INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DO RIO GRANDE DO NORTE
CAMPUS NATAL CENTRAL
DIRETORIA ACADÊMICA DE GESTÃO E TECNOLOGIA DA INFORMAÇÃO
TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

FABIANA CAMPOS SERRA DOS SANTOS

VETORES DINÂMICOS

NATAL - RN

2023

FABIANA CAMPOS SERRA DOS SANTOS

VETORES DINÂMICOS

Trabalho do curso de Tecnologia em Análise e Desenvolvimento de Sistemas apresentado como requisito parcial para aprovação na disciplina de Algoritmos.

Prof. Dr. Jorgiano Marcio Bruno Vidal
Professor

NATAL - RN

2023

SUMÁRIO

1. INTRODUÇÃO

Vetores dinâmicos de números inteiros são arrays de números inteiros que podem alterar seu tamanho durante a execução do programa. Eles são implementados usando alocação dinâmica de memória, o que significa que o espaço para armazenar os elementos do vetor é alocado dinamicamente, conforme necessário. Isso permite que os vetores dinâmicos cresçam e diminuam de tamanho de acordo com as necessidades do programa.

Listas duplamente ligadas são estruturas de dados que armazenam uma sequência de elementos ordenados, de forma que cada elemento contém um ponteiro para o elemento anterior e para o próximo elemento na sequência. Isso permite que os elementos da lista sejam acessados e percorridos em ambas as direções, independentemente da ordem em que foram inseridos na lista.

O uso de vetores dinâmicos de números inteiros se mostram mais eficientes que listas duplamente ligadas para acessar elementos aleatórios da sequência. Isso ocorre porque os elementos de um vetor dinâmico são armazenados em um array contíguo, o que permite que eles sejam acessados diretamente usando seu índice. Já as listas duplamente ligadas, por outro lado, armazenam seus elementos em uma estrutura de dados não contígua, o que significa que acessar um elemento aleatório da lista requer percorrer a lista a partir do início ou do fim até que o elemento desejado seja encontrado.

No entanto, as listas duplamente ligadas são mais eficientes que os vetores dinâmicos para inserir ou remover elementos no meio da sequência, já que os vetores dinâmicos precisam ser realocados quando seu tamanho muda, o que pode ser uma operação lenta. As listas duplamente ligadas, por outro lado, podem inserir ou remover elementos no meio da sequência sem precisar realocar a lista.

Este trabalho tem como objetivo, através da implementação de biblioteca de classes para manipulação de um vetor dinâmico de números inteiros, a prática de programação, em Linguagem C++, relativas a gerenciamento de memória. São duas classes a serem desenvolvidas: uma implementada com alocação dinâmica de arrays, e outra implementada com lista duplamente ligada.

2. REVISÃO DE LITERATURA

2.1 VETORES DINÂMICOS E ARRAYS

Vetores são classes de sequência que representam arrays que podem mudar de tamanho. Assim como arrays, os vetores usam locais de armazenamento próximos para seus elementos, o que significa que seus elementos também podem ser acessados usando deslocamentos em ponteiros regulares para seus elementos, e com a mesma eficiência que em arrays. Mas, ao contrário dos arrays, seu tamanho pode mudar dinamicamente, com seu armazenamento sendo gerenciado automaticamente pela classe.

Internamente, os vetores usam um array alocado dinamicamente para armazenar seus elementos. Estes arrays podem precisar ser realocados para crescer de tamanho quando novos elementos são inseridos, o que implica alocar um novo array e mover todos os elementos para ele. Esta é uma tarefa relativamente cara em termos de tempo de processamento e, portanto, os vetores não são realocados cada vez que um elemento é adicionado.

Em vez disso, os vetores podem alocar algum armazenamento extra para acomodar um possível crescimento, podendo ter uma capacidade real maior do que o armazenamento estritamente necessário para conter seus elementos (ou seja, seu tamanho). Portanto, em comparação com arrays, vetores consomem mais memória em troca da capacidade de gerenciar o armazenamento e crescer dinamicamente de forma eficiente.

Em comparação com as outras classes de alocação dinâmica, os vetores são muito eficientes para acessar seus elementos (assim como arrays) e relativamente eficientes para adicionar ou remover elementos de seu final. Para operações que envolvem a inserção ou remoção de elementos em posições diferentes do final, eles têm um desempenho pior do que os outros e possuem iteradores menos consistentes do que as listas.

2.2 LISTA DUPLAMENTE LIGADA

Nas listas duplamente ligadas, ou encadeadas, é formada por nós similares ao da lista encadeada simples, porém com um atributo adicional: um apontador para o nó anterior. Dessa forma, é possível percorrer esse tipo de lista em ambas as direções. Cada elemento mantém informações sobre como localizar o próximo e o elemento anterior, permitindo operações de inserção e exclusão de tempo constante antes ou depois de um elemento específico (mesmo

de intervalos inteiros), mas sem acesso aleatório direto a algum elemento no meio da lista, por exemplo.

2.2.1 Comparação

A escolha entre uma classe e outra depende inteiramente do seu objetivo. As listas duplamente ligadas são boas para classificação, por exemplo, pois você pode apenas trocar ponteiros em vez de copiar dados, mas se você deseja um acesso aleatório a elementos, não pode simplesmente visualizar diretamente esse elemento como pode com um vetor, é preciso começar a partir do ponteiro principal e iterar por todos os elementos até chegar ao que está buscando.

Quadro 1 – Comparação das características entre as duas classes usadas neste trabalho.

Vetores	**Lista duplamente ligada**
Acesso aleatório a elementos em tempo constante.	Sem acesso aleatório a elementos.
Boa localidade de cache.	Má localidade de cache, os elementos não são contíguos na memória.
Os elementos são contíguos na memória.	Inserções/deleções são rápidas em qualquer ponto da lista.
Inserções/deleções no meio/início do vetor são lentas.	Referências/iteradores para elementos permanecem válidos quando outros elementos são inseridos/excluídos (porém, depende da implementação).

3. METODOLOGIA

Para este trabalho foi implementado os métodos para cada uma das implementações (Array/Vetor de tamanho fixo, com alocação dinâmica, e listas duplamente encadeadas). Para cada classe foi implementado um arquivo de cabeçalho C++ (.hpp). Os métodos foram comentados, com explicações detalhadas do código e o seu desempenho, usando a notação big-Oh. Os arquivos de testes estão anexados separadamente.

