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

GABRIELLY STÉPHANY SIQUEIRA MAGALHÃES MARTINS

Definição e implementação de um vetor dinâmico

NATAL/RN
2024

GABRIELLY STÉPHANY SIQUEIRA MAGALHÃES MARTINS

Definição e implementação de um vetor dinâmico

Relatório apresentado ao Curso Análise e Desenvolvimento de Sistemas do Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Norte como requisito avaliativo para a disciplina de Algoritmos.

Orientador (a): Jorgiano Vidal

NATAL/RN
2024

1. Introdução

Vetor dinâmico, também conhecido como array ou lista dinâmica, são estruturas de dados que permitem a armazenação de uma coleção de elementos de tamanho variável que pode crescer ou diminuir conforme a necessidade do usuário, muito diferente dos vetores estáticos que como o próprio nome define, são estáticos.

Os vetores, quando definidos na memória, seguem uma organização linear/sequencial. O que permite o acesso aleatório aos elementos, tornando-os mais eficientes em relação a listas ligadas em questão de leitura – ou seja, acesso aos elementos.

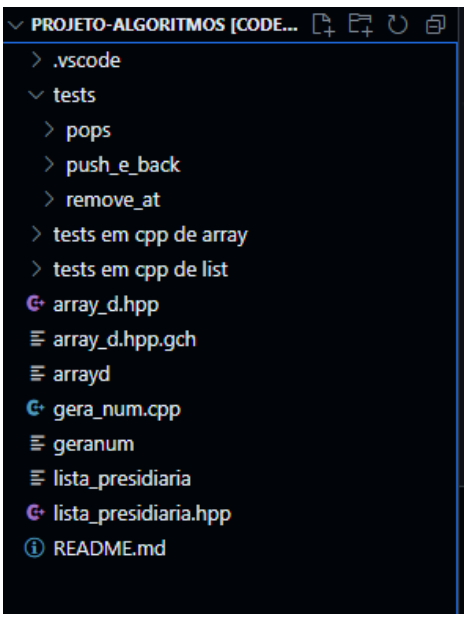
Já listas duplamente encadeadas, também são estruturas de dados que permitem a armazenação de uma coleção de elementos, porém, diferente dos vetores, sua organização na memória é aleatória e é feita através de nós encadeados. Por causa disso, as listas são muito mais eficientes que arrays quando se trata de inserções e remoções.

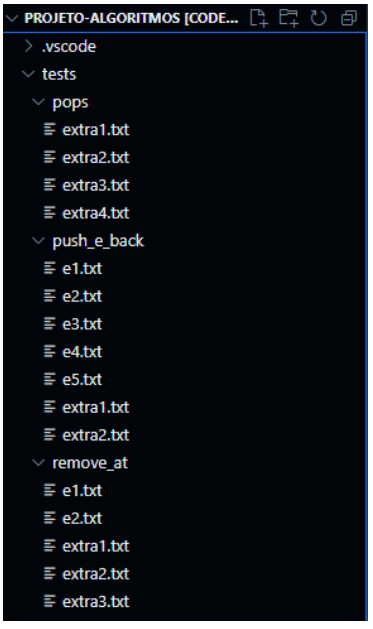
Diante disso, este trabalho consiste na prática de programação, em linguagem c++, relativas a gerenciamento de memória. O objetivo é fazer a implementação dos métodos do vetor dinâmico e da lista duplamente encadeada e os seus desempenhos.

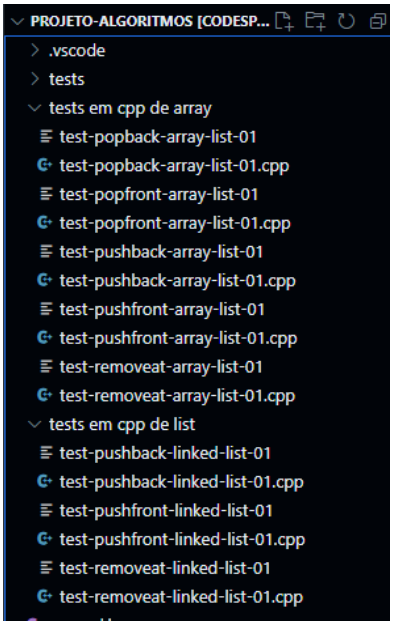
2. Implementação

2.1 Organização dos arquivos fontes

O projeto possui uma hierarquia de pastas, conforme observado nas imagens abaixo. Existem quatro tipos de pastas: a pasta principal do projeto, que contém os arquivos .hpp, o README.md e o arquivo geranum.cpp; e três subpastas: uma para testes em formato .txt, uma para testes em .cpp relacionados a arrays, e outra para testes em .cpp relacionados a listas duplamente encadeadas.







