Relatório do trabalho

Vetor dinâmico e lista ligada

Ana Maria Ferreira de Abreu Guedes

04 de agosto de 2024

INSTITUTO FEDERAL
Rio Grande do Norte
Campus Natal-Central

TADS

DIATINF

Sumário

01. Introdução

Nesse relatório será apresentado a implementação de um vetor dinâmico e de uma lista ligada na linguagem C + +. Ambos, foram implementados usando métodos ligados a classe contida no em um arquivo com extensão .hpp.

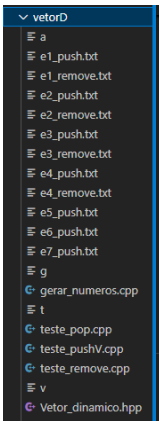
Esse relatório possuirá como foi implementado o código, tão quanto sua eficiência e a explicação dos métodos usados em cada classe.

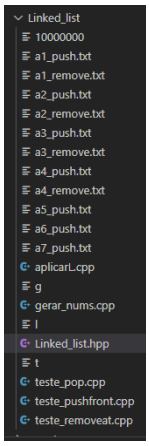
Vetor dinâmico: Através de um array são usados métodos de manipulação como por exemplo o de inserção de elementos, que por sua vez utilizam alocação dinâmica de memória.

Lista ligada: Outra forma de armazenamento de dados que não se organizam de forma contínuas na memória, utilizando ponteiros em cada nó (valor armazenado e ponteiro para o próximo elemento) para conseguir o acesso aos elementos, não necessitando o uso de alocação dinâmica de memória mas implicando na eficiência em alguns metódos.

01.2 Organização do código fonte

Como citado anteriormente, houve a criação de dois tipos de arquivos para a implementação do vetor dinâmico e da lista ligada sendo eles um **.cpp** para o código principal que tem como finalidade testar e um **.hpp** que possui os códigos dos métodos implementados.





Além dos códigos principais temos os arquivos gerar_numero.cpp, teste_pushV.cpp, teste_remove.cpp em vetor dinâmico e ... para lista ligada e todos eles possuem a finalidade de testes com diferentes tamanhos de entrada e de aumento da capacidade, no caso do vetor dinâmico. Os arquivos **.txt** contém os diferentes tipos de inputs testados em cada implementação.

02. Vetor dinâmico

Para execução dessa parte do trabalho houve a criação de dois arquivos, sendo um .cpp e outro .hpp, que estarão anexados juntamente ao relatório. No qual ,respectivamente, um serve para o teste dos métodos e o outro para a aplicação dos atributos e métodos.

02.2 Estrutura do .hpp

```
#ifndef ARRAY_LIST_IFRN
#define ARRAY_LIST_IFRN
class Vetor_dinamico{
	private:
		// Atributos
	public:
		//Metódos
};
#endif
```

Observações: O }; na penúltima linha é obrigatório. Como no código principal temos o #include "Vetor_dinamico.hpp" ele pega o código que está no arquivo .hpp e coloca no arquivo principal e o #ifndef ARRAY_LIST_IFRN e o #define ARRAY_LIST_IFRN servem exatamente para caso haja repetição desse mesmo código ele compilar apenas uma vez, então if not define(ifndef) define e compila a classe mas caso já esteja definida não inclui. O # É uma diretriz de compilação em cpp, ou seja, ao usar ele estamos incluindo no código naquele arquivo.

02.3 Atributos

Os atributos usados foram, juntamente com o seu tipo:

Unsigned int: size_, capacity_

Int: *array

Para a declaração dos atributos:

```
class Vetor_dinamico{

    //Atributos e incease capacity

    private:

        int *array;

        int size_;

        int capacidade_;

}
```

02.4 Métodos

Os métodos aplicados foram, juntamente com o seu tipo:

Unsigned int: size(), capacity()

Int: back(), count(int value),find(int value), front(), get_at(unsigned int index), sum()

Void: clear(), push_back(int value), push_front(int value)

Bool: insert_at(unsigned int index, int value), pop_back(), pop_front(), remove(int value), remove_at(unsigned int index)

Double: percent_occupied( )

Increase_capacity()

O método do increase capacity consiste em alocação da memória para que assim consigamos inserir mais elementos no array e caso a capacidade não seja suficiente essa função entra em ação.

```
void increase_capacity(){ //auemntar capacidade
            capacidade_+= 100;
            int *novo = new int[capacidade_];
            for (int i = 0; i < size_; i++){
                novo[i] = array[i];
            }
            delete [] array; //apagar o antigo
            array = novo; //atribuir a variável
        }
```

Big Oh da função: *O(n)*

Explicação do código: Comecei fazendo o aumento da capacidade para que assim na linha seguinte eu pudesse criar um novo array com uma capacidade maior. Em seguida fiz um laço para salvar os elementos que possuía até agora, no array com mais capacidade e deleto o array antigo, que possui uma capacidade menor e salvo o array novo no mesmo nome que o antigo.

~Vetor_dinamico() e Vetor_dinamico()

Esses métodos correspondem respectivamente ao construtor e destrutor

```
Vetor_dinamico(){ //construtor
            capacidade_ = 100;
            size_ = 0;
            array = new int[capacidade_];
        }
        ~Vetor_dinamico(){ //destrutor
            delete [] array;
        }
```

Explicação do código: O construtor reinicia os valores da capacidade_ e size_, e posteriormente cria um novo array com a capacidade reiniciada.

Back()

O método retira o último elemento do array.

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

Big Oh da função: *O(1)*

Explicação do código: Retornei o último elemento da lista mas coloquei size - 1 pois estamos lidando com índice.

Capacity()

O método retorna a capacidade do array, ou seja, quantos espaços de memórias eu tenho alocado.

```
unsigned int capacity(){
            return this->capacidade_;
        }
```

Big Oh da função: *O(1)*

Explicação do código: Retornamos o atributo capacidade_;

Clear()

O método retira todos os elementos do array.

```
void clear(){
            size_ = 0; //podia ser só isso
            capacidade_ = 100;
            delete [] this->array;
            array = new int[capacidade_];
        }
```

Big Oh da função: *O(n)*

Explicação do código: Reinicia os valores da capacidade_ e size_, e posteriormente cria um novo array com a capacidade original.

Count(int v)

O método retorna quantas vezes o elemento repetiu.

```
int count(int v){
            int c = 0;
            for (int i = 0; i < this->size_; i++) if (array[i] == v) c++;
            return c;
        }
```

Big Oh da função: *O(n)*

Explicação do código: Crio um contador e um loop que verifica se o número que está no array naquela posição é o valor apontado como parâmetro o contador incrementa +1.

Find(int v)

O método retorna o índice do elemento no array.

```
int find(int v){
            int ind = -1;
            for (int i = 0; i < this->size_; i++){
                if (array[i] == v){
                    ind = i;
                    break;
                }
            }
            return ind;
        }
```

Big Oh da função: *O(n)*

Explicação do código: Faço um laço até encontrar o elemento e eu atribuir a variável ind ao índice e caso não encontre retorna -1.

