráfico push_front → testes e1.txt até e7.txt (de 100 em 100)

Gráfico push_front → testes e1.txt até e7.txt (de 1000 em 1000)


Tempo de execução
1,5E+10
1,0E+10
5,0E+9
0

Gráfico push_front → testes e1.txt até e7.txt (começa em 8 e vai dobrando)

Testando o push_back()

Tempo de execução (em ms) Casos de teste	100 em 100	1000 em 1000	Começa em 8 e vai dobrando
e1.txt	3500	3200	3300
e2.txt	5000	6700	8000
e3.txt	217800	192500	202600
e4.txt	531700	396900	442900
e5.txt	764700	773000	633400

e6.txt	**2676100**	**2329800**	**3088700**
e7.txt	**55289100**	**53215400**	**28188700**
e8.txt	**5644609200**	**817792400**	**148995100**

Gráfico push_back → testes e1.txt até e8.txt (de 100 em 100)

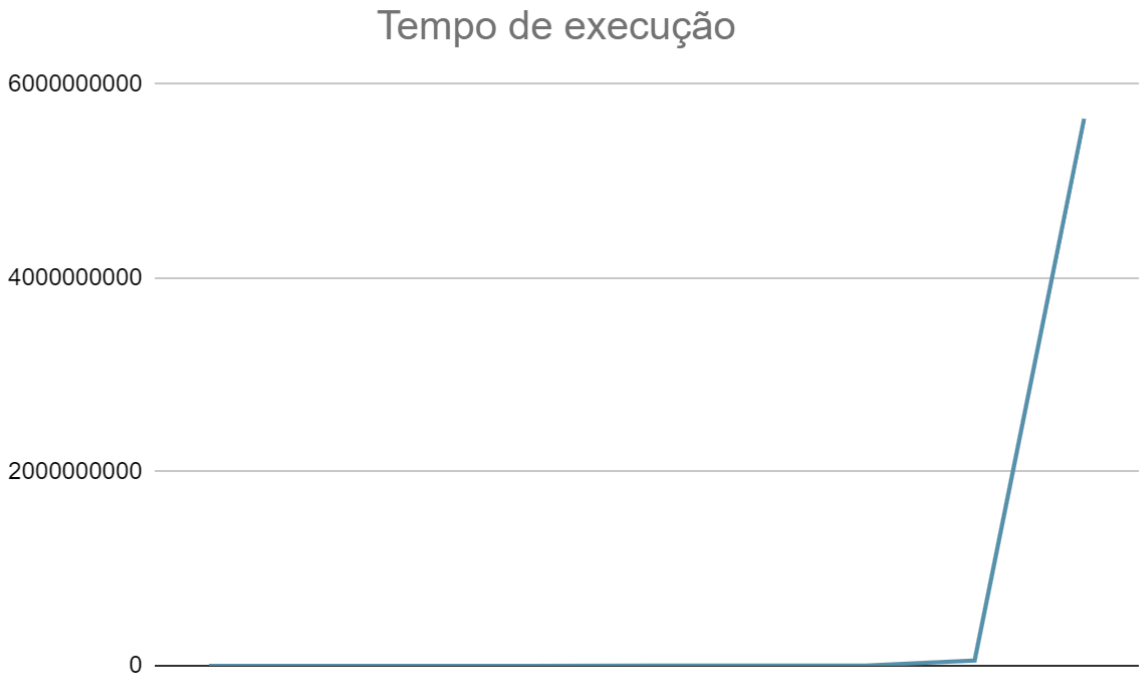


Gráfico push_back → testes e1.txt até e8.txt (de 1000 em 1000)

Gráfico push_back → testes e1.txt até e8.txt (começa em 8 e vai dobrando)

Testando o remove_at()

remove_at()

r1.txt → quantidade de números : 15
r2.txt → quantidade de números : 15
r3.txt → quantidade de números : 100
r4.txt → quantidade de números : 1000
r5.txt → quantidade de números : 10000
r6.txt → quantidade de números : 100000

Tempo de execução Casos de teste	Tentativa de remover elementos	Elementos removidos	Não removidos	Tempo de execução (em ms)
r1.txt	10	10	0	4900
r2.txt	10	8	2	5200
r3.txt	50	19	31	45600
r4.txt	100	90	10	437800
r5.txt	1000	1000	0	46295400
r6.txt	10000	10000	0	1810892300

