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

Mateus Vitor Cavalcanti Rodrigues

DEFINIÇÃO E IMPLEMENTAÇÃO DE UM VETOR DINAMICO

NATAL

2024

Mateus Vitor Cavalcanti Rodrigues

DEFINIÇÃO E IMPLEMENTAÇÃO DE UM VETOR DINAMICO

Trabalho realizado no curso de Tecnologia em Análise e Desenvolvimento de Sistemas (TADS) do Instituto Federal de Educação Ciência e Tecnologia do Rio Grande do Norte (IFRN), como um dos requisitos avaliativos para a disciplina de Algoritmos.
Docente: Jorgiano Vidal

NATAL

2024

RESUMO

Este documento tem como finalidade explicar o funcionamento de duas implementações de bibliotecas de classes para manipulação de um vetor dinâmico de números inteiros. As duas classes que foram desenvolvidas são respectivamente: alocação dinâmica de arrays (Array List) e Lista duplamente ligada (Linked list).

Palavras-chave: Lista duplamente ligada; Vetor dinâmico; Alocação dinâmica.

1 - INTRODUÇÃO

Este documento tem como finalidade explicar o funcionamento de duas implementações de bibliotecas de classes para manipulação de um vetor dinâmico de números inteiros. As duas classes que foram desenvolvidas são respectivamente: alocação dinâmica de arrays (Array List) e Lista duplamente ligada (Linked list).

Um vetor dinâmico é uma estrutura de dados que permite armazenar elementos de forma sequencial, com a capacidade de redimensionar-se automaticamente conforme novos elementos são adicionados ou removidos. Ao contrário de arrays estáticos, que têm tamanho fixo, vetores dinâmicos ajustam seu tamanho dinamicamente para acomodar o crescimento ou redução dos dados armazenados. Isso é feito geralmente duplicando o tamanho do array subjacente quando a capacidade é excedida e reduzindo-o quando há uma quantidade significativa de espaço não utilizado.

Alocação dinâmica de arrays (Array List) refere-se ao processo de alocar memória para um array em tempo de execução, permitindo que o tamanho do array seja definido e ajustado conforme necessário.

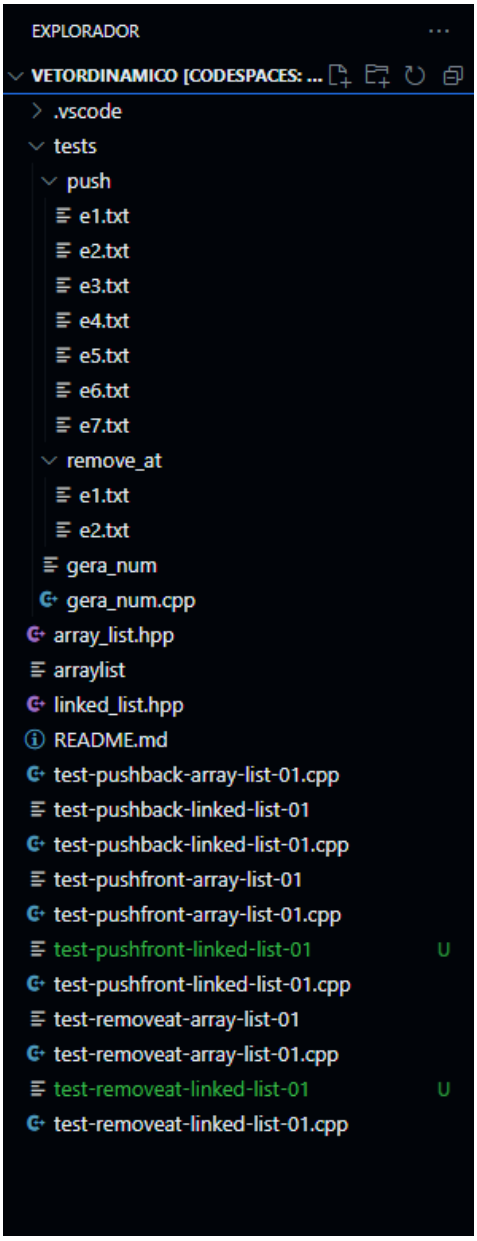
Uma lista duplamente ligada (Linked List) é uma lista linear de elementos em que cada elemento (ou nó) contém referências (ponteiros) para o próximo e o anterior elementos da lista. Isso permite navegação em ambas as direções (para frente e para trás) e facilita operações de inserção e remoção em qualquer posição da lista.