Front()

O método retorna o primeiro elemento do array.

```
int front(){
            return this->array[0];
        }
```

Big Oh da função: *O(1)*

Explicação do código: Retorno o array na primeira posição.

Get_at(unsigned int index)

O método retorna o elemento que está no índice dado.

```
int get_at(unsigned int index){
            if (index < 0 or index > this->size_) return -1;
            return this->array[index];
        }
```

Big Oh da função: *O(1)*

Explicação do código: Faço um if primeiro para verificar se o índice dado é válido.

Insert_at(int index, int v)

O método insere o elemento no índice dado.

```
bool insert_at(int index, int v){

            if (index > this->size_ or index < 0) return false;

            if (this->capacidade_ == this->size_ ) increase_capacity();

            int atual = this->array[index], proximo = this->array[index + 1], ultimo =
this->array[size_ - 1];

            this->array[index] = v;

            for (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 true;

        }
```

Big Oh da função: *O(n)*

Explicação do código: Verificar primeiro se o índice dado é válido caso não seja retorno false mas caso seja faço um laço para mover os elementos do array atual na posição index até o fim para uma casa a frente e no espaço que ficou livre a na posição index coloquei o valor no índice pedido.

Operator[ ]()

O método retorna o elemento que está no índice dado mas no .cpp conseguirei usar o [ ] em vez do .nome_função( ) .

```
int operator[](int index){
            return this->array[index];
        }
```

Big Oh da função: *O(1)*

Explicação do código: Ele possui a mesma função que o get_at mas no arquivo de teste .cpp posso usar o operador [ ] para acessar a função.

Percent_occupied()

O método retorna a porcentagem ocupada do capacity .

```
double percent_occupied(){
            if (this->capacidade_ == this->size_) return 1.0;
            else return (double)(double(this->size_) / double(this->capacidade_));
        }
```

Big Oh da função: *O(1)*

Explicação do código: Preciso inicialmente transformar o size e a capacidade em double para que dessa forma ao fazer a divisão para descobrir a porcentagem ocupada eu tenha uma resposta em ponto flutuante

Pop_back()

O método elimina o último elemento do array .

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

Big Oh da função: *O(1)*

Explicação do código: Apenas conferir primeiro se o array não está vazio e caso não esteja só diminuo o size

Pop_front()

O método elimina o primeiro elemento do array .

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

Big Oh da função: *O(n)*

Explicação do código: Faço um laço para mover todos os elementos para o índice anterior e diminuo o size.

Push_back()

O método insere um elemento no fim do array .

```
void push_back(int v){
            if (this->capacidade_ == this->size_) increase_capacity();
            array[size_] = v;
            size_++;
        }
```

Big Oh da função: *O(1)*

Explicação do código: Confiro se o size é igual a capacidade e caso seja chamo o método incease capacity para aumentar a capacidade do meu novo array e assim insiro o novo elemento e aumento o size

Push_front()

O método insere um elemento no início do array .

```
void push_front(int v){
            if (this->size_ == this->capacidade_) increase_capacity();
            this->size_++;
            for (int i = this->size_ - 1; i > 0; i--){
                this->array[i] = this->array[i - 1];
            }
            this->array[0] = v;
        }
```

Big Oh da função: $O(n)$

Explicação do código: Preciso fazer um laço para mover todos os elementos do array para um índice para a frente e no índice 0 inserimos o valor dado

Remove(int v)

O método remove o elemento dado no V.

```
bool remove(int v){
            for (int i = 0; i < this->size_; i++){
                if (this->array[i] == v){
                    for (int j = i; j < this->size_ - 1; j++) this->array[j] =
this->array[j + 1];
                    this->size_--;
                    return true;
                }
            }
            return false;
        }
```

Big Oh da função: $O(n^2)$

Explicação do código: Ele faz um primeiro laço que procura o índice do elemento dado e quando o mesmo é encontrado outro laço ocorre que serve para a partir daquele índice colocar o elemento em uma posição anterior.

Remove_at(unsigned int index)

O método remove o elemento no índice que foi dado .

```
 bool remove_at(unsigned int index){

            if (index > this->size_ or index < 0) return false;

            this->size_--;

            for (int i = index; i < this->size_; i++) this->array[i] = this->array[i +
1];

            return true;

        }
```

Big Oh da função: *O(n)*

Explicação do código: Faz a mesma coisa que o remove a diferença é a extinção do laço externo pois não precisaremos procurar o índice do elemento dado.

Size()

O método retorna quantos elementos possui no array .

```
unsigned int size(){

            return this->size_;

        }
```

Big Oh da função: *O(1)*

Explicação do código: Retorna o atributo size.

Sum()

A função retorna a soma de todos os elementos do array .

```
int sum(){
            int s = 0;
            for(int i = 0; i < this->size_;i++) s += this->array[i];
            return s;
        }
```

Big Oh da função: *O(n)*

Explicação do código: Faz um laço que incrementa em uma variável acumulador cada elemento do array.

02.5 Testes vetor dinâmico

Essa sessão é destinada aos testes dos arquivos anexados à proposta do trabalho aqui apresentado.

Test remove_at

Tamanho de entrada Aumento	e1.txt	e2.txt	e3.txt	e4.txt
100	4000	4200	72700	Tempo excedido
1000	2700	2100	127100	Tempo excedido
Duplicar	2300	4400	76500	Tempo excedido

E1.txt

15, 1, 3, 5, 4, 2, 12, 89, 32, 12, 32, 8, 6, 9, 1, 2, 10, 11, 9, 11, 5, 4, 3, 8, 4, 3, 2

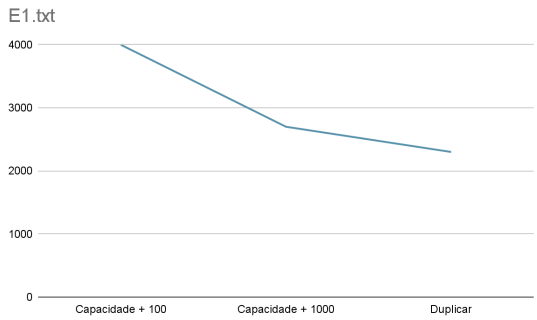
Saída do primeiro teste:

[DEBUG] Tried to remove 10 element(s)

[DEBUG] removed 10 element(s)

[DEBUG] not removed 0 element(s)

[INFO] Elapsed time for 10 remove_at success : 4000



[DEBUG] Tried to remove 10 element(s)

[DEBUG] removed 10 element(s)

[DEBUG] not removed 0 element(s)

[INFO] Elapsed time for 10 remove_at success : 2700

[DEBUG] Tried to remove 10 element(s)

[DEBUG] removed 10 element(s)

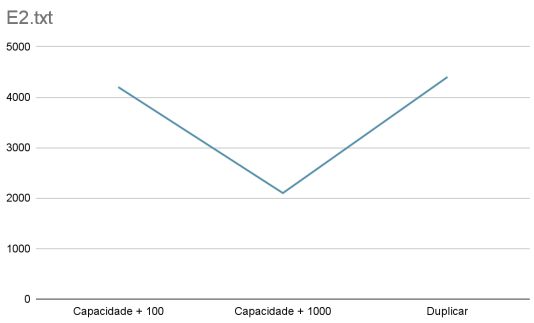
[DEBUG] not removed 0 element(s)