2.2 Arrays com alocação dinâmica

Foi realizada a implementação completa do array e da lista duplamente encadeada. O código foi disponibilizado no GitHub no repositório 'projeto-algoritmos'. E a seguir, serão apresentadas as funções, com uma descrição do que cada uma faz e o desempenho delas.

linked_list() {}

Função: Chamado de construtor, este método define valores iniciais e padrões no nosso código.

Eficiência: O(1) em ambas implementações.

~linked_list() {}

Função: Chamado de destrutor, este método é responsável por limpar e liberar recursos alocados quando um objeto deixa de ser necessário.

Eficiência: O(1) no array dinâmico e O(N) na lista duplamente encadeada.

unsigned int size() {}

Função: Essa função vai retornar a quantidade de elementos armazenados numa lista ou num array.

Eficiência: O(1) em ambas as implementações.

unsigned int capacity() {}

Função: Essa função retorna a capacidade, ou seja, o espaço reservado para armazenar os elementos de uma lista ou de um array. No caso de um array, a capacidade é fixa. Se for necessário aumentar a capacidade, é preciso criar um novo array com uma capacidade maior e copiar os valores do array antigo para o novo. Já em uma lista, a capacidade é dinâmica e, tecnicamente, pode-se considerar que ela é equivalente ao tamanho da lista.

Eficiência: O(1) em ambas as implementações.

double percent_occupied() {}

Função: Essa função retorna um valor entre 0.0 e 1.0 que representa o percentual da memória utilizada. No caso de um array, foi necessário usar a lógica

de regra de três para calcular o percentual, uma vez que o tamanho e a capacidade são diferentes. Em uma lista, como se presume que a capacidade e o tamanho são iguais, a capacidade está sendo utilizada 100% do tempo, refletindo o percentual ocupado.

Eficiência: O(1) em ambas as implementações.

void increase_capacity() {}

Função: Responsável por aumentar a capacidade quando está cheia. Num array ela copia todos os elementos do vetor atual para um novo vetor com capacidade aumentada. Não inserido essa função na lista encadeada.

Eficiência: O(N) no array.

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

Função: Essa função vai inserir um elemento no índice index que for escolhido pelo usuário.

Eficiência: O(N) em ambas as implementações

bool remove_at(unsigned int index) {}

Função: Essa função vai remover o elemento do índice index escolhido pelo usuário.

Eficiência: O(N) em ambas implementações.

int get_at(unsigned int index) {}

Função: Vai retornar o elemento no índice index escolhido pelo usuário, mas irá retornar o valor -1 caso o índice for inválido.

Eficiência: O(1) no array e O(N) na lista duplamente encadeada.

void clear() {}

Função: Esta função vai remover todos os elementos, deixando o vetor no estado inicial.

Eficiência: O(N) em ambas as implementações

void push_back(int value) {}

Função: Esta função é responsável por adicionar um elemento no “final” do array ou da lista.

Eficiência: O(N) no array e O(1) na lista duplamente encadeada.

void push_front(int value) {}

Função: Esta função é responsável por adicionar um elemento no “início” da lista ou do array.

Eficiência: O(N) no array e O(1) na lista duplamente encadeada.

bool pop_back() {}

Função: Esta função é responsável por remover um elemento do “final” da nossa lista ou array.

Eficiência: O(1) em ambas as implementações

bool pop_front() {}

Função: Responsável por remover um elemento do “início” da lista ou array.

Eficiência: O(N) no array e O(1) na lista duplamente encadeada.

int back(){}

Função: Responsável por retornar o valor do elemento que estiver no “final” da lista ou array.