Também foram desenvolvidos casos de testes para realizar testes de correção e de desempenho, em adição a uma análise comparativa entre as formas de implementação, array alocado dinamicamente e lista duplamente ligada, com indicações de quando usar cada uma. Os casos de testes desenvolvidos foram: Inserção consecutiva de elementos no início do vetor; Inserção consecutiva de elementos no final do vetor e conjunto de remoções de elementos pelo índice.

Para a realização dos testes de inserção no arraylist, foram executados três 3 (três) formas distintas de aumento do array: começando com capacidade 100 (cem) e aumentando em 100 quando necessário, começando com capacidade 1000 (mil) e aumentando em 1000 e começa com capacidade 8 (oito) e duplicando cada vez que precisar de mais. Os testes de inserção da lista duplamente foram feitos apenas começando capacidade 8 (oito) e duplicando.

3.1 IMPLEMENTAÇÃO

3.1.1 Increase_capacity

Arraylist:

```
class array_list {
private:
    int* data; //Ponteiro.
    unsigned int size_, capacity_;

    void increase_capacity() { //O(n) n=quantidade de elementos.
        int *new_data = new int[this->capacity_ *2]; // Aloca um novo array com o dobro da capacidade atual.
        for (unsigned int i=0 ; i<this->size() ; ++i){ // Copia os dados do array antigo para o novo array.
            new_data[i] = this->data[i]; }
        delete [] this->data; // Deleta o array antigo.
        this->data = new_data; // Atualiza o ponteiro `data` para apontar para o novo array.
        this->capacity_ = this->capacity_*2; // Atualiza a capacidade máxima do array.
    }
```

3.1.2 Construtor e destrutor

Arraylist:

```
public:
    array_list() {  //construtor
        data = new int[8]; //Aloca um array de tamanho 8.
        this->size_ = 0; //Inicializa o tamanho atual do array para 0.
        this->capacity_ = 8; //Inicializa a capacidade máxima atual do array para 8.
    }
    ~array_list() {   //Deleta o array que armazena os dados do objeto. O(1)
        delete[] data;
    }
```

Linkedlist:

```
public:
    Node* head;
    Node* tail;
    int size;
    linked_list() : head(nullptr), tail(nullptr), size(0) {} //Construtor para criar uma ll vazia
```

3.1.3 Size

Arraylist:

```
unsigned int size() { // Retorna a quantidade de elementos armazenados
    return this->size_; //O(1)
}
```

Linkedlist:

```
int Size() {
    return size;
}
```

3.1.4 Capacity

Arraylist:

```
unsigned int capacity() { // Retorna o espaço reservado para armazenar os elementos
    return this->capacity_; //O(1)
}
```

Linkedlist:

```
int capacity() {
    return -1;
}
```

3.1.5 Percent_occupied

Arraylist:

```
double percent_occupied() { //Retorna um valor entre 0.0 a 1.0 com o percentual da memória usada.
    return (double)size_ / (double)capacity_; //O(1)
}
```

Linkedlist:

```
int capacity() {
    return -1;
}

double percent_occupied() {
    return -1.0;
}
```

3.1.6 Insert_at

Arraylist:

```
bool insert_at(unsigned int index, int value) {  //O(n) / Insere elemento no indice index/ N é a quantidade de elementos
    if ( index > this->size_ ) // Verifica se o índice é válido.
            return false;
        if ( index == this->size_ ){ // Se o índice for o último elemento do array, insere o elemento no final do array.
            this->push_back(value);
            return true;}
        if ( this->size_ == this->capacity_ ) // Aumenta a capacidade do array se necessário.
            increase_capacity();
        for ( unsigned int i = this->size_; i > index; i-- ) // Desloca os elementos do array para a direita
            this->data[i] = this->data[i-1];
        this->data[index] = value; // Insere o novo elemento no array
        this->size_++;  // Aumenta o tamanho do array e retorna `true`
        return true;
    }
```

Linkedlist:

```
bool insert_at(int index, int data) { // inserir um elemento em um índice especificado na lista
    if (index < 0 || index > size) {
        return false;
    }

    Node* newNode = new Node(data);
    if (size == 0) { // Inserindo em uma lista vazia
        head = newNode;
        tail = newNode;
    } else if (index == 0) { // Inserindo no início
        newNode->next = head;
        head->prev = newNode;
        head = newNode;
    } else if (index == size) { // Inserindo no fim
        newNode->prev = tail;
        tail->next = newNode;
        tail = newNode;
    } else {    // Inserindo no meio da lista
        Node* curr = head;
        for (int i = 0; i < index; i++) {
            curr = curr->next;
        }
        newNode->prev = curr->prev;
        newNode->next = curr;
        curr->prev->next = newNode;
        curr->prev = newNode;
    }
    size++;
    return true;
}
```

3.1.7 Remove_at

Arraylist:

```
bool remove_at(unsigned int index) { // O(n) remove elemento no indice index
    if (index >= this->size_)
        return false; // Não removeu
    for (unsigned i = index + 1; i < this->size_; ++i) {
        this->data[i - 1] = this->data[i];
    }
    this->size_--;
    return true; // Removeu
}
```

Linkedlist:

```
bool remove_at(int index) { //remover um elemento em um indice especificado na lista
    if (index < 0 || index >= size) {
        return false;
    }

    Node* curr = head;
    if (index == 0) {
        head = head->next;
        if (head) {
            head->prev = nullptr;
        } else {
            tail = nullptr;
        }
    } else if (index == size - 1) {
        curr = tail;
        tail = tail->prev;
        if (tail) {
            tail->next = nullptr;
        } else {
            head = nullptr;
        }
    } else {
        for (int i = 0; i < index; i++) {
            curr = curr->next;
        }
        curr->prev->next = curr->next;
        curr->next->prev = curr->prev;
    }
    delete curr;
    size--;
    return true;
}
```

3.1.8 Get_at

Arraylist:

```
int get_at(unsigned int index) { //retorna elemento no indice index. O(1).
    if(index >= size_){
        return -1;}
    else return this->data[index];
}
```

Linkedlist:

```
int get_at(int index) { // Retorna elemento no índice index
    if (index < 0 || index >= size) {
        throw std::out_of_range("Index out of range.");
    }

    Node* curr = head;
    for (int i = 0; i < index; i++) {
        curr = curr->next;
    }
    return curr->data;
}
```

3.1.9 Clear

Arraylist:

```
void clear() { //remove todos os elementos do vetor deixando ele no estado inicial. O(1).
    this->size_ = 0;
    this->capacity_=8;
    this->data = new int[8];
    delete [] this->data;
    }
```