[INFO] Elapsed time for 10 remove_at success : 2300

E2.txt

15, 1, 3, 5, 4, 2, 12, 89, 32, 12, 32, 8, 6, 9, 1, 2, 10, 11, 15, 9

Saída do segundo teste:



[DEBUG] Tried to remove 10 element(s)

[DEBUG] removed 8 element(s)

[DEBUG] not removed 2 element(s)

[INFO] Elapsed time for 8 remove_at success : 4200

[DEBUG] Tried to remove 10 element(s)

[DEBUG] removed 8 element(s)

[DEBUG] not removed 2 element(s)

[INFO] Elapsed time for 8 remove_at success : 2100

[DEBUG] Tried to remove 10 element(s)

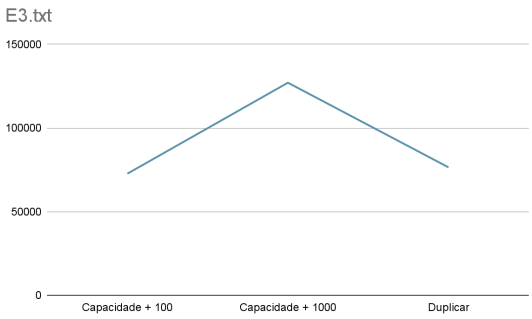
[DEBUG] removed 8 element(s)

[DEBUG] not removed 2 element(s)

[INFO] Elapsed time for 8 remove_at success : 4400

E3.txt

....



Tamanho 10000

Saída do primeiro teste:

[DEBUG] Tried to remove 3 element(s)

[DEBUG] removed 3 element(s)

[DEBUG] not removed 0 element(s)

[INFO] Elapsed time for 3 remove_at success : 72700

[DEBUG] Tried to remove 3 element(s)

[DEBUG] removed 3 element(s)

[DEBUG] not removed 0 element(s)

[INFO] Elapsed time for 3 remove_at success : 127100

[DEBUG] Tried to remove 3 element(s)

[DEBUG] removed 3 element(s)

[DEBUG] not removed 0 element(s)

[INFO] Elapsed time for 3 remove_at success : 76500

E4.txt

...

Tamanho 2200000

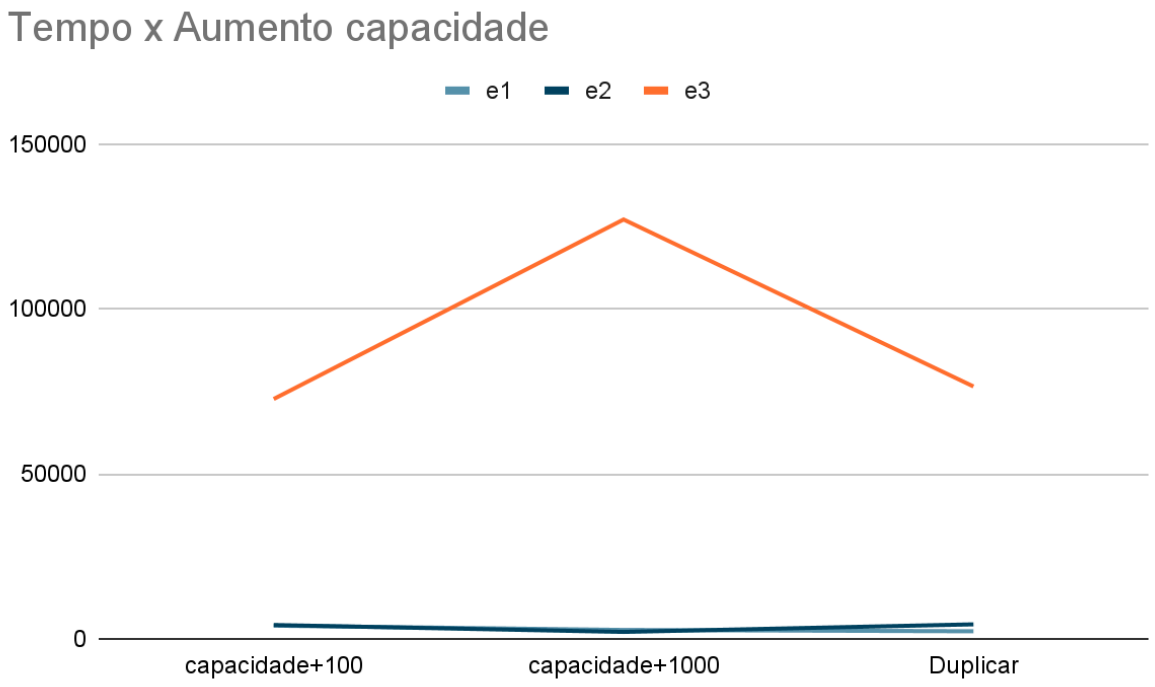
Saída do quarto teste:

Tempo limite excedido

Gráfico do Tempo de execução x Aumento da capacidade

O gráfico representa para cada caso teste o tempo de execução em cada aumento de capacidade.

Obs: O tempo não foi convertido para segundo pois os valores ficam muito pequenos não ficando uma boa visualização dos valores.



Test push_front

Tamanho Aumento	e1.txt	e2.txt	e3.txt	e4.txt	e5.txt	e6.txt	e7.txt
100	1600	7300	2531300	6564900	19108801	161197399	Tempo excedido
1000	2900	4900	2485900	7120200	19121099	162868909	Tempo excedido
Duplicar	3200	4200	1986300	8408000	22011201	149157697	Tempo excedido

E1.txt

5, 1, 3, 5, 4, 2

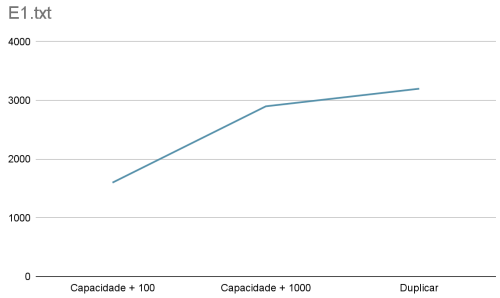
Tamanho do input: 5

Saída do primeiro teste:

[INFO] Elapsed time for 5 pushes front :1600

[INFO] Elapsed time for 5 pushes front :2900

[INFO] Elapsed time for 5 pushes front :3200



E2.txt

10, 15, 7, 2, 20, 17, 18, 20, 13, 5, 13

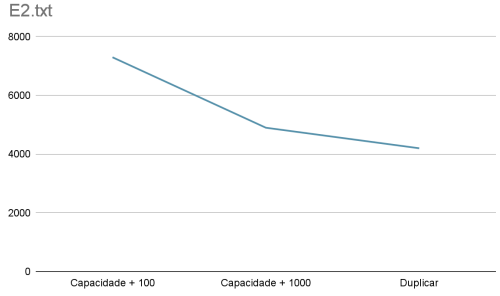
Tamanho do input: 10

Saída do segundo teste:

[INFO] Elapsed time for 10 pushes front :7300

[INFO] Elapsed time for 10 pushes front :4900

[INFO] Elapsed time for 10 pushes front :4200



E3.txt

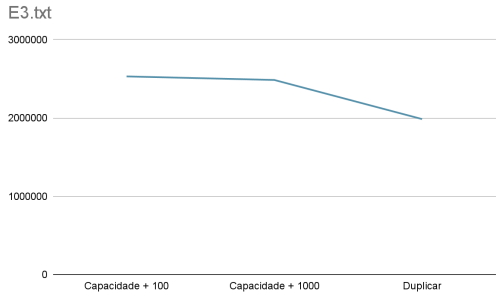
Tamanho do input: 1000

Saída do terceiro teste:

[INFO] Elapsed time for 1000 pushes front :2531300

[INFO] Elapsed time for 1000 pushes front :2485900

[INFO] Elapsed time for 1000 pushes front :1986300



E4.txt

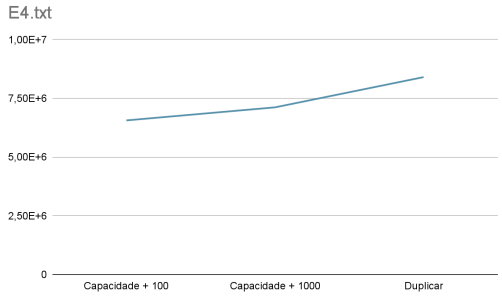
Tamanho do input: 2000

Saída do quarto teste:

[INFO] Elapsed time for 2000 pushes front :6564900

[INFO] Elapsed time for 2000 pushes front :7120200

[INFO] Elapsed time for 2000 pushes front :8408000



E5.txt

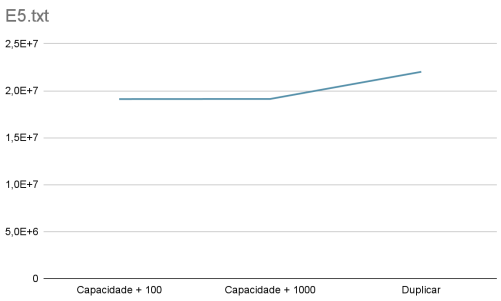
Tamanho do input: 3000

Saída do quinto teste:

[INFO] Elapsed time for 3000 pushes front :19108801

[INFO] Elapsed time for 3000 pushes front :19121099

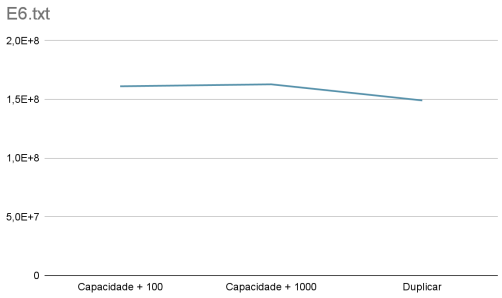
[INFO] Elapsed time for 3000 pushes front :22011201



E6.txt

...

Tamanho do input: 10000



Saída do sexto teste:

[INFO] Elapsed time for 10000 pushes front :161197399

[INFO] Elapsed time for 10000 pushes front :162868909

[INFO] Elapsed time for 10000 pushes front :149157697

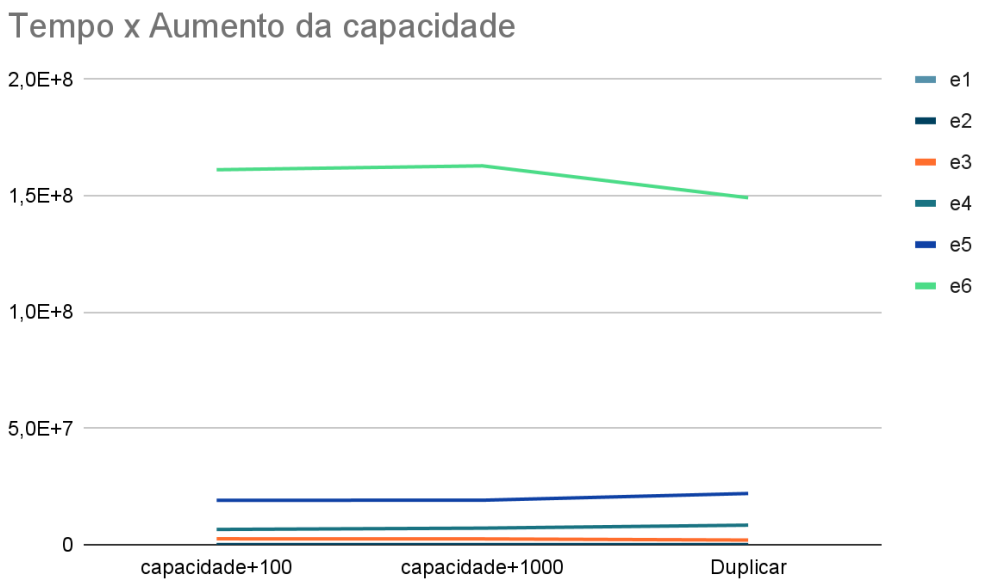
E7.txt

Tamanho do input: 10000000

Saída do sétimo teste:

Tempo limite excedido

Gráfico do Tempo de execução x Aumento da capacidade



Os testes a seguir foram realizados com os mesmo arquivos e casos testes dados para o push_front e o remove, só havendo alteração no método chamado:

Test push_back

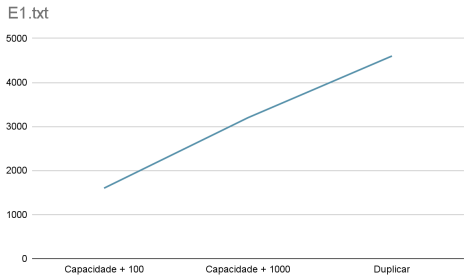
Tamanho Aumento	e1.txt	e2.txt	e3.txt	e4.txt	e5.txt	e6.txt	e7.txt
100	1600	3700	274300	939300	9212498	21461006	Tempo excedido
1000	3200	2600	345600	597300	882500	4923101	Tempo excedido
Duplicar	4600	11900	449200	1016400	1442100	4809500	Tempo excedido

Saída do e1.txt:

[INFO] Elapsed time for 5 pushes front :1600

[INFO] Elapsed time for 5 pushes front :3200

[INFO] Elapsed time for 5 pushes front :4600

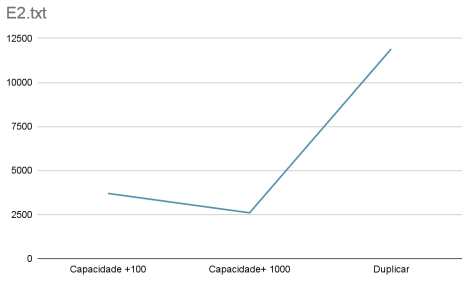


Saída do e2.txt:

[INFO] Elapsed time for 10 pushes front :3700

[INFO] Elapsed time for 10 pushes front :2600

[INFO] Elapsed time for 10 pushes front :11900

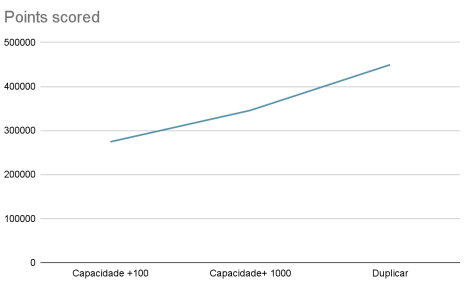


Saída do e3.txt:

[INFO] Elapsed time for 1000 pushes front :274300

[INFO] Elapsed time for 1000 pushes front :345600

[INFO] Elapsed time for 1000 pushes front :449200

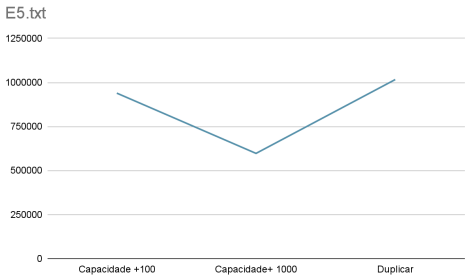


Saída do e4.txt:

[INFO] Elapsed time for 2000 pushes front :939300

[INFO] Elapsed time for 2000 pushes front :597300

[INFO] Elapsed time for 2000 pushes front :1016400

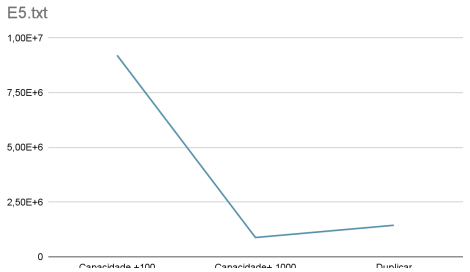


Saída do e5.txt:

[INFO] Elapsed time for 3000 pushes front :9212498

[INFO] Elapsed time for 3000 pushes front :882500

[INFO] Elapsed time for 3000 pushes front :1442100



Saída do e6.txt:

[INFO] Elapsed time for 10000 pushes front :21461006

[INFO] Elapsed time for 10000 pushes front :4923101

[INFO] Elapsed time for 10000 pushes front :4809500

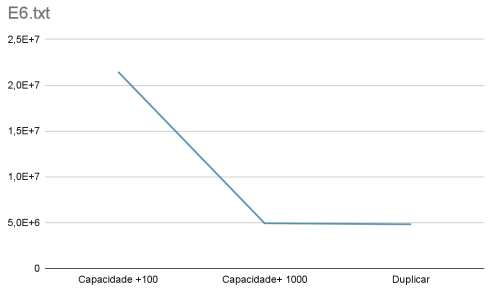
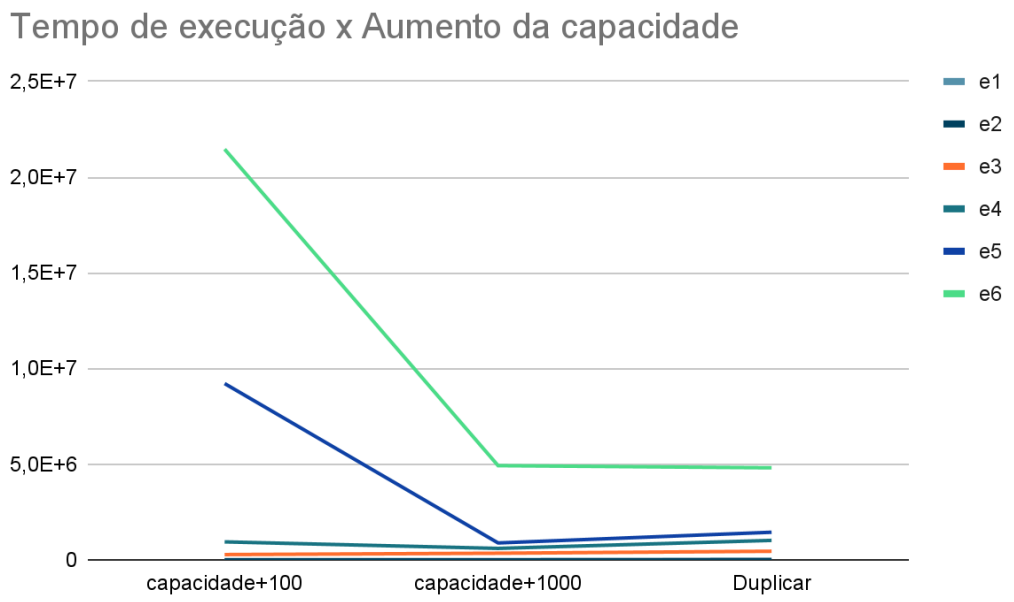


Gráfico do Tempo de execução x Aumento da capacidade



Test pop_back

Tamanho de entrada Aumento	e1.txt	e4.txt	e6.txt
100	0	0	0
1000	0	0	0
Duplicar	0	0	0

E1.txt

Tamanho do input: 5

Saída do primeiro teste:

Quantidade de números lidos: 4

Tamanho do vetor: 8

Tempo de processamento: 0 microseconds(s)

E4.txt

Tamanho do input: 2000

Saída do primeiro teste:

Quantidade de números lidos: 1999

Tamanho do vetor: 2048

Tempo de processamento: 0 microseconds(s)

E6.txt

Tamanho do input: 10000

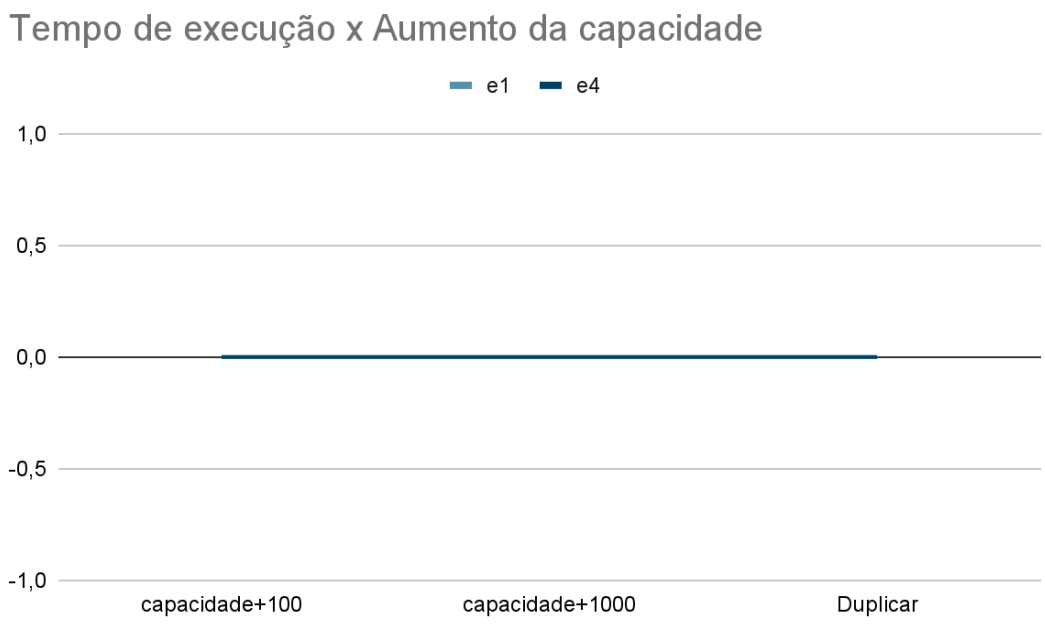
Saída do primeiro teste:

Quantidade de números lidos: 9999

Tamanho do vetor: 16384

Tempo de processamento: 0 microseconds(s)

Gráfico do Tempo de execução x Aumento da capacidade



03. Lista ligada

Para execução dessa parte do trabalho houve a criação de dois arquivos, sendo um .cpp e outro .hpp, que estarão anexados juntamente ao relatório. No qual ,respectivamente, um serve para o teste dos métodos e o outro para a aplicação dos atributos e métodos.

03.2 Criando um nó

```
struct Node{
            int valor;
            struct Node *next, *ant;
        };
```

03.3 Atributos

Os atributos usados foram, juntamente com o seu tipo:

Unsigned int: size_

Int: valor

Node: *next, *prev, *head, *tail

...

03.4 Métodos

Os métodos aplicados foram, juntamente com o seu tipo:

Unsigned int: size()

Int: back(), count(int value),find(int value), front(), get_at(unsigned int index), sum()

Void: clear(), push_back(int value), push_front(int value)

Bool: insert_at(unsigned int index, int value), pop_back(), pop_front(), remove(int value), remove_at(unsigned int index)

Os métodos possuem as mesmas características do vetor dinâmico, a diferença é apenas como os dados são armazenados e acessados, alterando assim o big oh do número.

Linked_list() e ~Linked_list()

Esses métodos correspondem respectivamente ao construtor e destrutor

```
Linked_list(){
            this->head = nullptr;
            this->tail = nullptr;
            this->size_ = 0;
        }
~Linked_list(){
            Node *percorrer = this->head;       //Preciso do primeiro nó para
assim só ir pegando o próximo
            while (percorrer != nullptr){
                Node *elemento = percorrer;
                percorrer = percorrer->next;
                delete elemento;
            }
        }
```

Explicação do código: O construtor inicia os valores do size_, do head e do tail, enquanto o destrutor faz um laço que passa deletando individualmente cada nó .

Back()

```
 int back(){
            return this->tail->valor;
        }
```

Big Oh da função: *O(1)*

Explicação do código: Como o tail aponta para o último nó da lista consigo pegar o valor do último elemento mais facilmente.

Clear()

```
void clear(){
            Node *elemento = this->head;
            while (elemento != nullptr){
                Node *apagar = elemento;
                elemento = apagar->next;
                delete apagar;
            }
        }
```

Big Oh da função: *O(n)*

Explicação do código: Ele possui um funcionamento parecido com o destrutor, passando em cada nó excluindo eles individualmente.

Count(int v)

```
int count(int v){
            Node *elemento = this->head;
            int cont = 0;
            while (elemento != nullptr){
                if (elemento->valor == v) cont++;
                elemento = elemento->next;
            }
            return cont;
        }
```

Big Oh da função: *O(n)*

Explicação do código: Com um loop que pega de nó em nó e conferindo se o valor armazenado naquele nó é igual ao valor dado.

Find(int v)

```
  int find(int v){
            int pos = 0;
            Node *elemento = this->head;
            while (elemento != nullptr){ //Pois depois do tail vai ser vazio
                if (elemento->valor == v) break;
                pos++;
                elemento = elemento->next;
            }
            if (pos == this->size_) return -1; //Pois isso significa que ele passou
por todos e não foi para o break
            return pos;
        }
```

Big Oh da função: *O(n)*

Explicação do código: É feito um loop que possui a finalidade de conferição dos elementos e no momento em que ele é encontrado é dado um break para ficar guardado na variável pos o “indicie”.

Front()

```
 int front(){
            return this->head->valor;
        }
```

Big Oh da função: *O(1)*

Explicação do código: Retorno do valor armazenado no head que é justamente o primeiro elemento inserido.

Get_at(unsigned int index)

```
int get_at(unsigned int index){
            Node *percorrer = this->head;
            int elemento = -1;
            unsigned int cont = 0;

            while (percorrer != nullptr){
                if (cont == index){
                    elemento = percorrer->valor;
                    break;
                }
                percorrer = percorrer->next;
                cont++;
            }
            return elemento;
        }
```

Big Oh da função: *O(n)*

Explicação do código: Como a lista ligada não armazena dados em endereços sequenciais na memória, acessar com o índice não é possível, por conta disso foi preciso fazer um laço para encontrar o elemento a partir de um índice.

Insert_at(int index, int v)

```
bool insert_at(unsigned int index, int v){
            unsigned int pos = 0;
            Node *elemento = this->head;
            while (elemento != nullptr){
                if (pos == index - 1){
                    Node *no = new Node;
                    no->valor = v;
                    no->next = elemento->next;
                    elemento->next = no; //Elemento atual vai apontar para o novo nó
                    no->ant = elemento;
                    this->size_++;
                    return true;
                }
                pos++;
                elemento = elemento->next;
            }
            return false;
        }
```

Big Oh da função: *O(n)*

Explicação do código: Ao encontrar o nó que está no índice apontado é feita a criação de um novo nó e a alteração dos ponteiros dos nós antes e depois dele.

Percent_occupied()

Como não fazemos o uso do atributo capacidade para o controle de quanto espaço de memória tenho reservado que posso colocar novos elementos, a porcentagem ocupada não é possível ser calculada.

Pop_back()

```
bool pop_back(){
            if (this->head == nullptr) return false;
            Node *anterior = this->tail->ant;
            this->tail = anterior;
            this->size_--;
            return true;
        }
```

Big Oh da função: *O(1)*

Explicação do código: Apenas mudei o tail para o elemento anterior ao tail atual e diminui o tamanho.

Pop_front()

```
bool pop_front(){
            if (this->head == nullptr) return false;
            Node *anterior = this->head->next;
            this->head = anterior;
            this->size_--;
            return true;
        }
```

Big Oh da função: *O(1)*

Explicação do código: Apenas altero o head para o elemento seguinte ao head atual e diminuo o tamanho

Push_back()

```
void push_back(int v){
            Node *no = new Node;
            no->valor = v;
            no->next = nullptr;
            no->ant = this->tail;
            if (this->tail == nullptr){
                this->head  = no;
            }
            else{
                this->tail->next = no;
            }
            this->tail = no;
            this->size_++;
        }
```

Big Oh da função: *O(1)*

Explicação do código: Apenas criou um novo nó, alterou o tail para o nó recentemente criado e mudou os ponteiros do nó anterior.