Eficiência: O(1) em ambas as implementações.

int front(){}

Função: Responsável por retornar o valor do elemento que estiver no “início” da lista ou array.

Eficiência: O(1) em ambas as implementações.

bool remove(int value) {}

Função: Esta função vai remover o valor de um índice de um array ou de uma lista, caso ele esteja presente.

Eficiência: O(N) em ambas as implementações.

int find(int value) {}

Função: Esta função é responsável por retornar o índice de um valor de um array ou lista, e caso este valor não esteja presente no vetor ela vai retornar -1.

Eficiência: O(N) em ambas as implementações.

int count(int value) {}

Função: Responsável por retornar quantas vezes um valor vai ocorrer numa lista ou num array.

Eficiência: O(N) em ambas as implementações.

int sum() {}

Função: Retorna a soma dos elementos de uma lista ou array. É basicamente, um contador.

Eficiência: O(N) em ambas as implementações.

2.3 Testes

Para a realização dos testes foi utilizado um notebook com essas seguintes configurações:

- Processador: Intel(R) Celeron(R) CPU N3350 @ 1.10GHz 1.10 GHz
- Memória RAM: 4,00 GB
- Sistema operacional: Windows 10 de 64 bits

Os testes realizados serão apresentados neste documento por meio de uma tabela para cada função, acompanhada de gráficos baseados nos dados da tabela.

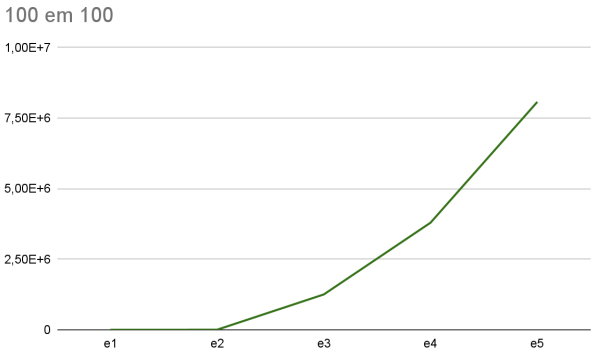
Primeiramente, foram realizados os testes das funções para o array dinâmico. Após concluir os testes com o array dinâmico, foram feitos os testes para a lista

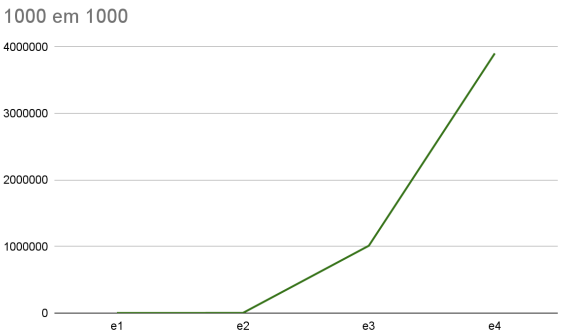
duplamente encadeada. Testes adicionais, entretanto, foram realizados somente com os arrays dinâmicos.

TESTES DO ARRAY DINÂMICO:

Tabela da função push_front:

Entradas	100 em 100	1000 em 1000	Começa em 8 e duplica
e1	961	1042	981
e2	1864	1794	2715
e3	1252312	1011907	990257
e4	3795457	3902556	3858474
e5	8074555	7890362	7812377
extra1	13709749	14905428	14013523
extra2	81279507	81440326	81810282