Essa combinação de características torna os vetores dinâmicos e as listas duplamente ligadas ferramentas extremamente versáteis e poderosas na programação. Eles oferecem uma solução eficiente para o gerenciamento de coleções de dados que mudam de tamanho dinamicamente, adaptando-se às necessidades específicas do aplicativo em tempo real.

2 – IMPLEMENTAÇÃO

2.1 - ORGANIZAÇÃO DOS ARQUIVOS FONTES

Segue foto de organização de arquivos presente no github. Os testes do autor desse documento foram todos realizados através do Codespace presente no github, pois, o mesmo não obteve sucesso ao tentar instalar o WSL em sua máquina (Erro presente: VS Code Server for WSL closed unexpectedly. Check WSL terminal for more details. Terminal log: Habilite o componente opcional “Plataforma de maquina virtual” e verifique se a virtualização está habilitada na BIOS (Foi realizada ambas as checagem e ainda sim não funcionou).).



2.2 – CRIAÇÃO DAS FUNÇÕES: ARRAY LIST E LINKED LIST

ARRAYLIST:

Legenda: Array List estará em VERMELHO e Linked List estará em ROXO

Array List - Increase_capacity(){}

Eficiência: Complexidade O(N)

Função: Aumenta a capacidade do array

```
void increase_capacity() {
    capacity_ += 100;
    int* new_data = new int[capacity_];
    for (unsigned int i = 0; i < size_; ++i) {
        new_data[i] = data[i];
    }
    delete[] data;
    data = new_data;
}
```

Array List – array_list(){}

Eficiência: Complexidade O(1)

Função: Construtor

```
array_list(){
    data = new int[100];
    size_ = 0;
    capacity_ = 100;
}
```

Linked List – linked_list(){}

Eficiência: Complexidade O(1)

Função: Construtor

```
linked_list() {
    this->head = nullptr;
    this->tail = nullptr;
    this->size_ = 0;
}
```

Array List – ~array_list(){}

Eficiência: Complexidade O(1)

Função: Destrutor

```
~array_list() {
    delete[] data;
}
```

Linked List – ~linked_list(){}

Eficiência: Complexidade O(1)

Função: Destrutor

```
~linked_list() {
    this->clear();
}
```

Array List – size(){}

Eficiência: Complexidade O(1)

Função: Retorna o tamanho da lista

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

Linked List – size(){}

Eficiência: Complexidade O(1)

Função: Retorna o tamanho da lista

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

Array List – capacity(){}

Eficiência: Complexidade O(1)

Função: Retorna a capacidade da lista

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

Linked List – capacity(){}

Eficiência: Complexidade O(1)

Função: Retorna a capacidade da lista

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

Array List – percent_occupied(){}

Eficiência: Complexidade O(1)

Função: Retorna o percentual ocupado

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

Linked List – percent_occupied(){}

Eficiência: Complexidade O(1)

Função: Retorna o percentual ocupado

```
double percent_occupied() {
    return this->size_ > 0 ? 1.0 : 0.0;
}
```

Array List – insert_at(){}

Eficiência: Complexidade O(N)

Função: Insere um elemento em uma posição especifica

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

Linked List – insert_at(){}

Eficiência: Complexidade O(N)

Função: Insere um elemento em uma posição especifica

```
bool insert_at(unsigned int index, int value) {
    if (index > this->size_) return false;
    if (index == 0) {
        this->push_front(value);
        return true;
    }
    if (index == this->size_) {
        this->push_back(value);
        return true;
    }
    int_node* new_node = new int_node{value, nullptr, nullptr};
    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;
}
```

Array List – remove_at(){}

Eficiência: Complexidade O(N)

Função: Remove um elemento de uma posição especifica

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

Linked List – remove_at(){}

Eficiência: Complexidade O(N)

Função: Remove um elemento de uma posição especifica

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

Array List – get_at(){}

Eficiência: Complexidade O(N)

