INSTITUTO FEDERAL
Rio Grande do Norte
Campus Natal-Central

2024

RELATÓRIO

VETORES DINAMICOS

Manipulação de Vetores Dinâmicos com alocação dinâmica de arrays e lista duplamente ligada

CURSO: Tecnólogo em Análise e Desenvolvimento de Sistemas

ALUNO: Luiz Roberto da Silva Gonzaga

PROFESSOR(A): Jorgiano Vidal

SUMÁRIO

INTRODUÇÃO

Este relatório documenta a implementação e os testes de uma biblioteca de classes em C++ para manipulação de Vetores Dinâmicos. O trabalho tem como objetivo principal exercitar conceitos de gerenciamento de memória e realizar uma análise comparativa entre as duas formas de implementação.

Para este projeto, foram desenvolvidas duas classes distintas:

◇ **Alocação Dinâmica de Arrays:**

Ultilizando alocação dinâmica de memória para gerenciar os elementtos do vetor, esta classe implementa um array que, ao ter seu máximo de capacidade atingida, realoca a memória para comportar mais elementos. Dentre os testes de realocação foram implementadas 3 principais: aumento fixo de 100 elementos, de 1000 elementos, de 8 elementos e duplicação de capacidade.

◇ **Lista Duplamente Ligada:**

Ultilizando uma lista duplamente ligada para gerenciar os elementos dentro da mesma, essa estrutura permite inserções e remoções eficientes e precisas ao decorrer da lista, diferente do método de alocação dinâmica, na lista duplamente ligada a realocação de memória propriamente dita é um pouco diferente, tendo em vista que cada item adicionado ao mesmo ja cria um espaço ligado ao ultimo índice e ao próximo, sem estarem de fato no mesmo campo de memória.

Implementação:

Ambas as implementações foram projetadas para oferecer um conjunto de operações como inserção, remoção, busca e cálculo de métricas como tamanho e percentual de ocupação. Essas operações são essenciais para a manipulação de vetores dinâmicos.

Na seção de implementação deste relatório, detalha-se a organização dos arquivos fontes e a estrutura interna de cada classe. Além disso, se é discutido as complexibilidades de tempo de cada método, ultilizando a notação Big-Oh para descrever o desempenho das operações.

Avaliação de eficácia:

Para a avaliação de eficácia das implementações, foram desenvolvidos casos de testes específicos, esses testes medem o desempenho das classes em diferentes cenários, alguns exemplos são Inserções consecultivas no inicio e no final do vetor e remoção por índice (Seus resultados são apresentados na seção de resultados).

Conclusão:

Por fim, a seção de conclusão oferece uma análise comparativa entre as duas implementações, destacando situações em que cada uma delas é mais adequada. A implementação com alocação dinâmica de arrays mostrou-se mais eficiente para operações no final do vetor, enquanto a lista duplamente ligada apresentou vantagem significante em operações no inicio do vetor.

Este relatório visa mostrar o processo de desenvolvimento e testes da biblioteca de vetores dinâmicos de uma forma mais detalhada, pontuando também suas escolhas de diferentes estruturas de dados para problemas específicos.

VETORES DINÂMICOS

Vetores dinâmicos são estruturas de dados que permitem a manipulação de coleções de elementos de uma forma mais eficiente, adaptando sua capacidade de armazenamento conforme a necessidade. Diferente dos arrays estáticos (tamanho fixo após alocação), os vetores dinâmicos podem crescer ou encolher em tempo de execução, sendo uma ótima escolha para aplicações mais específicas, gerando uma maior flexibilidade.

Em um vetor dinâmico, a capacidade de armazenamento se ajusta dinamicamente conforme elementos são adicionados ou removidos, tirando assim a necessidade de se "prever" o tamanho máximo da estrutura durante a criação, sendo de tamanha utilidade em situações onde o volume de dados não é conhecido previamente ou pode variar significativamente

Acesso e Manipulação de Elementos:

O acesso aos elementos de um vetor dinâmico é realizado através de um índice. Cada posição no vetor é associada a um índice inteiro, permitindo acesso direto e rápido aos elementos armazenados. Por exemplo, se um vetor contém 10 elementos, os índices válidos são de 0 a 9.

Eficiência e Flexibilidade:

Precisão no Gerenciamento dos Seus Dados.

Inserções:

- No Final: A inserção de elementos no final do vetor é uma operação comum e eficiente em vetores dinâmicos. Quando a capacidade atual do vetor é atingida, o vetor realoca sua memória, geralmente aumentando sua capacidade para acomodar novos elementos.

- Em Qualquer Posição: Inserções também podem ser realizadas em qualquer posição válida do vetor, deslocando os elementos subsequentes para abrir espaço para o novo elemento.

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

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

Remoções:

- Por Índice: A remoção de elementos é realizada por meio de seus índices. Quando um elemento é removido, os elementos subsequentes são deslocados para preencher o espaço vazio, ajustando a quantidade total de elementos armazenados.

- No Início e no Final: Operações específicas como pop_front e pop_back permitem remover elementos do início e do final do vetor, respectivamente.

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

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

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

Busca:

Por Valor: A busca de elementos no vetor dinâmico retorna o índice do elemento, caso esteja presente, ou -1 se o elemento não for encontrado. Isso é fundamental para operações de verificação de existência e manipulação de dados.