Gráfico remove_at() → testes r1.txt até r6.txt

Testando o pop_front()

pop_front()/pop_back()

e1.txt → quantidade de números : 5
e2.txt → quantidade de números : 10
e3.txt → quantidade de números : 1000
e4.txt → quantidade de números : 2000
e5.txt → quantidade de números : 3000
e6.txt → quantidade de números : 10000
e7.txt → quantidade de números : 100000
e8.txt → quantidade de números : 500000

Tempo de execução (em ms) **Casos de teste**	**100 em 100**	**1000 em 1000**	**Começa em 8 e vai dobrando**
e1.txt	**1200**	**500**	**500**
e2.txt	**900**	**600**	**800**

e3.txt	**1027100**	**1163400**	**1037500**
e4.txt	**4216500**	**4218300**	**4087900**
e5.txt	**11032800**	**9424200**	**9745600**
e6.txt	**106210600**	**105561000**	**104495000**
e7.txt	**10474777200**	**10480347900**	**10473069400**
e8.txt	**Time Limit Exceeded**	**Time Limit Exceeded**	**Time Limit Exceeded**

Gráfico pop_front() → testes e1.txt até e7.txt (100 em 100)

Gráfico pop_front → testes e1.txt até e7.txt (1000 em 1000)

Gráfico pop_front → testes e1.txt até e7.txt
(Começa em 8 e vai dobrando)

Testando o pop_back()

Tempo de execução (em ms) Casos de teste	100 em 100	1000 em 1000	Começa em 8 e vai dobrando
e1.txt	1100	1000	1000
e2.txt	1700	1100	1100
e3.txt	11800	6700	12100
e4.txt	19700	12400	17900
e5.txt	20000	17800	35700
e6.txt	62600	118000	57500
e7.txt	638100	1004200	562600
e8.txt	Time Limit Exceeded	1395200	3200600

Gráfico pop_back() → testes e1.txt até e7.txt (100 em 100)

Gráfico pop_back() → testes e1.txt até e8.txt

(1000 em 1000)

Gráfico pop_back() → testes e1.txt até e8.txt (Começa em 8 e vai dobrando)

04. 02 Testes com Lista Ligada

Testando o push_front()

Push_front()/Push_back()

e1.txt → quantidade de números : 5
e2.txt → quantidade de números : 10
e3.txt → quantidade de números : 1000
e4.txt → quantidade de números : 2000
e5.txt → quantidade de números : 3000
e6.txt → quantidade de números : 10000
e7.txt → quantidade de números : 100000

Casos de teste	Tamanho da entrada	Tempo de execução (em ms)
e1.txt	5	8800
e2.txt	10	7400
e3.txt	1000	551900

e4.txt	**2000**	**1172200**
e5.txt	**3000**	**2126700**
e6.txt	**10000**	**7786300**
e7.txt	**100000**	**73325800**
e8.txt	**500000**	**240580400**

Gráfico push_front() → testes e1.txt até e8.txt

Testando o push_back()

Casos de teste	Tamanho da entrada	Tempo de execução (em ms)
e1.txt	5	10600
e2.txt	10	13700
e3.txt	1000	560000
e4.txt	2000	1191200
e5.txt	3000	1769200
e6.txt	10000	3100400
e7.txt	100000	72522400
e8.txt	500000	223625600

Gráfico push_back() → testes e1.txt até e8.txt

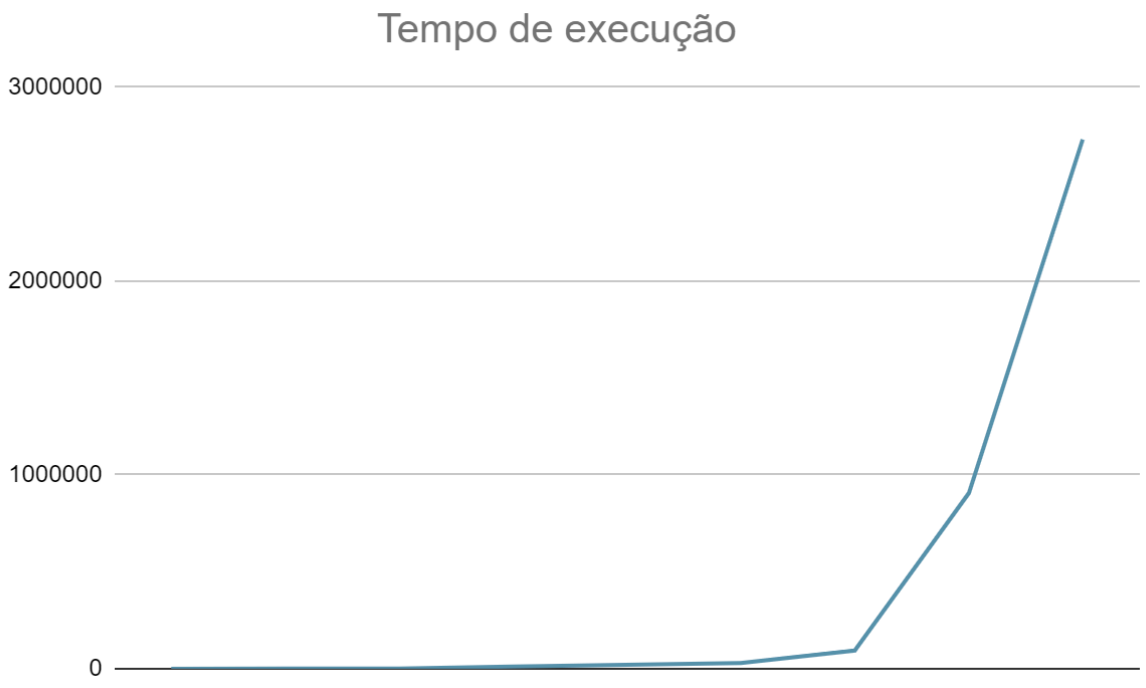
Testando o pop_front()