Linkedlist:

```
void clear() { // Remove todos os elementos, deixando o vetor no estado inicial
    Node* curr = head;
    while (curr) {
        Node* temp = curr;
        curr = curr->next;
        delete temp;
    }
    head = nullptr;
    tail = nullptr;
    size = 0;
}
```

3.1.10 Push_front

Arraylist:

```
void push_front(int value) {//adiciona um elemento no inicio do vetor O(n)
if (this->size_ == capacity_) // Verifica se o array precisa ser expandido
    increase_capacity();
for ( unsigned int i = this->size_; i > 0; i--) // Desloca os elementos do array para a direita
        this->data[i] = this->data[i-1];
      this->data[0] = value;  // Insere o novo elemento no array na posição 0
      this->size_++;} // Aumenta o tamanho do array
```

Linkedlist:

```
void push_front(int data) { // Adiciona um elemento no ``início'' do vetor
    insert_at(0, data);
}
```

3.1.11 Push_back

Arraylist:

```
void push_back(int value) { //adiciona um elemento no final do vetor O(1).
    if (this->size_ == this->capacity_) // Verifica se o array precisa ser expandido
        this->increase_capacity();
    this->data[this->size_++]=value; // Insere o novo elemento no array na última posição
    }
```

Linkedlist:

```
void push_back(int data) { // Adiciona um elemento no ``final'' do vetor
    insert_at(size, data);
}
```

3.1.12 Pop_front

Arraylist:

```
bool pop_front() { //remove um elemento do inicio do vetor O(n)
    if (this->size_==0) // Verifica se o array está vazio
            return false;
        for ( unsigned i=1;i< this->size_; i++) // Desloca os elementos para a esquerda
            this->data[i-1] = this->data[i];
        this->size_--; // Remove o elemento do início do array
        return true;}
```

Linkedlist:

```
   void pop_front() { // Remove um elemento do ``início'' do vetor
       remove_at(0);
   }
```

3.1.13 Pop_back

Arraylist:

```
bool pop_back() { //remove um elemento do final do vetor O(1).
    if ( this->size_ == 0 ) // Verifica se o array está vazio
            return false;
        this->size_--; // Remove o elemento do final do array
        return true;}
```

Linkedlist:

```
   void pop_back() { // Remove um elemento do ``final'' do vetor
       remove_at(size - 1);
   }
```

3.1.14 Front

Arraylist:

```
int front(){//retorna o elemento do inicio do vetor O(1).
    if (this->size_==0) //Verifica se está vazio
        return -1;
    return data[0];} // Retorna o elemento do início do array
```

Linkedlist:

```
int front() { // Retorna o elemento do ``início'' do vetor
    if (empty()) {
        throw std::runtime_error("List is empty.");
    }
    return head->data;
}
```

3.1.15 Back

Arraylist:

```
int back(){//retorna o elemento do final do vetor O(1).
    if (this->size_==0) // Verifica se o array está vazio
        return -1;
    return data[size_-1];} // Retorna o ultimo elemento
```

Linkedlist:

```
int back() { // Retorna o elemento do ``final'' do vetor
    if (empty()) {
        throw std::runtime_error("List is empty.");
    }
    return tail->data;
}
```

3.1.16 Remove

Arraylist:

```
bool remove(int value) {//remove value do vetor caso esteja presente O(n).
    for (unsigned int i = 0; i < this->size(); i++){  // Percorre o array procurando o elemento
        if(this->find(value)){
            for(unsigned int j=i; j <this->size() - 1; j++){ //executa se encontrar o elemento
                this->data[j] = this->data[j+1];
            }
            this->size_--; // Remove o elemento do array
            return true;
        }
    }
    return false;
    }
```

Linkedlist:

```
bool remove(int data) { // remove o valor caso esteja presente
    int index = find(data);
    if (index != -1) {
        remove_at(index);
        return true;
    }
    return false;
}
```

3.1.17 Find

Arraylist:

```
int find(int value) { // O(n)
    for (unsigned int i = 0 ; i<this->size() ; ++i) // Percorre o array procurando o elemento especificado
        if (value == this->data[i])
            return true;
    return -1;} // Retorna -1 para indicar que o elemento não foi encontrado
```

Linkedlist:

```
int find(int data) { // Retorna o índice de value, -1 se inválido
    Node* curr = head;
    int index = 0;

    while (curr) {
        if (curr->data == data) {
            return index;
        }
        curr = curr->next;
        index++;
    }
    return -1;
}
```

3.1.18 Count

Arraylist:

```
int count(int value) {//retorna quantas vezes value ocorre no vetor. O(n)
    int contar=0;
    for( unsigned i =0; i < this->size_; i++)
        if(this-> data[i] == value) contar++;
    return contar;}
```

Linkedlist:

```
int count(int data) {  // Retorna quantas vezes value occorre no vetor
    int count = 0;
    Node* curr = head;
    while (curr) {
        if (curr->data == data) {
            count++;
        }
        curr = curr->next;
    }
    return count;
}
```

3.1.19 Sum

Arraylist:

```
int sum() {//retorna a soma dos elementos do vetor O(n)
    int sum = 0;
for (unsigned int i = 0; i < size_; i++) {
  sum += data[i];}
return sum;}
```

Linkedlist:

```
int sum() {  // Retorna a soma dos elementos do vetor
    int total = 0;
    Node* curr = head;
    while (curr) {
        total += curr->data;
        curr = curr->next;
    }
    return total;
}
```

Em relação a Notação Big oh, podemos dizer que:

Quadro 2 - Notação Big oh dos métodos aplicados.