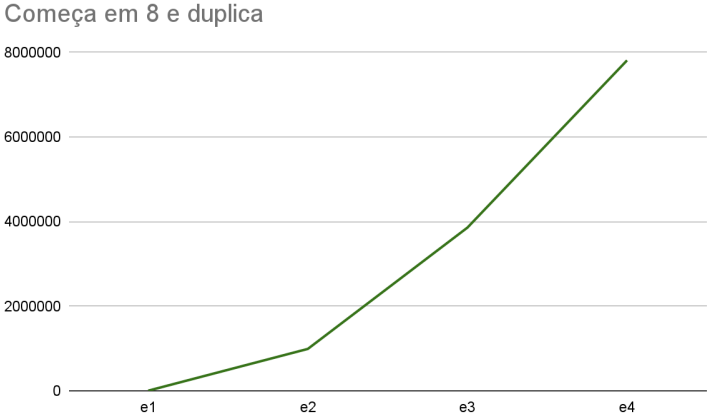
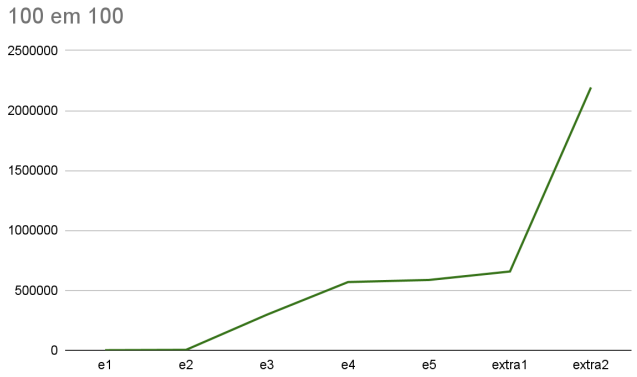
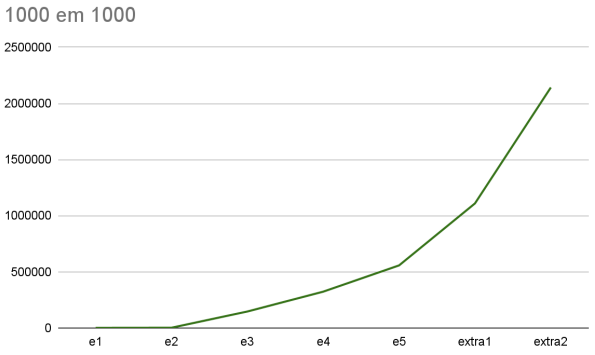


Tabela da função push_back:

Entradas	100 em 100	1000 em 1000	Começa em 8 e duplica
e1	842	1302	882
e2	3767	2495	4969
e3	297444	146824	145721
e4	569182	323523	297755
e5	586704	557509	482819
extra1	657354	1109187	717206
extra2	2192684	2141710	2308260





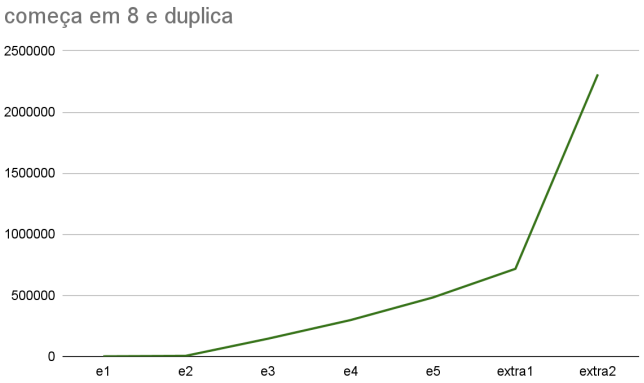
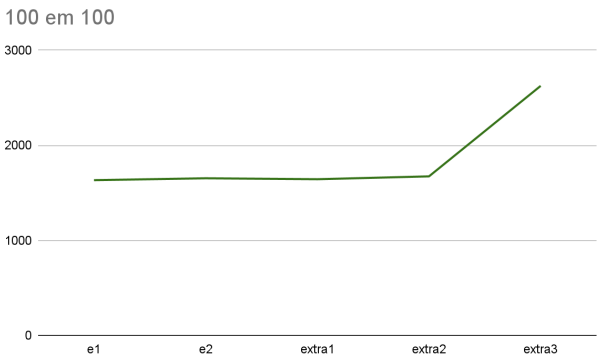
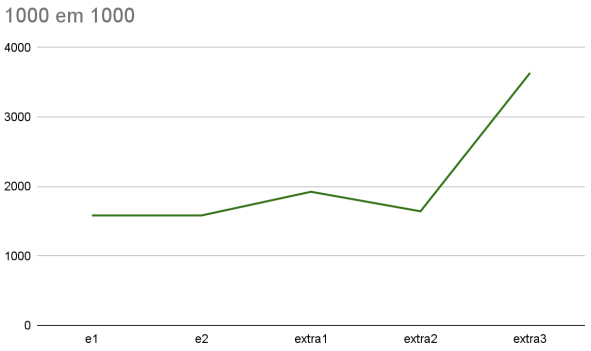