pop_front()/pop_back()

e1.txt → quantidade de números : 5
e2.txt → quantidade de números : 10
e3.txt → quantidade de números : 1000
e4.txt → quantidade de números : 2000
e5.txt → quantidade de números : 3000
e6.txt → quantidade de números : 10000
e7.txt → quantidade de números : 100000

Casos de teste	Tamanho da entrada	Tempo de execução (em ms)
e1.txt	5	1100
e2.txt	10	1300
e3.txt	1000	10900
e4.txt	2000	20100
e5.txt	3000	29900
e6.txt	10000	94000
e7.txt	100000	906100
e8.txt	500000	2730100

Gráfico pop_front() → testes e1.txt até e8.txt



Testando o pop_back()

Casos de teste	Tamanho da entrada	Tempo de execução (em ms)
e1.txt	5	600
e2.txt	10	500
e3.txt	1000	5100
e4.txt	2000	9800
e5.txt	3000	13100
e6.txt	10000	52800
e7.txt	100000	413400
e8.txt	500000	2225100

Gráfico pop_back() → testes e1.txt até e8.txt

05. Resultados

Métodos	Taxa de desempenho array_list	Taxa de desempenho linked_list
push_front()	**O(N)**	**O(1)**
push_back()	**O(N)**	**O(1)**
pop_back()	**O(1)**	**O(1)**
pop_front()	**O(N)**	**O(1)**
remove_at()	**O(N)**	**O(n)**

05. Arraylist x Lista duplamente ligada

Para o Arraylist foi levado em consideração o caso de teste: Começa em 8 e vai dobrando quando for necessário.

Gráfico para o método push_front()