Método	**Notação Big oh Lista duplamente ligada**	**Notação Big oh Arraylist**
push_front	O(1)	O(n)
push_back	O(1)	O(1)/O(n)
pop_front	O(1)	O(n)
pop_back	O(1)	O(1)
front	O(1)	O(1)

back	O(1)	O(1)
sum	O(n)	O(n)
insert_at	O(n)	O(n)
remove_at()	O(n)	O(n)
get_at()	O(n)	O(n)
size()	O(1)	O(1)
capacity()	O(1)	O(1)
clear()	O(1)	O(1)
find()	O(n)	O(n)
percent_ocupied()	O(1)	O(1)
count	O(n)	O(n)

4. RESULTADOS E DISCUSSÃO

4.1 INSERÇÃO NO INÍCIO DO VETOR

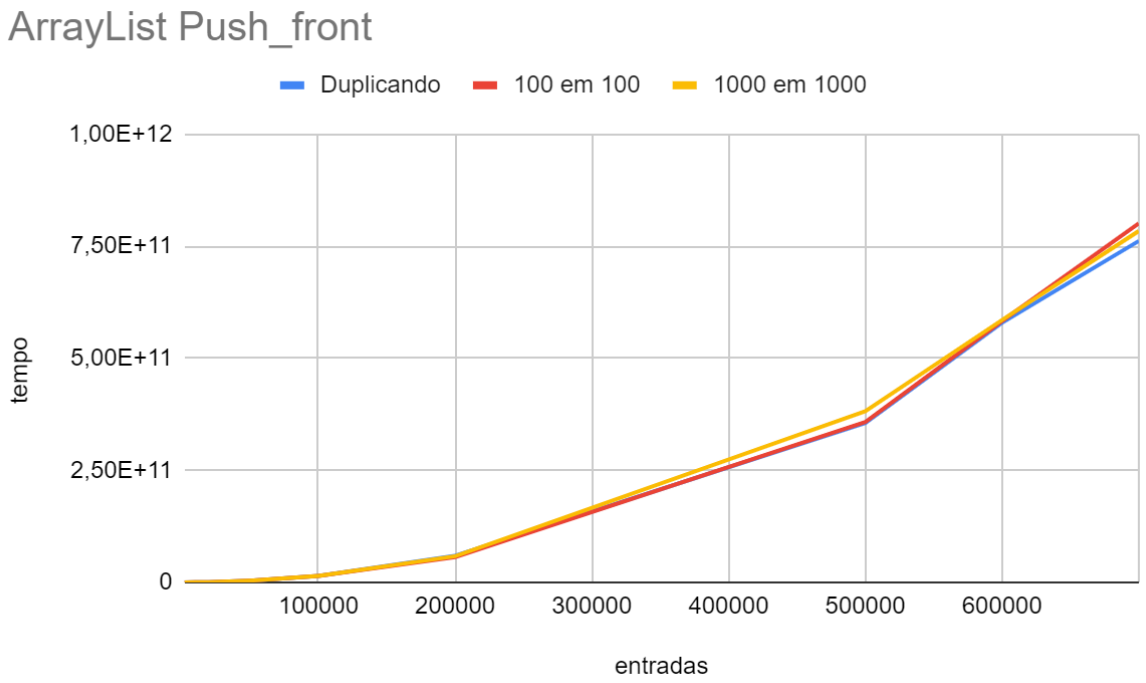
4.1.1 Pushfront Arraylist

Quadro 3 - Resultados de 3 diferentes formas do aumento do array obtidos após testes de inserção no início do vetor.

pushfront	2000	10000	20000	50000	100000	200000	500000	600000	700000
Duplicando	5085000	152127000	586932000	3591607000	14077550000	59152518000	355534140000	5796752460000	761764844000
100 em 100	5397000	145237000	589443000	3562523000	14596791000	56411287000	357914064000	5829381500000	801056948000

1000 em 1000	0	154912000	594099000	379703800	1422033700	582762910	382039898	585273391	784034051

Gráfico 1 - Teste de desempenho do método push_front no arraylist duplicando a capacidade, aumentando de 100 em 100 e 1000 em 1000.



Observa-se com testes de entrada com uma maior quantidade de elementos, a forma de duplicar a capacidade se torna mais eficiente em relação às outras, isso acontece porque à medida que o tamanho da entrada aumenta, a quantidade de memória necessária para armazenar a lista também aumenta. Se a capacidade do vetor for aumentada de 100 em 100, o vetor precisará ser realocado várias vezes à medida que a entrada cresce. Isso pode ser um processo caro, pois requer a cópia de todos os dados do vetor antigo para o novo vetor.

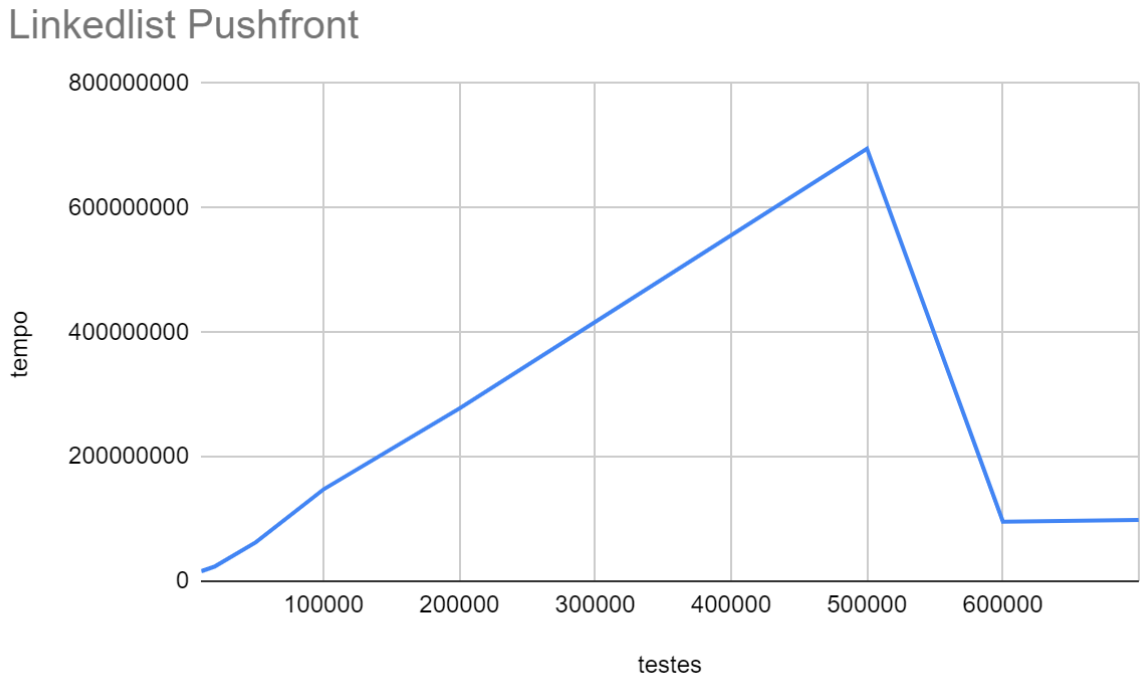
O método de duplicação, por outro lado, só precisa ser chamado uma vez, quando a entrada atinge a capacidade máxima do vetor. Isso economiza tempo e memória, pois evita a necessidade de realocar o vetor várias vezes. Além disso, mesmo que o método push front tenha um custo de O(n) para cada operação, o custo total de executar um número de operações push front pode ser O(1).