Função: Retorna um elemento de uma posição especifica

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

Linked List – get_at(){}

Eficiência: Complexidade O(N)

Função: Retorna um elemento de uma posição especifica

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

Array List – clear(){}

Eficiência: Complexidade O(1)

Função: Limpa a lista

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

Linked List – clear(){}

Eficiência: Complexidade O(N)

Função: Limpa a lista

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

Array List – push_front(){}

Eficiência: Complexidade O(N)

Função: Insere um elemento no inicio da lista

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

Linked List – push_front(){}

Eficiência: Complexidade O(1)

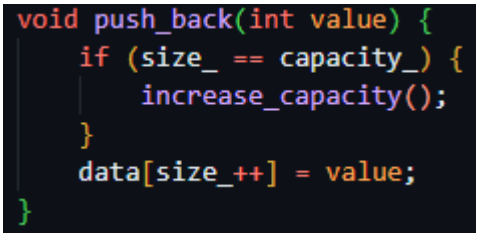
Função: Insere um elemento no inicio da lista

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

Array List – push_back(){}

Eficiência: Complexidade O(1)

Função: Insere um elemento no final da lista

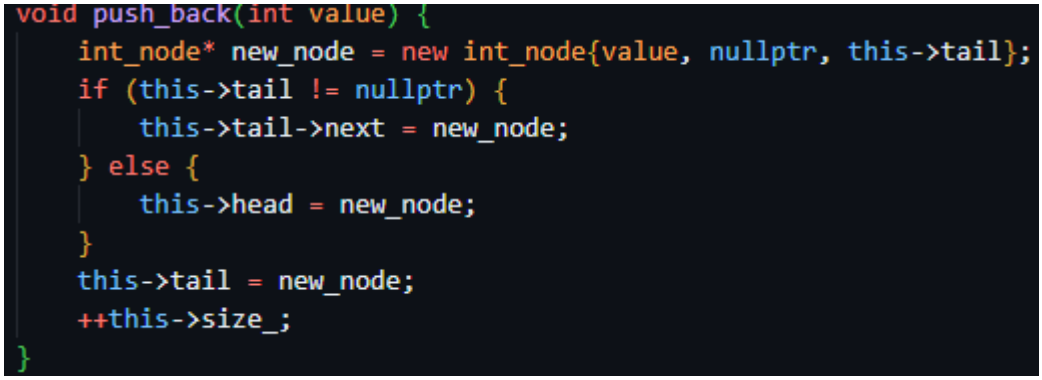


```
void push_back(int value) {
    if (size_ == capacity_) {
        increase_capacity();
    }
    data[size_++] = value;
}
```

Linked List – push_back(){}

Eficiência: Complexidade O(1)

Função: Insere um elemento no final da lista

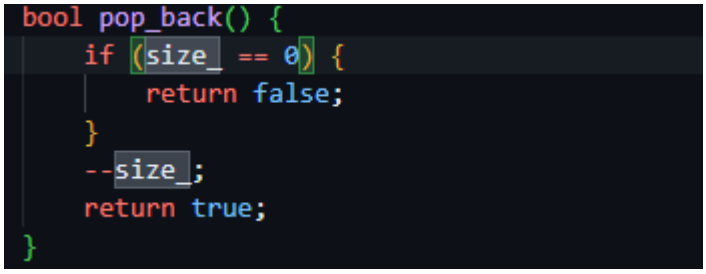


```
void push_back(int value) {
    int_node* new_node = new int_node{value, nullptr, this->tail};
    if (this->tail != nullptr) {
        this->tail->next = new_node;
    } else {
        this->head = new_node;
    }
    this->tail = new_node;
    ++this->size_;
}
```

Array List – pop_back(){}

Eficiência: Complexidade O(1)

Função: Remove o elemento do final da lista



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

Linked List – pop_back(){}

Eficiência: Complexidade O(1)

Função: Remove o elemento do final da lista

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

Array List – pop_front(){}

Eficiência: Complexidade O(N)

Função: Remove o elemento do inicio da lista

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

Linked List – pop_front(){}

Eficiência: Complexidade O(1)

Função: Remove o elemento do inicio da lista

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

Array List – back(){}

Eficiência: Complexidade O(1)