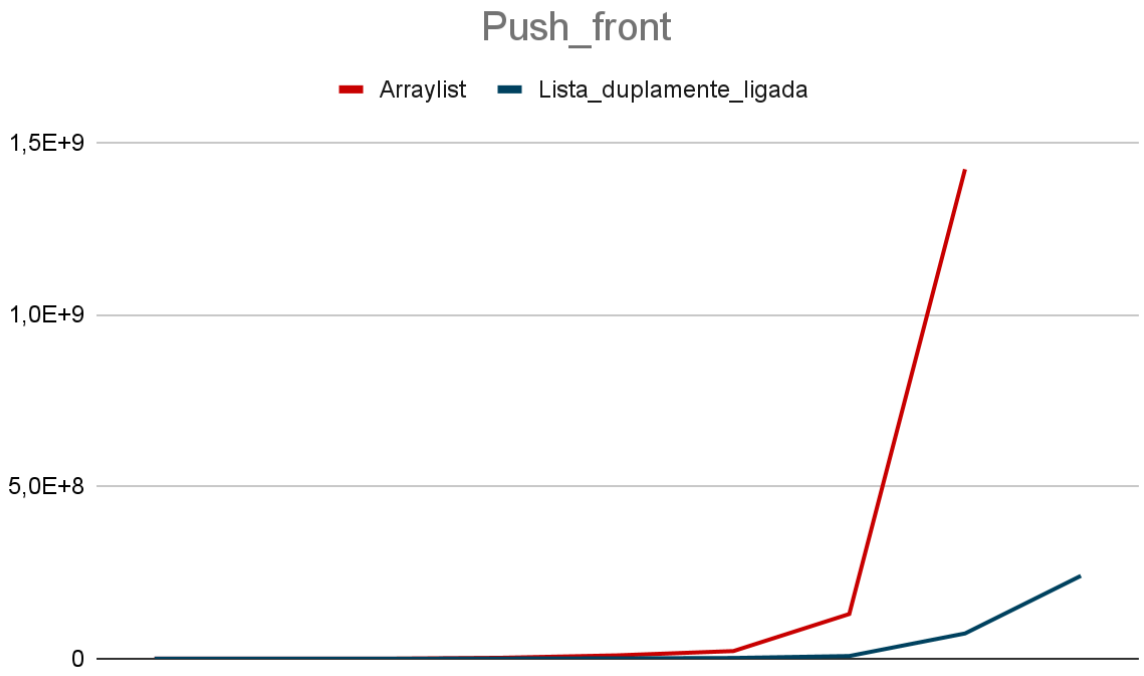


Gráfico para o método push_back()

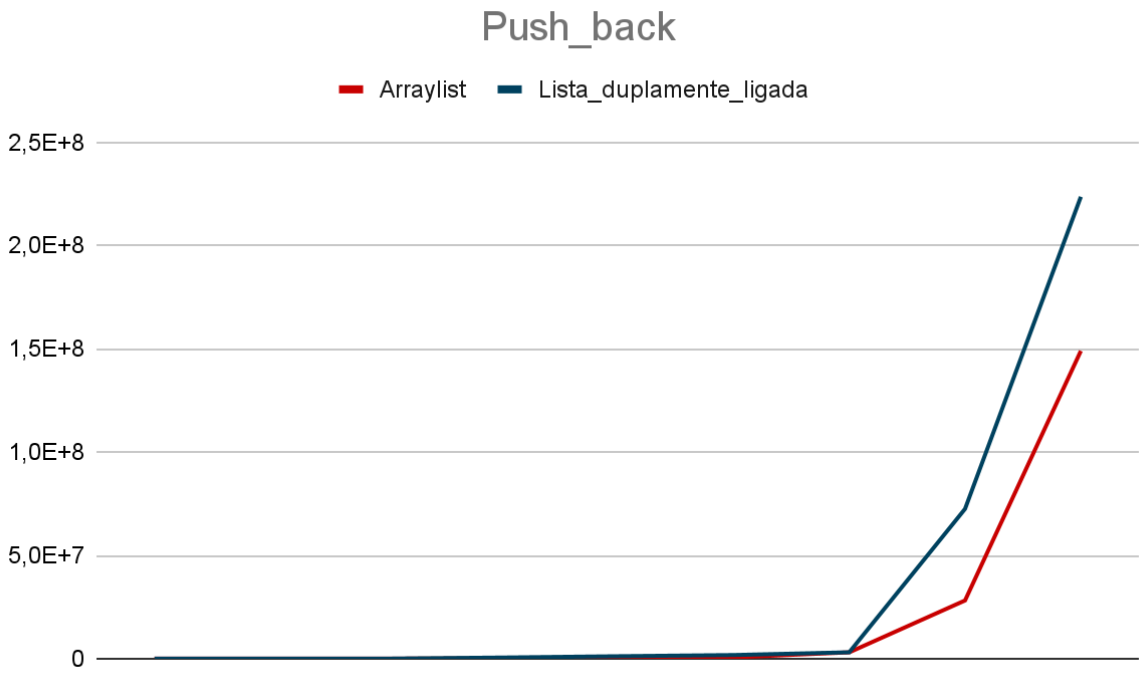


Gráfico para o método pop_back()