4.1.2 Pushfront Linkedlist

Quadro 4 - Resultados do teste pushfront na lista duplamente ligada duplicando a capacidade.

2000	10000	20000	50000	100000	200000	500000	600000	700000
0	16016000	23552000	62281000	147445000	277567000	695014000	95516000	98550000

Gráfico 2 - Teste de desempenho do método pushfront na linkedlist duplicando a capacidade.



Observando o gráfico é possível concluir que a eficiência do método pushfront na lista duplamente ligada é superior ao arraylist, já que não é necessário realocar mais memória para adicionar mais elementos. Isso significa que o método push front é executado de forma mais eficiente na lista duplamente ligada (O1), independentemente do tamanho da entrada.

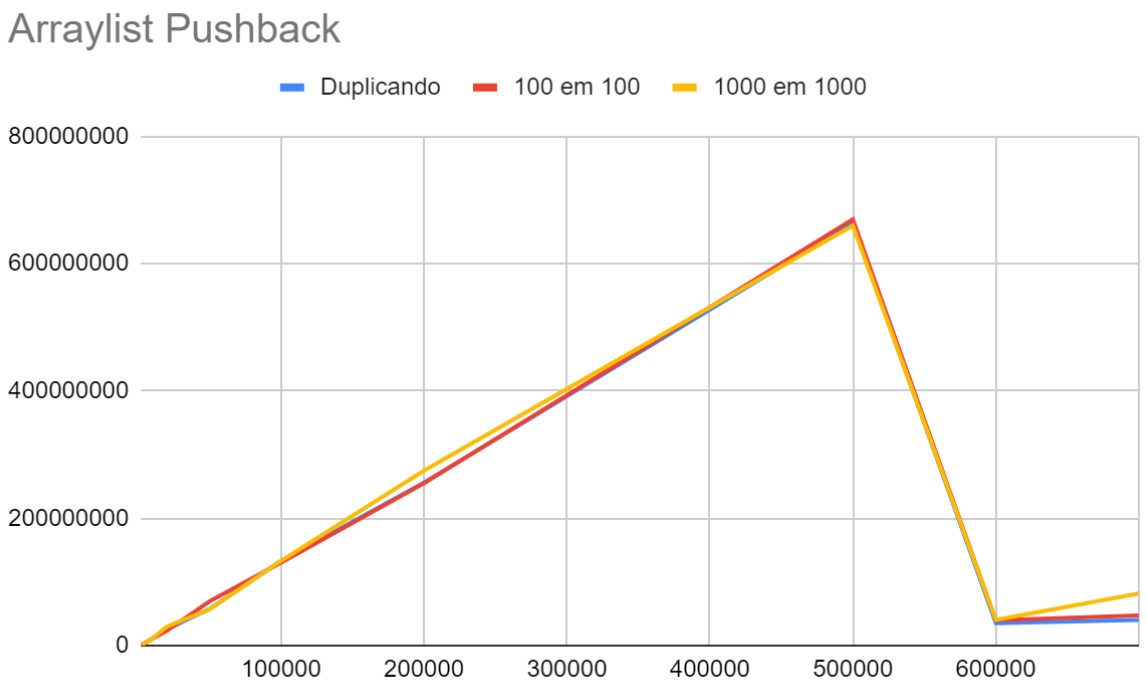
4.2 INSERÇÃO NO FINAL DO VETOR

4.2.1 Pushback Arraylist

Quadro 5 - Resultados de 3 diferentes formas do aumento do array obtidos após testes de inserção no final do vetor.

pushback	2000	10000	20000	50000	100000	200000	500000	600000	700000
Duplicando	0	11452000	25099000	57347000	133068000	255809000	663640000	35097000	40314000
100 em 100	0	10310000	22898000	69314000	130228000	255137000	669666000	39648000	47058000
1000 em 1000	0	10784000	29617000	56916000	132992000	274662000	659980000	40180000	81528000

Gráfico 3 - Teste de desempenho do método pushback no arraylist duplicando a capacidade, aumentando de 100 em 100 e 1000 em 1000.



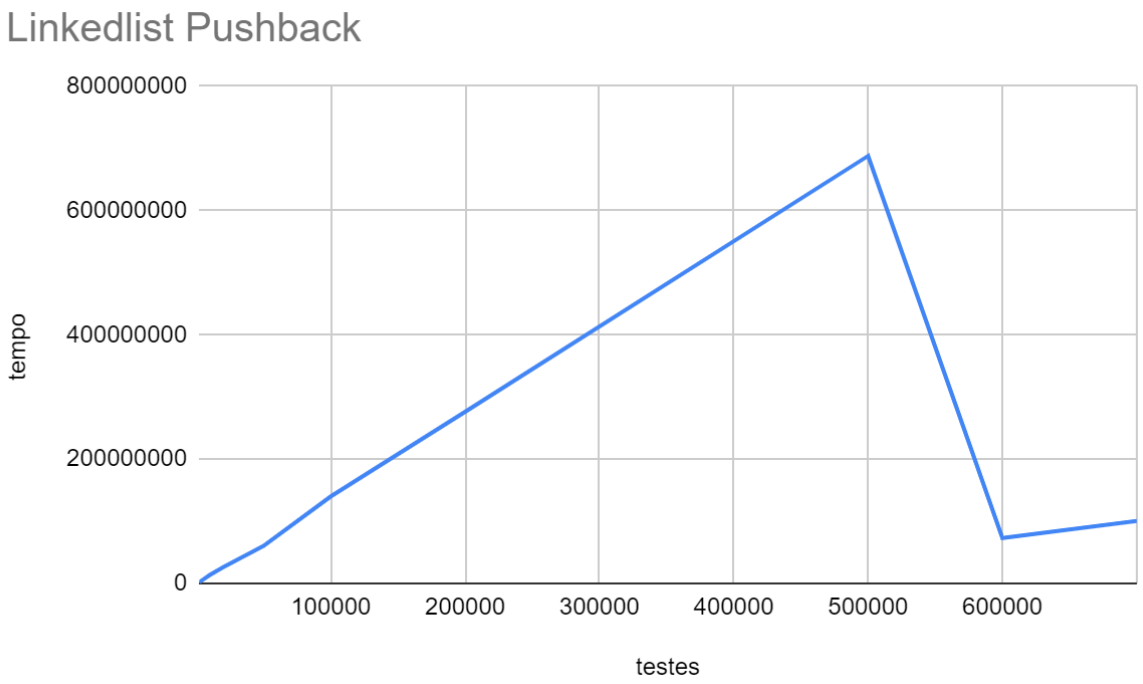
Assim como no método pushfront, levando em consideração a realocação de memória, duplicar a capacidade se mostra mais vantajoso com entradas maiores. O pushback no arraylist é O(1) se a capacidade do vetor for suficiente para acomodar o novo elemento. No entanto, se a capacidade do vetor for excedida, ele vai precisar ser realocado. Ou seja, O(1) na melhor das hipóteses e O(n) na pior das hipóteses.