Função: Retorna o elemento do final da lista

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

Linked List – back(){}

Eficiência: Complexidade O(1)

Função: Remove o elemento do final da lista

```
int back() {
    if (this->tail != nullptr) {
        return this->tail->value;
    } else {
        return -1;
    }
}
```

Array List – front(){}

Eficiência: Complexidade O(1)

Função: Retorna o elemento do inicio da lista

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

Linked List – front(){}

Eficiência: Complexidade O(1)

Função: Remove o elemento do inicio da lista

```
int front() {
    if (this->head != nullptr) {
        return this->head->value;
    } else {
        return -1;
    }
}
```

Array List – remove(){}

Eficiência: Complexidade O(N)

Função: Remove o primeiro elemento com o valor especificado

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

Linked List – front(){}

Eficiência: Complexidade O(N)

Função: Remove o primeiro elemento com o valor especificado

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

Array List – find(){}

Eficiência: Complexidade O(N)

Função: Retorna o índice do primeiro elemento com o valor especificado

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

Linked List – find(){}

Eficiência: Complexidade O(N)

Função: Retorna o índice do primeiro elemento com o valor especificado

```
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;
}
```

Array List – count(){}

Eficiência: Complexidade O(N)

Função: Retorna a quantidade de elementos com o valor especificado

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

Linked List – count(){}

Eficiência: Complexidade O(N)

Função: Retorna a quantidade de elementos com o valor especificado

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

Array List – sum(){}

Eficiência: Complexidade O(N)

Função: Retorna a soma de todos os elementos da lista

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

Linked List – sum(){}

Eficiência: Complexidade O(N)

Função: Retorna a soma de todos os elementos da lista

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

2.3 – CASOS DE TESTES

Os casos de teste foram realizados pelo codespace do github no qual todos foram feitos em um computador com as seguintes configurações:

Nome do dispositivo	DESKTOP-M1MP9TC
Processador	AMD Ryzen 5 1600 Six-Core Processor 3.20 GHz
RAM instalada	16,0 GB (utilizável: 7,93 GB)
ID do dispositivo	FE216B2C-E6DC-4212-A1FE-4BDED01FE69E
ID do Produto	00330-80000-00000-AA131
Tipo de sistema	Sistema operacional de 64 bits, processador baseado em x64
Caneta e toque	Nenhuma entrada à caneta ou por toque disponível para este vídeo

Gráfico e tabela 01

Array List – push_front(){}

Caso de teste	100 em 100	1000 em 1000	começa em 8 e duplica
e1.txt	982	1002	1002
e2.txt	1714	1764	2765
e3.txt	1189439	1102948	1196813
e4.txt	4449461	4431685	4441547
e5.txt	9515763	9699937	9401771
e6.txt	933607336	899882248	888778392
e7.txt	10246523734	10116922954	10149187122

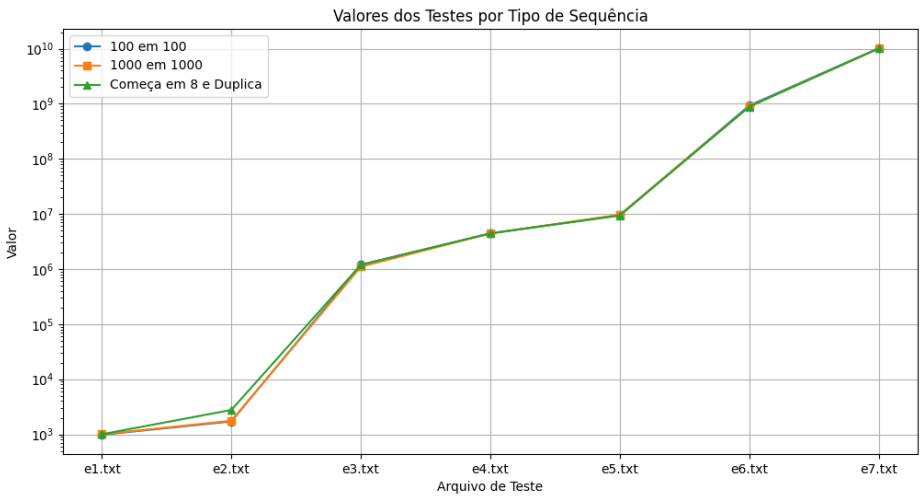


Gráfico e tabela 02

Array List – remove_at(){}

Caso de teste	100 em 100	1000 em 1000	começa em 8 e duplica
e1.txt	1653	1543	2455
e2.txt	1653	1714	2636