Tabela da função remote_at:

Entradas	100 em 100	1000 em 1000	Começa em 8 e duplica
e1	1633	1583	1623
e2	1653	1583	1574
extra1	1643	1924	1563
extra2	1673	1643	1644
extra3	2625	3636	1493





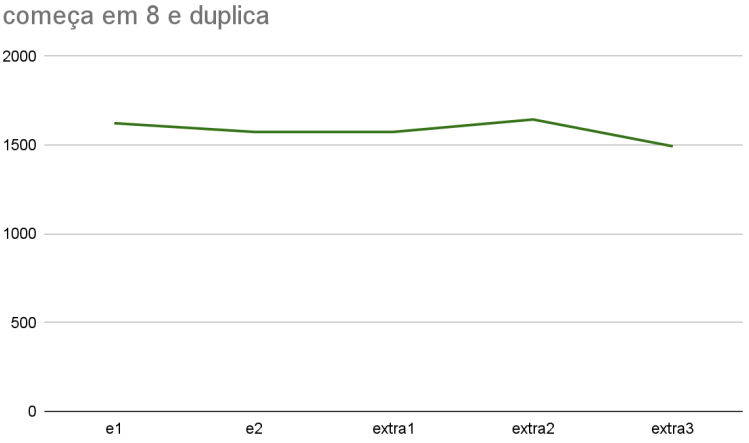
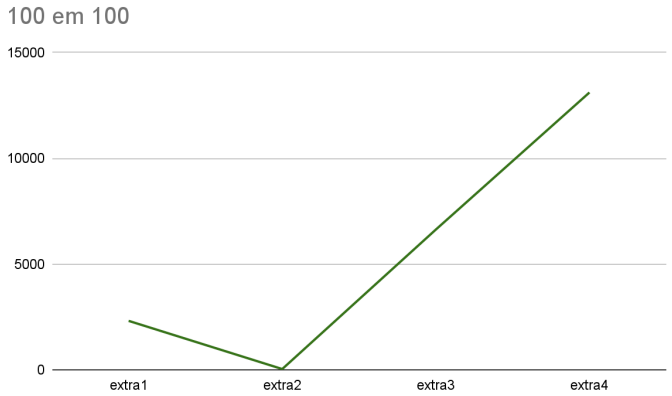
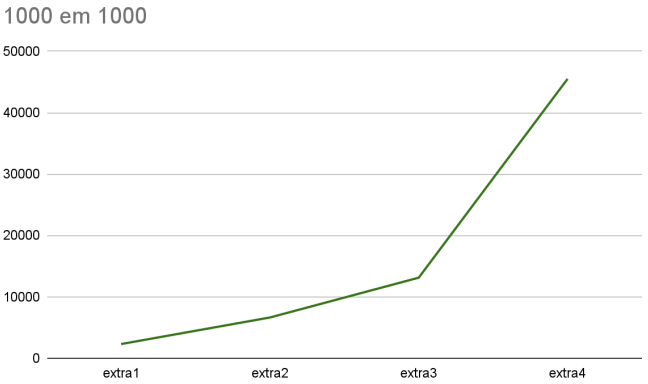


Tabela da função pop_front:

Entradas	100 em 100	1000 em 1000	Começa em 8 e duplica
extra1	2314	2314	2334
extra2	6632	6640	6702
extra3	13114	13124	13125
extra4	26049	45494	26079