4.2.2 Pushback Linkedlist

Quadro 6 - Resultados do teste pushback na lista duplamente ligada duplicando a capacidade.

2000	10000	20000	50000	100000	200000	500000	600000	700000
1508000	13540000	26199000	60223000	140101000	276147000	687950000	72994000	100257000

Gráfico 4 - Teste de desempenho do método pushback na linkedlist duplicando a capacidade.



O método pushback é executado de forma mais eficiente (O1) na lista duplamente ligada, independentemente do tamanho da entrada.

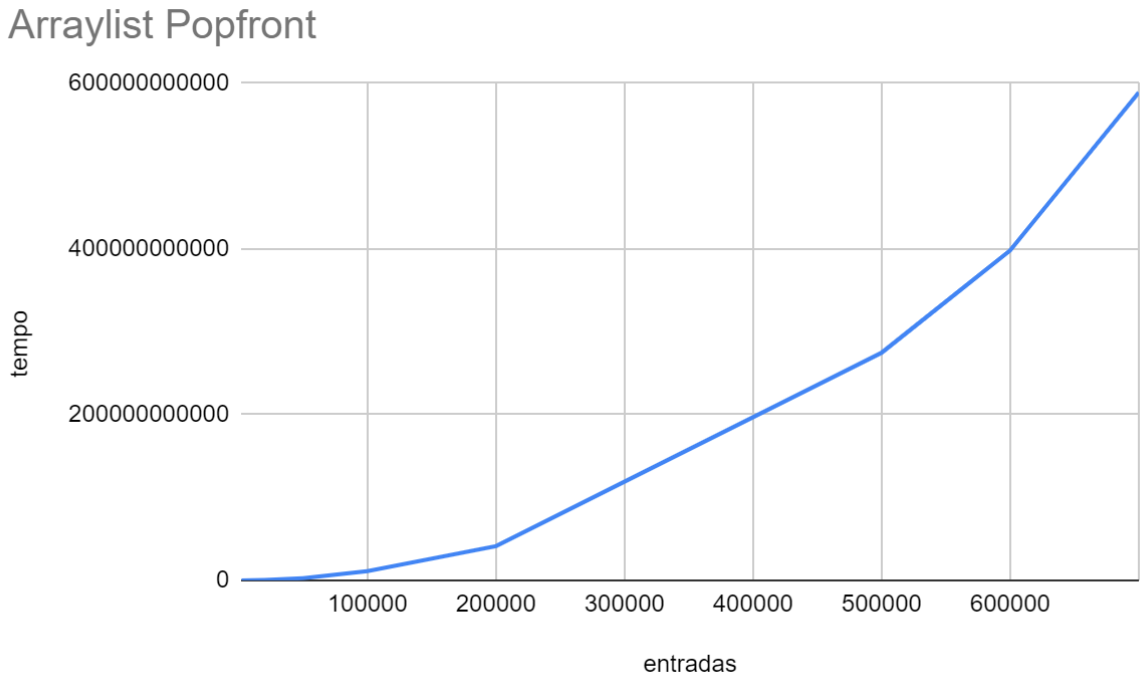
4.3 MÉTODOS DE REMOÇÃO

4.3.1 Popfront Arraylist e Linkedlist

Quadro 7 - Teste de desempenho do método popfront no arraylist.

Arraylist	2000	10000	20000	50000	100000	200000	500000	600000	700000
duplicando	7494000	95426000	41969300 0	25802810 00	11089558 000	41229979 000	27476603 9000	39822109 3000	58882760900 0

Gráfico 5 - Teste de desempenho do método popfront no arraylist.

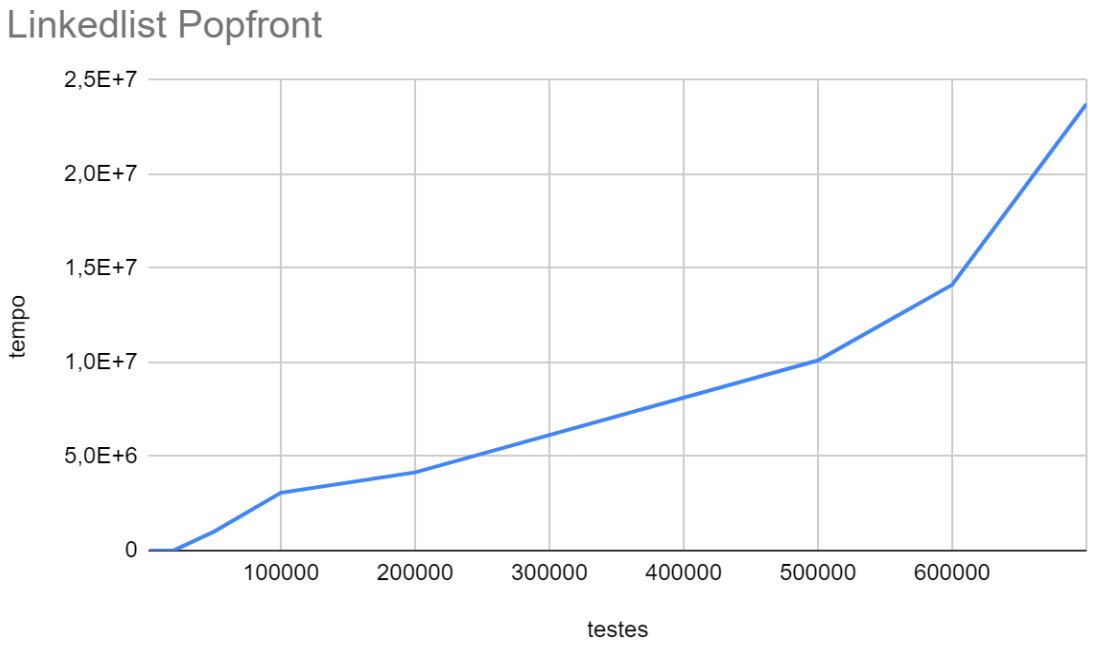


O método popfront no arraylist é O(n) na pior das hipóteses. Isso ocorre porque é necessário deslocar todos os elementos da lista para trás para preencher o espaço deixado pelo elemento excluído. O custo de deslocar os elementos da lista é proporcional ao tamanho da lista.