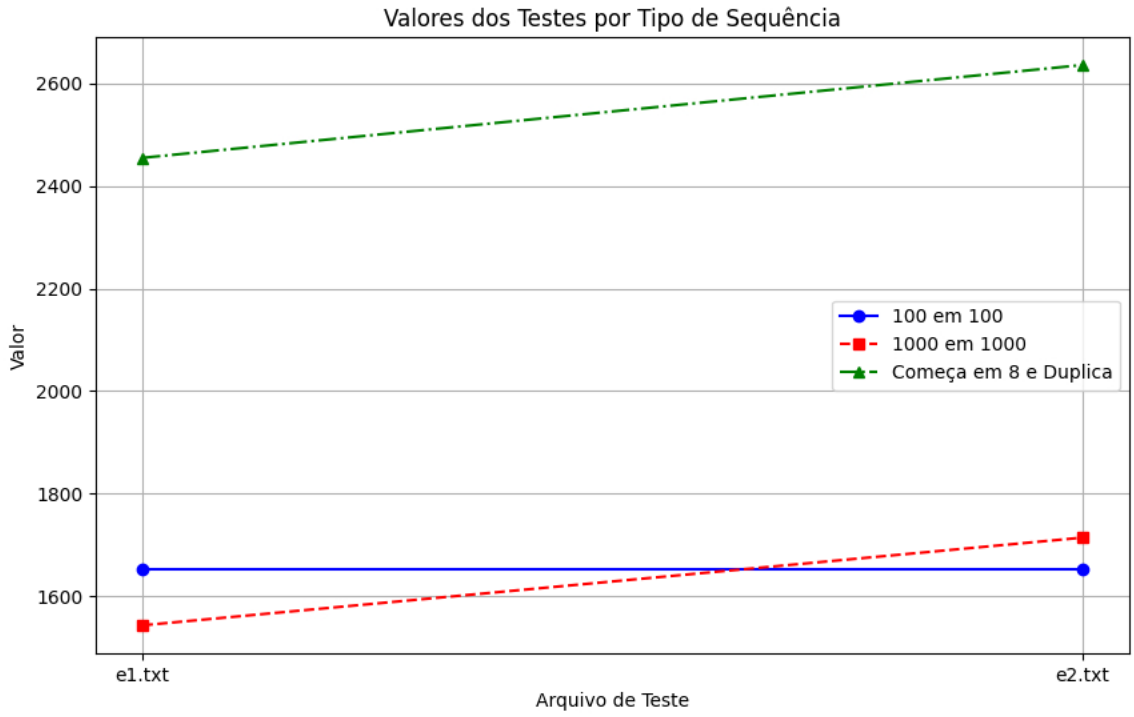


Gráfico e tabela 03

Array List – push_back(){}

Caso de teste	100 em 100	1000 em 1000	começa em 8 e duplica
e1.txt	1283	1252	851
e2.txt	1482	1433	3215
e3.txt	151302	138728	149640
e4.txt	419523	389947	329064
e5.txt	544957	549736	440903
e6.txt	14290462	8335701	8399672
e7.txt	152546923	33933339	25225572