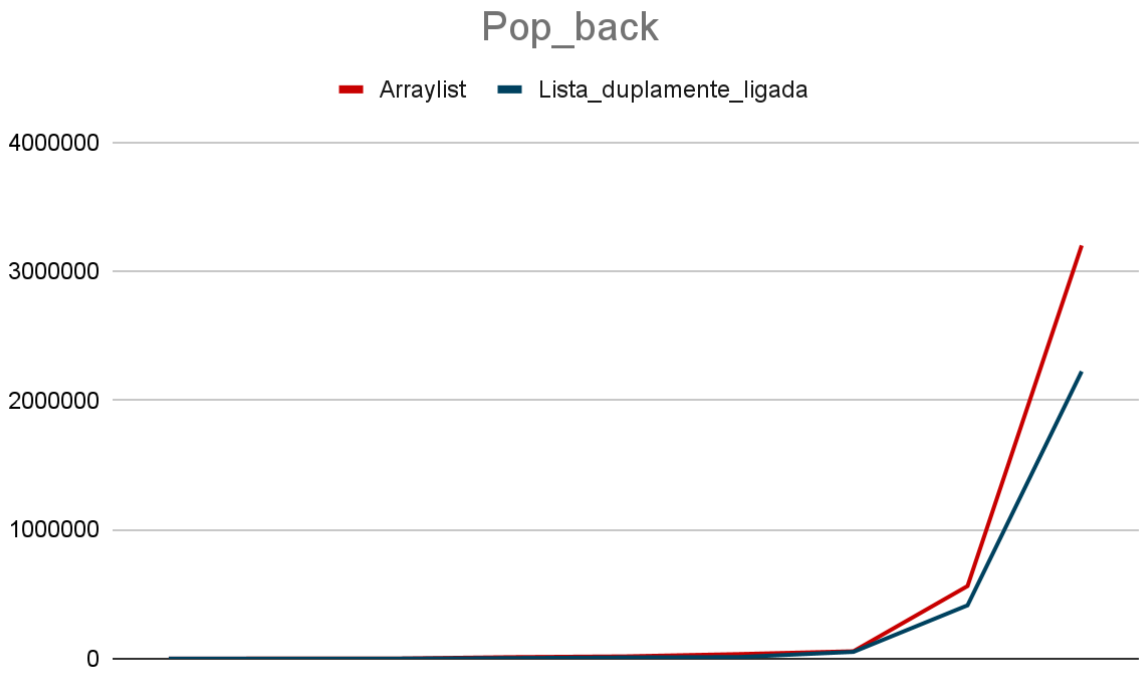
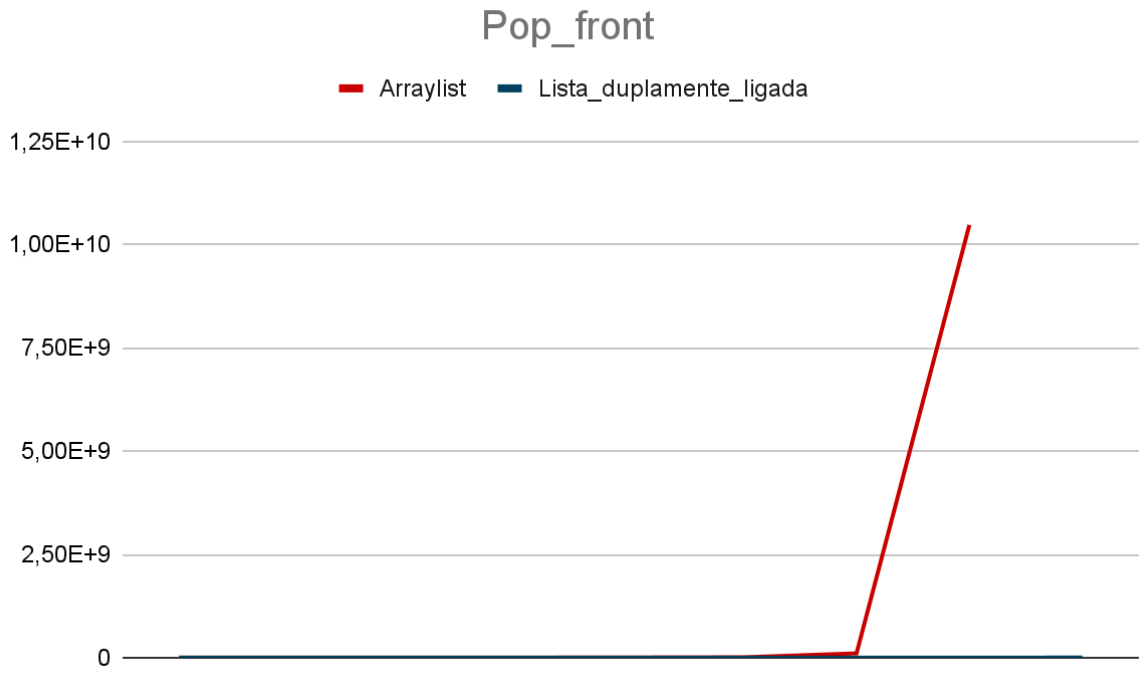


Gráfico para o método pop_front()



06. Conclusão

Com Isso após os diversos testes usados tanto como em Arraylist como em Lista duplamente ligada, chegamos a conclusão que para métodos como inserir elemento no início de um vetor, inserir elemento no final de um vetor, remover um elemento do início de um vetor e remover um elemento no final do vetor é mais eficiente a utilização de lista duplamente ligada, porém caso o usuário queira remover um elemento, ou remover um elemento pelo índice a utilização de Arraylist se torna mais rápida e eficaz.