Push_front()

```
void push_front(int v){
            Node *no = new Node;
            no->valor = v;
            no->ant = nullptr;
            no->next = this->head;
            if (this->head == nullptr){
                this->tail  = no;
            }
            else{
                this->head->ant = no;
            }
            this->head = no;
            this->size_++;
        }
```

Big Oh da função: *O(1)*

Explicação do código: Apenas criou um novo nó, alterou o head para o nó recentemente criado e mudou os ponteiros do nó seguinte.

Remove(int v)

```
bool remove(int v){
            Node *elemento = this->head;
            while (elemento != nullptr){
                if (elemento->valor == v){
                    Node *anterior = elemento->ant;
                    anterior->next = elemento->next;
                    this->size_--;
                    return true;
                }
                elemento = elemento->next;
            }
            return false;
        }
        }
```

Big Oh da função: *O(n)*

Explicação do código: Ele faz um laço para encontrar o elemento entre todos os nós e assim alterar os ponteiros do elemento anterior e o seguinte para um apontar para o outro.

Remove_at(unsigned int index)

```
bool remove_at(unsigned int index){
            Node *elemento = this->head;
            int pos = 0;
            while (elemento != nullptr){
                if (pos == index){
                    Node *anterior = elemento->ant;
                    anterior->next = elemento->next;
                    this->size_--;
                    return true;
                }
                pos++;
                elemento = elemento->next;
            }
            return false;
        }
```

Big Oh da função: *O(n)*

Explicação do código: É criado uma variável que armazena uma posição relativa ao índice até ela corresponder ao índice dado e os ponteiros serem restabelecidos.

Size()

```
unsigned int size(){
            return this->size_;
        }
```

Big Oh da função: *O(1)*

Explicação do código: Retorna o atributo size.

Sum()

```
int sum(){
            Node *elemento = this->head;
            int ac = 0;
            while (elemento != nullptr){
                ac += elemento->valor;
                elemento = elemento->next;
            }
            return ac;
        }
```

Big Oh da função: *O(n)*

Explicação do código: Faz um laço que incrementa em uma variável acumulador cada elemento.

03.5 Testes lista ligada

Test push_front

E1.txt

Saída do primeiro teste:

[INFO] Elapsed time for 5 pushes front :3800

E2.txt

Saída do segundo teste:

[INFO] Elapsed time for 10 pushes front :8800

E3.txt

Saída do terceiro teste:

[INFO] Elapsed time for 1000 pushes front :814600

E4.txt

Saída do quarto teste:

[INFO] Elapsed time for 2000 pushes front :1333900

E5.txt

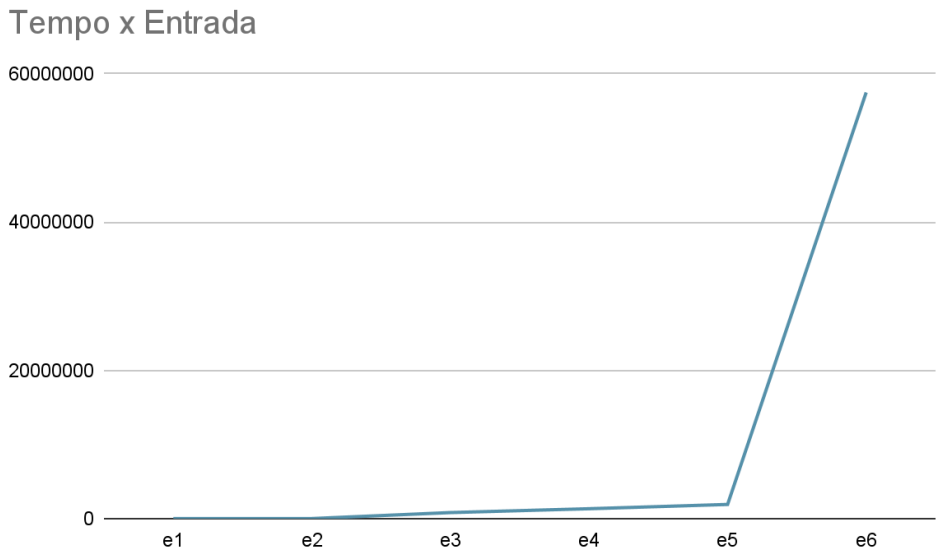
Saída do quinto teste:

[INFO] Elapsed time for 3000 pushes front :1910100

E6.txt

Saída do sexto teste:

[INFO] Elapsed time for 10000 pushes front :57484837



Test removeat

E1.txt

Saída do primeiro teste:

[DEBUG] Tried to remove 10 element(s)

[DEBUG] removed 10 element(s)

[DEBUG] not removed 0 element(s)

[INFO] Elapsed time for 10 remove_at success : 5100

E2.txt

Saída do segundo teste:

[DEBUG] Tried to remove 10 element(s)

[DEBUG] removed 8 element(s)

[DEBUG] not removed 2 element(s)

[INFO] Elapsed time for 8 remove_at success : 7100

E3.txt

Saída do terceiro teste:

[DEBUG] Tried to remove 3 element(s)

[DEBUG] removed 3 element(s)

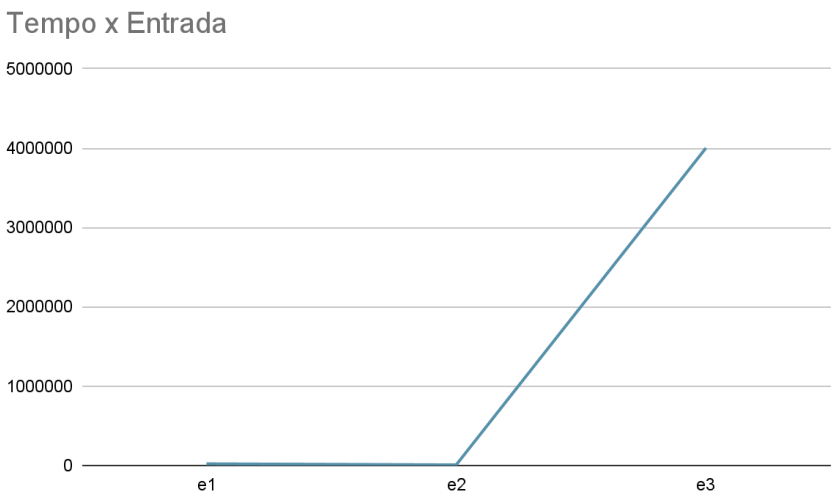
[DEBUG] not removed 0 element(s)

[INFO] Elapsed time for 3 remove_at success : 62100

E4.txt

Saída do quarto teste:

Tempo limite excedido



Test pop_back

E1.txt

Saída do primeiro teste:

Quantidade de números lidos: 4

Tempo de processamento: 0 microseconds(s)

E4.txt

Saída do quarto teste:

Quantidade de números lidos: 1999

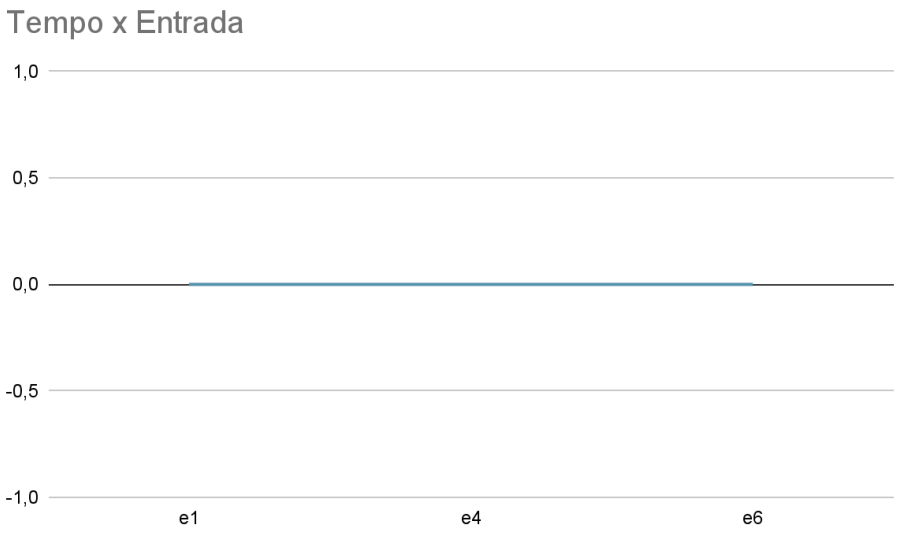
Tempo de processamento: 0 microseconds(s)

E6.txt

Saída do quarto teste:

Quantidade de números lidos: 9999

Tempo de processamento: 0 microseconds(s)

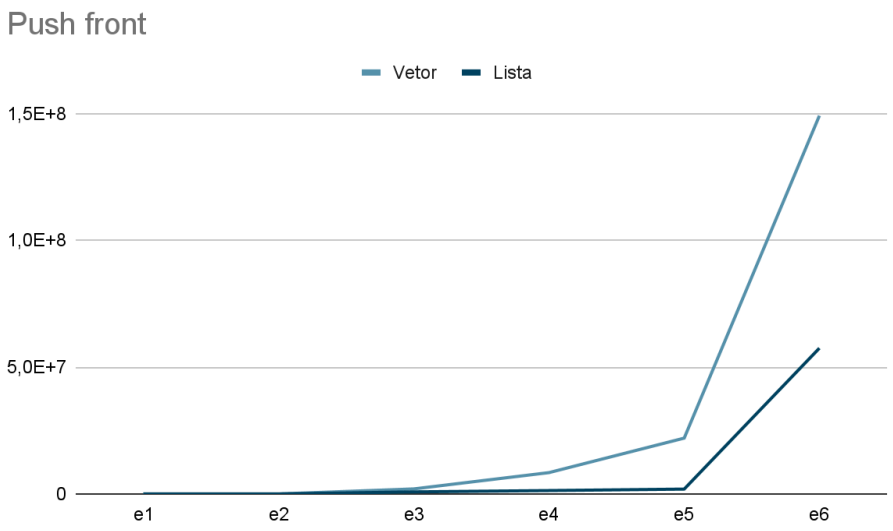


04. Vetor dinâmico x Lista ligada

04.2 Tabelas e gráficos

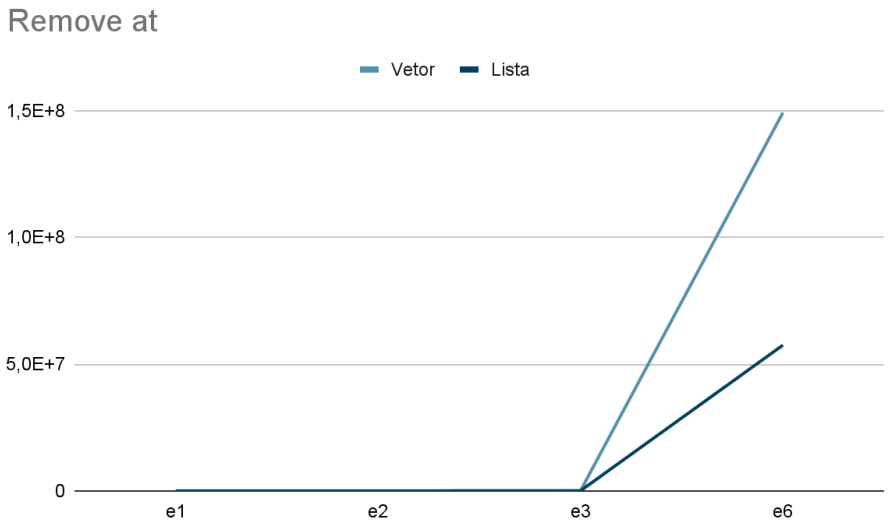
	Vetor dinâmico	Lista ligada
Inserir fim	O(1)	O(1)
Inserir início	O(N)	O(1)
Remover fim	O(1)	O(1)
Remover início	O(1)	O(1)
Remove at	O(N)	O(N)
Acessar elementos aleatórios	O(1)	O(N)

Test push_front

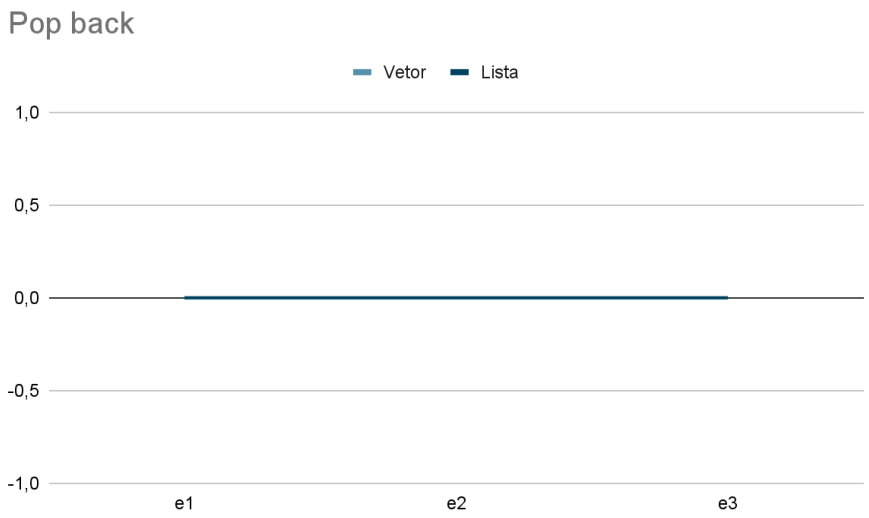


Obs: O dado do vetor dinâmico foi levando em consideração ele começando com 8 e em cada increase capacity ele duplica

Test remove_at



Test pop_back



05. Conclusão

O armazenamento de dados podem ocorrer de diversas formas, mesmo que em suas implementações possuem os mesmo métodos, a forma de como os dados são organizados e acessados muda a eficiência. Concluímos assim, que a estrutura de implementação mais adequada vai depender do propósito do algoritmo.