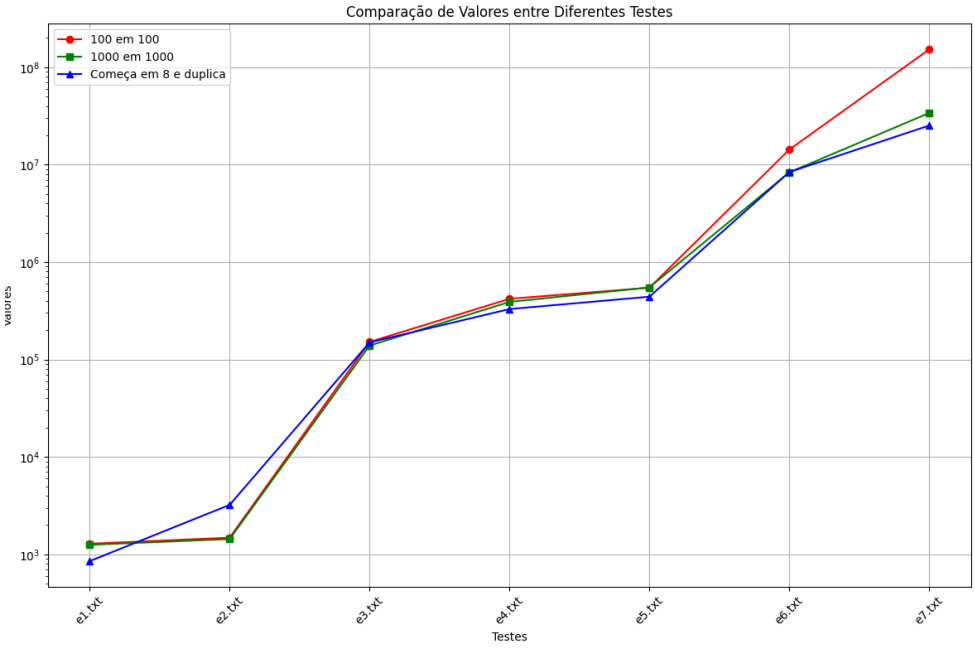


Gráfico e tabela 04

Linked List – push_front(){}

Caso de teste	100 em 100
e1.txt	1143
e2.txt	1763
e3.txt	166531
e4.txt	342439
e5.txt	515341
e6.txt	6361979
e7.txt	21425032