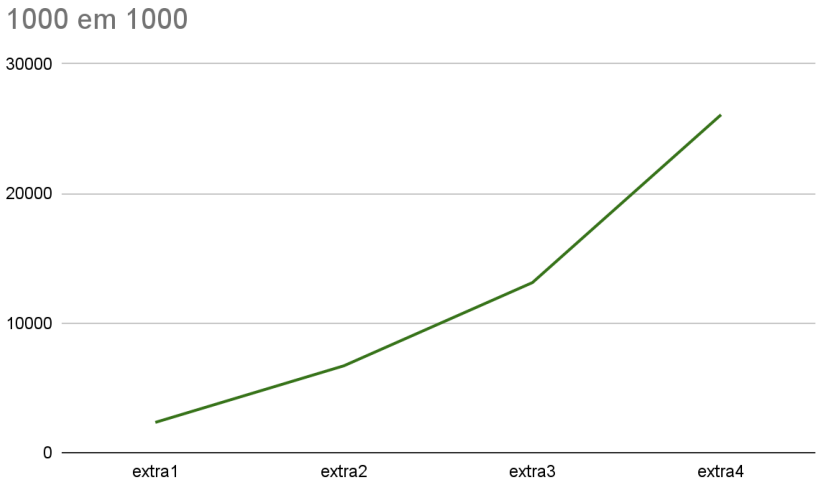
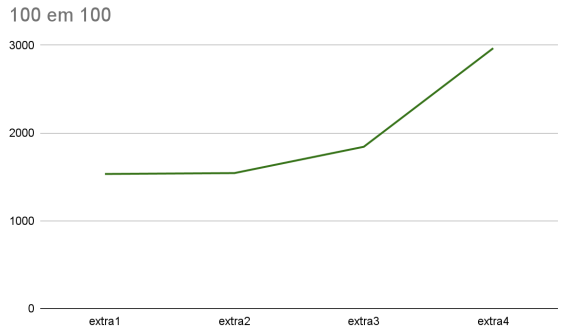
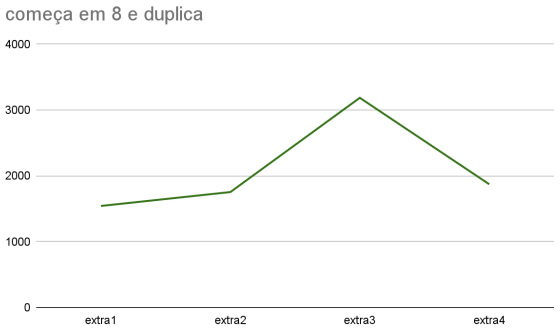
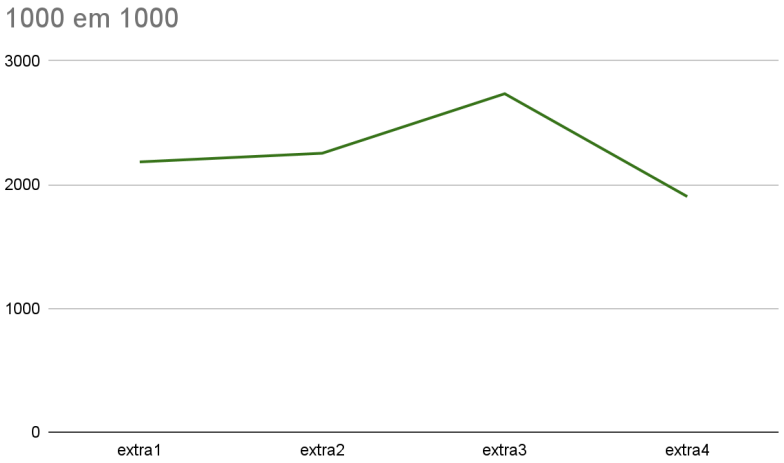


Tabela do pop_back:

Entradas	100 em 100	1000 em 1000	Começa em 8 e duplica
extra1	1533	2184	1543
extra2	1543	2254	1754
extra3	1843	2735	3186
extra4	2966	1904	1873







TESTES DA LISTA DUPLAMENTE LIGADA

Tabela da função push front:

Entradas	Tamanhos da entradas	Resultados
e1	6	1132
e2	11	1813
e3	1001	167292
e4	2001	438546
e5	3001	631416