Quadro 8 - Teste de desempenho do método popfront na linkedlist.

Linkedlist	2000	10000	20000	50000	100000	200000	500000	600000	700000
duplicando	0	0	0	999000	3071000	4154000	10105000	14122000	23728000

Gráfico 6 - Teste de desempenho do método popfront no arraylist.



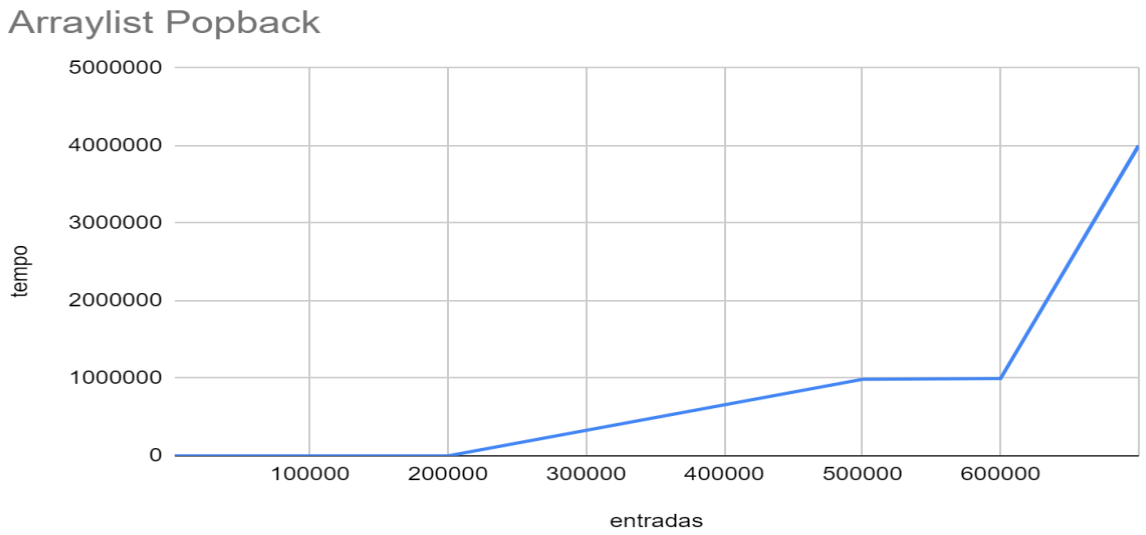
Já na lista duplamente ligada, o método popfront é O(1), porque ele remove o elemento do início da lista modificando os ponteiros do elemento anterior e do próximo elemento do elemento a ser removido.

4.3.2 Popback Arraylist e Linkedlist

Quadro 9- Teste de desempenho do método popback no arraylist.

Arraylist	2000	10000	20000	50000	100000	200000	500000	600000	700000
duplicando	0	0	0	0	0	0	985000	996000	4003000

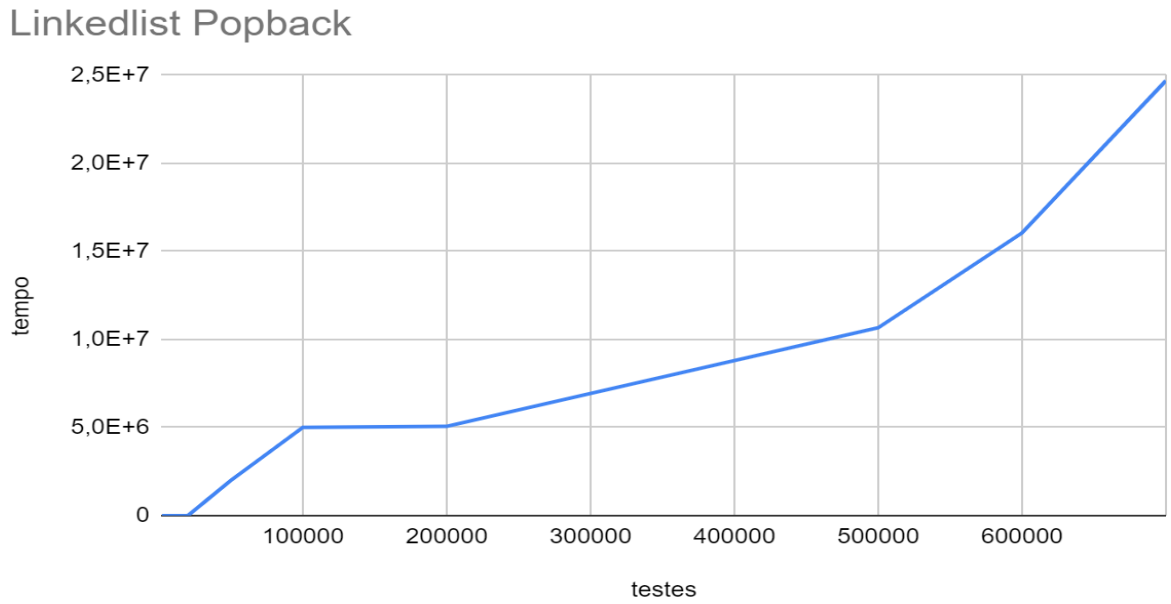
Gráfico 7 - Teste de desempenho do método popback no arraylist.



Quadro 10 - Teste de desempenho do método popback na linkedlist.

Linkedlist	2000	10000	20000	50000	100000	200000	500000	600000	700000
duplicando	0	0	0	2009000	5005000	5066000	10661000	16053000	24692000

Gráfico 8 - Teste de desempenho do método popback na linkedlist..