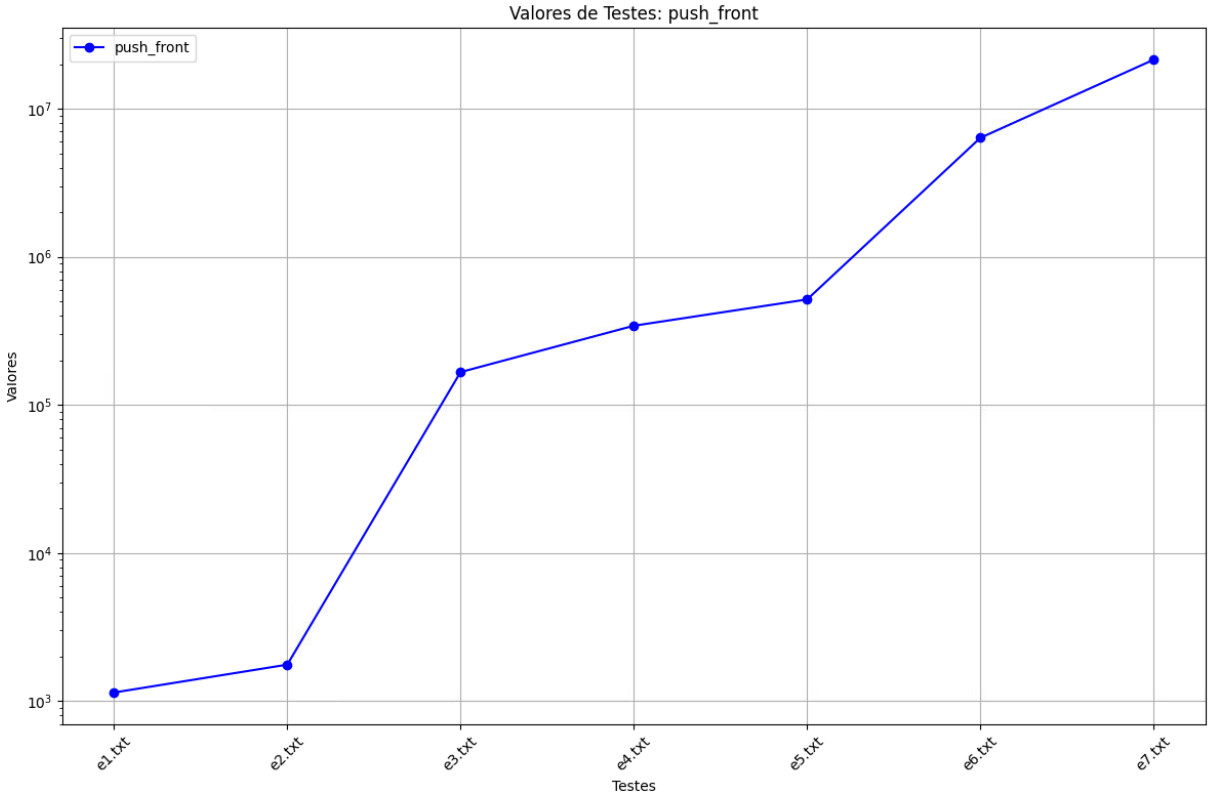


Gráfico e tabela 05

Linked List – remove_at(){}

Remove at

Caso de teste	100 em 100
e1.txt	1623
e2.txt	1654

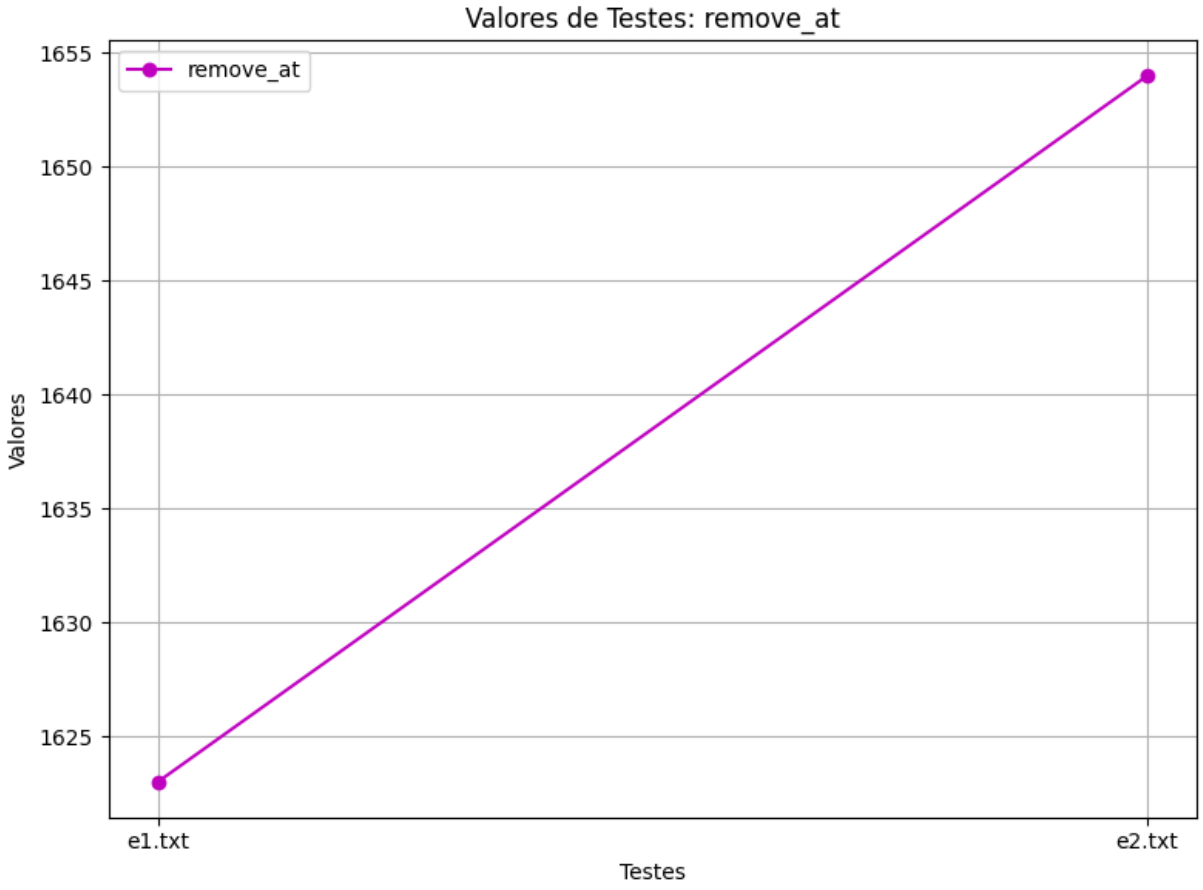
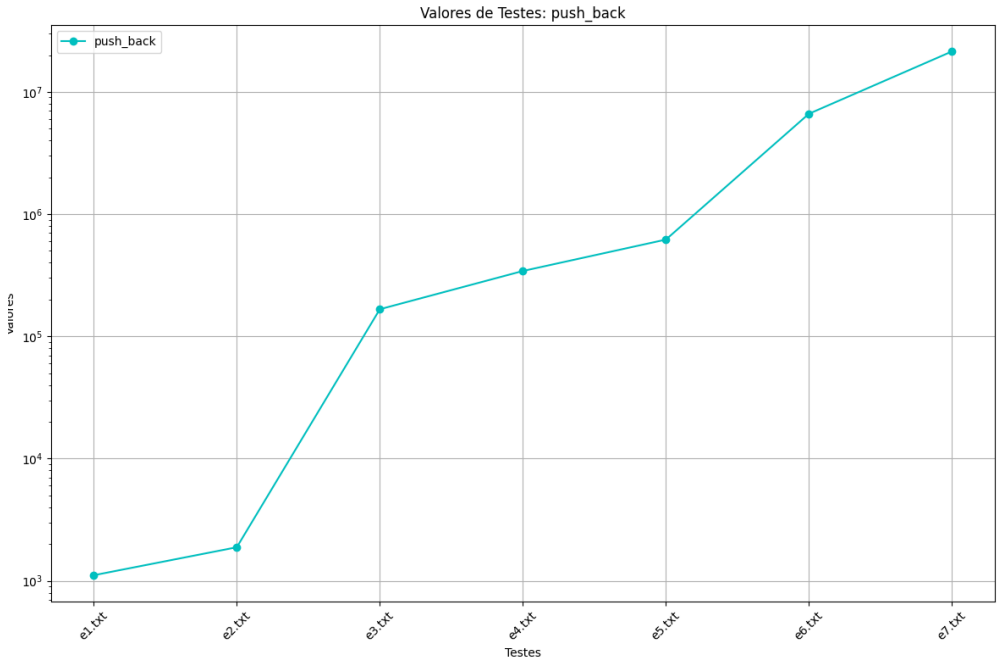