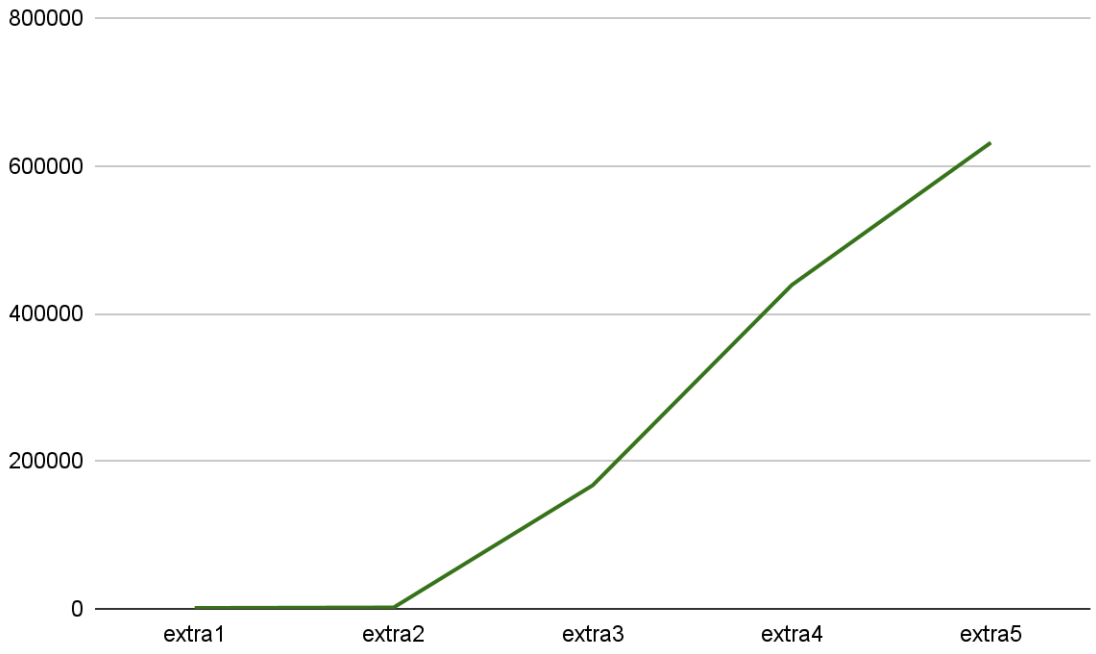


Tabela do push_back:

Entradas	Tamanhos da entradas	Resultados
e1	6	1152
e2	11	1853
e3	1001	168475
e4	2001	447334
e5	3001	524458