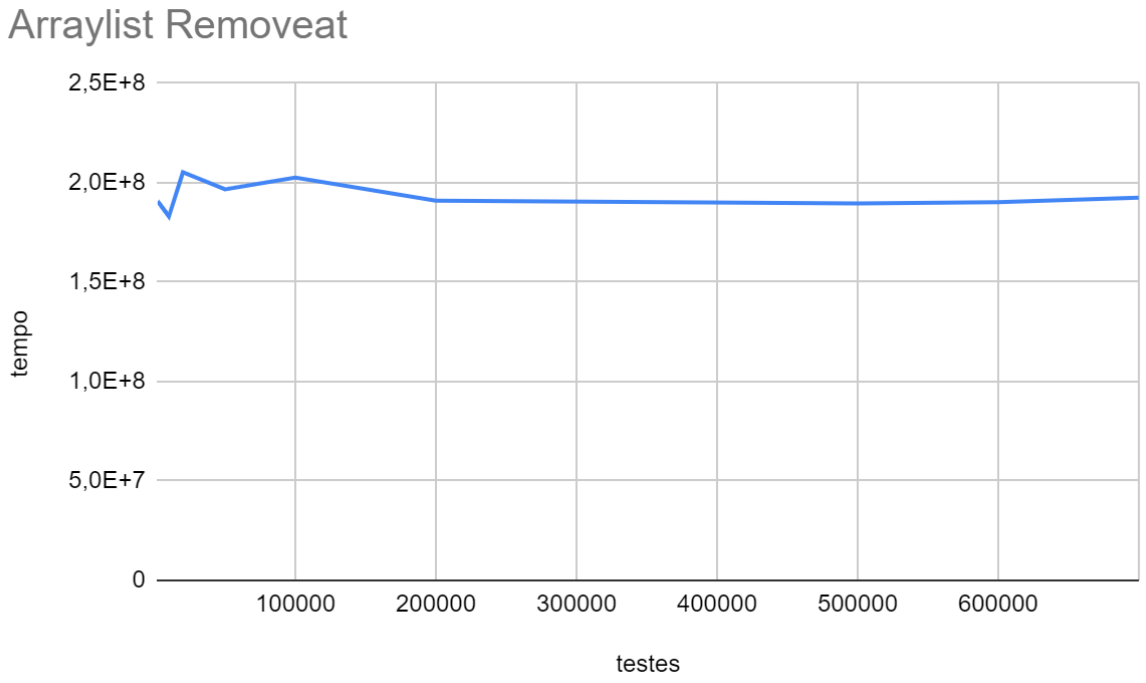


4.3.3 Remove_at Arraylist e Linkedlist

Quadro 11 - Teste de desempenho do método removeat no arraylist.

arraylist	2000	10000	20000	50000	100000	200000	500000	600000	700000
	190737000	183101000	205235000	196658000	202569000	190924000	189513000	190185000	192498000

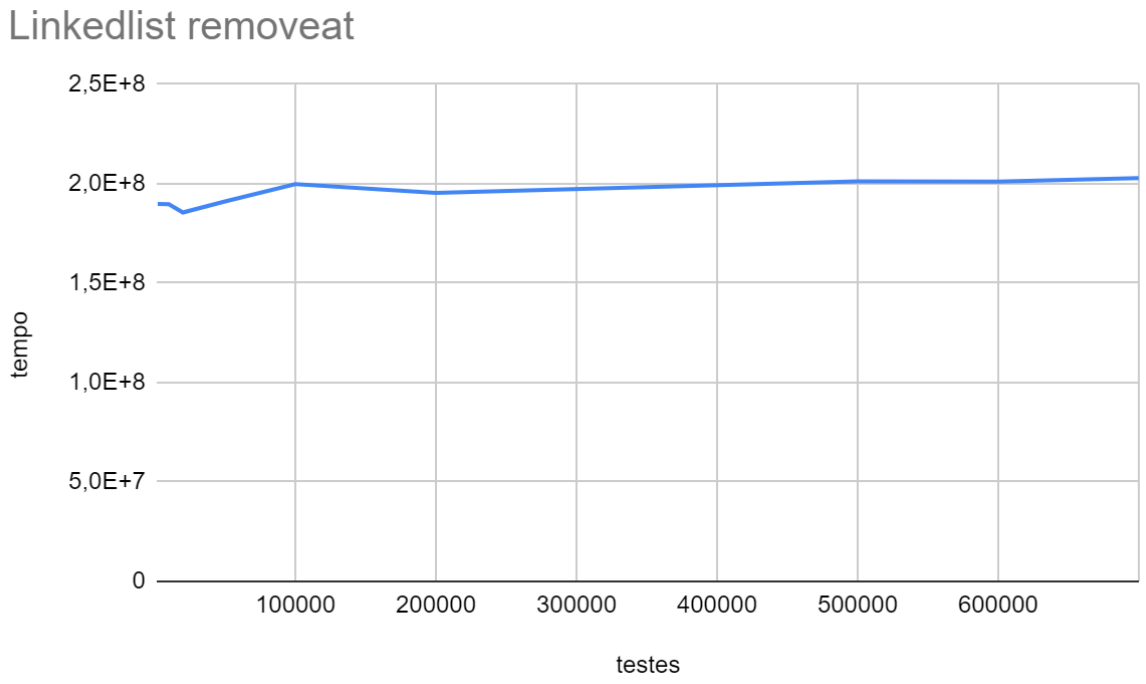
Gráfico 9 - Teste de desempenho do método removeat no arraylist.



Quadro 12 - Teste de desempenho do método removeat na linkedlist.

Linkedlist	2000	10000	20000	50000	100000	200000	500000	600000	700000
	189816000	189715000	185516000	191075000	199791000	195438000	201176000	201056000	202866000

Gráfico 10 - Teste de desempenho do método removeat na linkedlist.



Na maioria dos casos a complexidade do método removeat na lista duplamente ligada é O(1), porém se o elemento a ser removido for o primeiro ou o último elemento da lista a complexidade será O(n) (pior caso). Já no arraylist, esse método funciona iterando sobre a lista, a partir do índice especificado, e movendo todos os elementos restantes para trás para preencher o espaço deixado pelo elemento excluído O(n).

5. CONCLUSÃO

Vetores dinâmicos de números inteiros e listas duplamente ligadas são duas estruturas de dados comumente usadas para armazenar sequências de elementos ordenados. Vetores dinâmicos são mais eficientes para acessar elementos aleatórios da lista, enquanto que as listas duplamente ligadas são mais eficientes para inserir ou remover elementos no meio da lista. A escolha da estrutura de dados mais adequada para uma aplicação específica depende dos requisitos específicos da aplicação.

REFERÊNCIAS

https://cplusplus.com/reference/array/array/

https://cplusplus.com/reference/vector/vector/

https://cplusplus.com/reference/list/list/

https://www.interviewcake.com/concept/java/dynamic-array-amortized-analysis

Programação: Listas Duplamente Encadeadas Prof. André Grégio

https://en.cppreference.com/w/cpp/container/vector