Gráfico e tabela 06

Linked List – (){}

Caso de teste	100 em 100
e1.txt	1112
e2.txt	1883
e3.txt	166731
e4.txt	342619
e5.txt	618945
e6.txt	6621806
e7.txt	21415975



3 – CONCLUSÃO

Em resumo, tanto a implementação de alocação dinâmica de arrays (Array List) quanto a lista duplamente ligada (Linked List) oferecem soluções eficientes e flexíveis para o gerenciamento de coleções de dados dinâmicos. A alocação dinâmica de arrays se destaca pela simplicidade e eficiência em operações de acesso e inserção sequencial. Por outro lado, as listas duplamente ligadas proporcionam uma maior flexibilidade em operações de inserção e remoção, permitindo que essas operações sejam realizadas em qualquer posição da lista com eficiência.

Os testes realizados ao longo deste documento demonstraram que cada uma dessas implementações possui suas próprias vantagens e desvantagens, dependendo do contexto e das necessidades específicas da aplicação. Em cenários onde o acesso aleatório rápido e a inserção sequencial são cruciais, a alocação dinâmica de arrays se mostrou mais vantajosa. Já em aplicações onde a inserção e remoção frequente de elementos em posições arbitrárias são necessárias, a lista duplamente ligada apresentou melhor desempenho.

Portanto, a escolha entre uma Array List e uma Linked List deve ser feita com base nas características específicas da aplicação em questão. Entender as necessidades de acesso, inserção e remoção de dados é essencial para selecionar a estrutura de dados mais adequada, garantindo assim a eficiência e a performance do sistema.

Em conclusão, ambas as estruturas de dados são ferramentas valiosas no arsenal de um desenvolvedor, oferecendo soluções robustas para o desafio de gerenciar coleções de dados dinâmicos. Ao compreender as diferenças e aplicações ideais de cada uma, podemos maximizar o desempenho e a eficiência das nossas aplicações, adaptando-as às necessidades em constante evolução do mundo real.