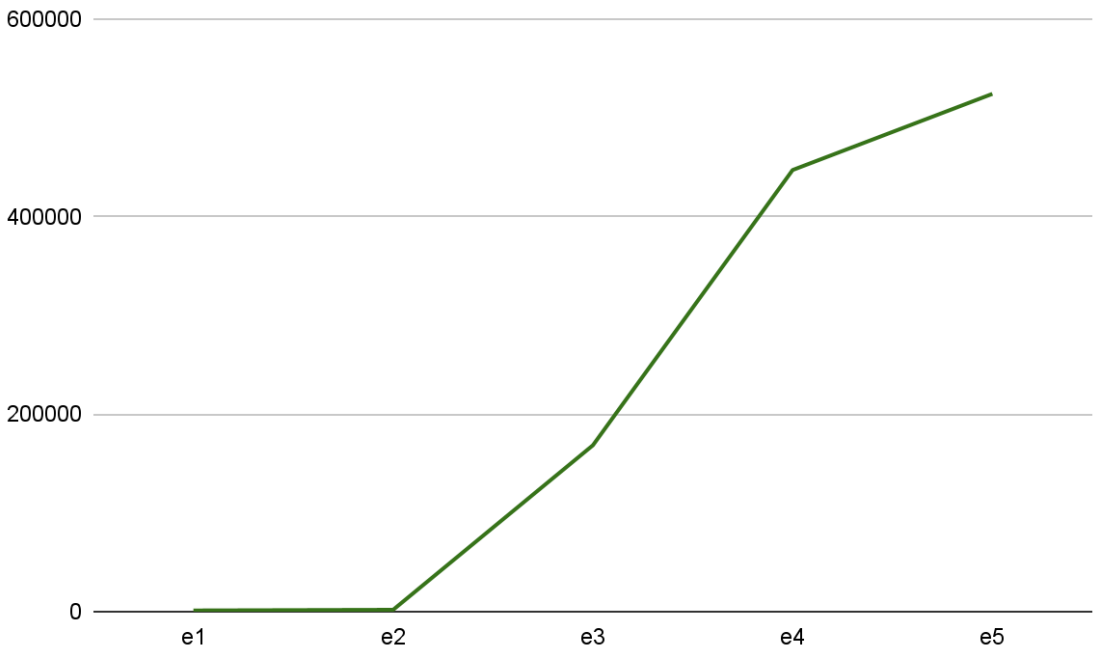
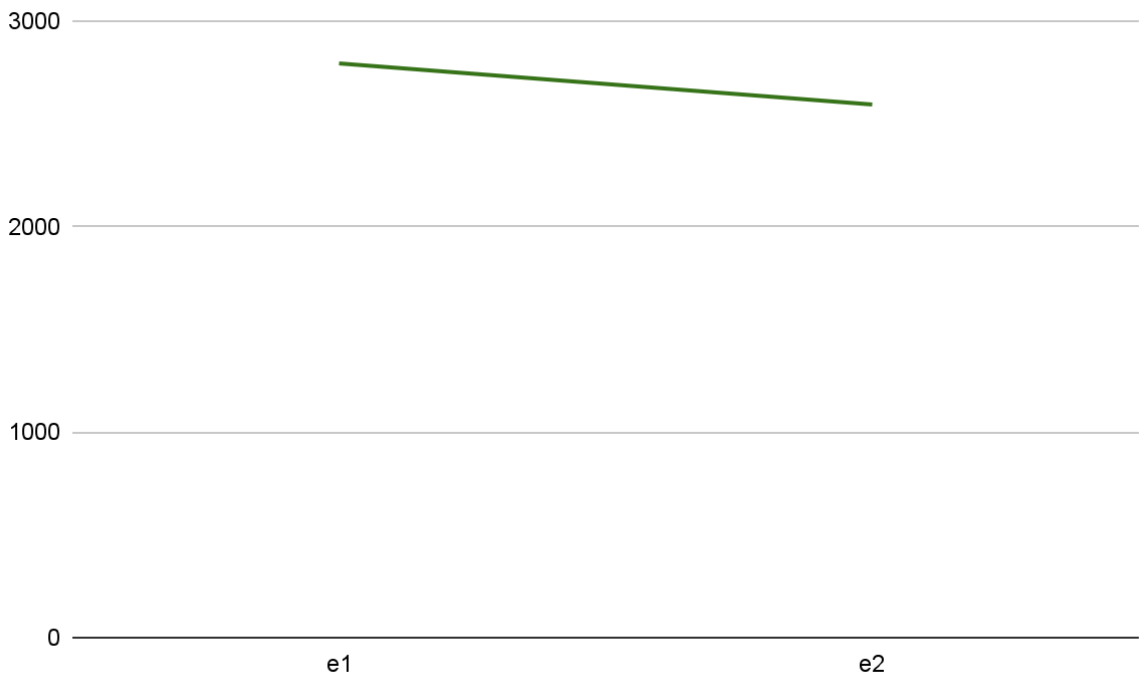


Tabela do remove_at:

Entradas	Tamanhos da entradas	Resultados
e1	27	2795
e2	27	



3. Conclusão

Neste trabalho, foram apresentadas duas implementações de estruturas de dados: array dinâmico com alocação dinâmica e lista duplamente encadeada. A análise revela diferenças significativas entre elas, tanto na forma como funcionam quanto na sua eficiência, com base no conhecimento prévio e nos resultados dos testes realizados.

O array dinâmico se destaca pela eficiência no acesso e leitura dos elementos, proporcionando um acesso direto e rápido através de índices. Em contraste, a lista duplamente encadeada oferece vantagens na inserção e remoção de elementos, mostrando-se mais eficiente nessas operações.

No entanto, em termos de implementação, a lista duplamente encadeada resulta em um código mais extenso devido ao seu acesso sequencial e ao uso adicional de memória para armazenar ponteiros. Esses fatores contribuem para uma maior complexidade na manipulação da lista.