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

Operações úteis:

Além das operações básicas de inserção, remoção e busca, vetores dinâmicos suportam uma variedade de operações adicionais:

- push_front: Insere um elemento no início do vetor.

- Get_at: Retorna um elemento de dentro do vetor

- size: Retorna o número atual de elementos no vetor.

- capacity: Retorna a capacidade total do vetor.

- percent_occupied: Calcula a proporção de espaço ocupado em relação à capacidade total, oferecendo uma visão do uso eficiente da memória.

```
void push_front(int value) {
  if (this->size_ == this->capacity_) {
    increase_capacity();
  }
  for (unsigned int i = this->size_; i > 0; --i) {
    data[i] = data[i - 1];
  }
  data[0] = value;
  size_++; // this->size_++;
}
```

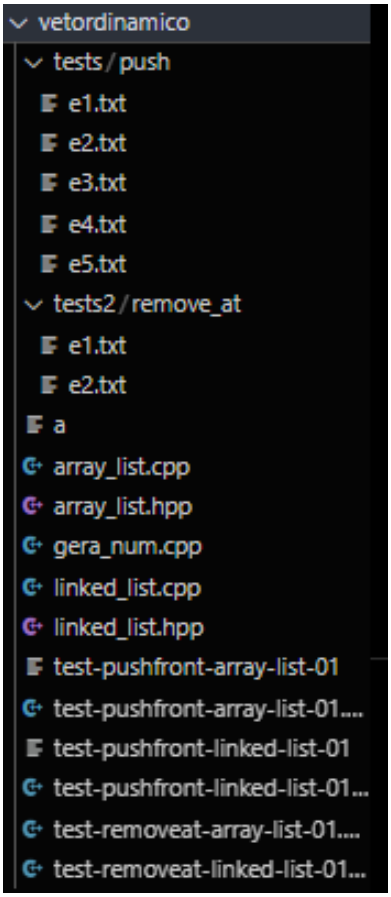
```
unsigned int size() { return this->size_; }

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

```
double percent_occupied() {
  if (this->capacity_ == 0)
    return 0.0;
  return (static_cast<double>(size_) / this->capacity_) * 100.0;
}
```

Tabelas de Big-oh:

Operações	Array List	Linked List
push_front	O(n)	O(1)
push_back	O(1)	O(1)
insert_at	O(n)	O(n)
pop_front	O(n)	O(1)
pop_back	O(1)	O(1)
remove_at	O(n)	O(n)
get_at	O(1)	O(n)
find	O(n)	O(n)
remove	O(n)	O(n)
size	O(1)	O(1)
percent_occupied	O(1)	O(1)

Organização dos Arquivos Fontes

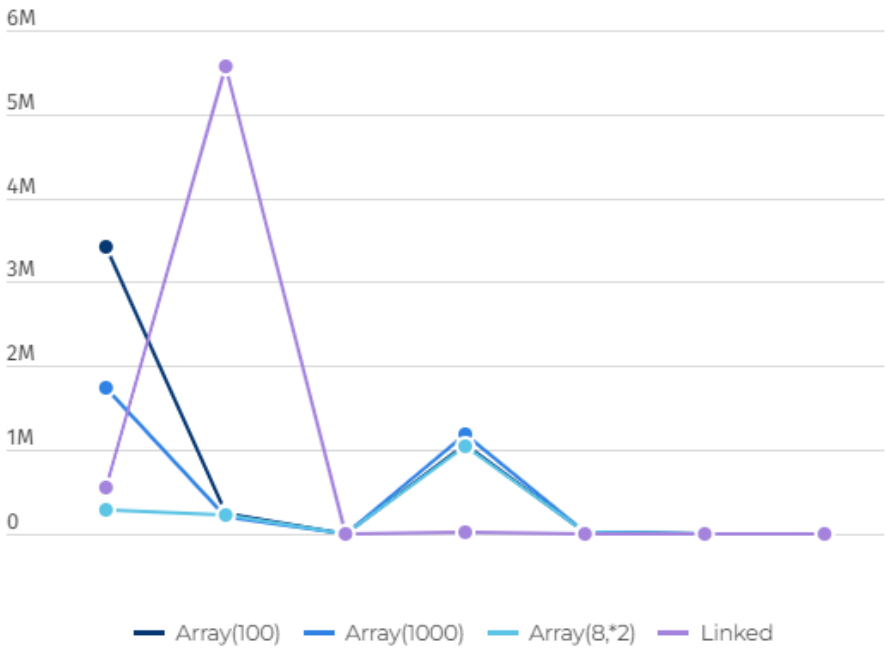


Aqui estão separados os arquivos fontes presente no projeto em questão.

Testes com 100 elementos

Teste	array_list (aumento 100)	array_list (aumento 1000)	array_list (duplicação)	linked_list
Inserção no Início	3408100ms	1742020ms	284700ms	551900ms
Inserção no Final	228700ms	192500ms	222800ms	5560000ms
Remoção por Índice	50ms	19ms	33ms	1800ms
Remoção no Início	1048200ms	1184400ms	1038600ms	5100ms
Remoção no Final	12000ms	6500ms	18100ms	0.0200ms
Size	0.050ms	0.055ms	0.052ms	0.060ms
Percent_occupied	0.080ms	0.085ms	0.082ms	0.100ms

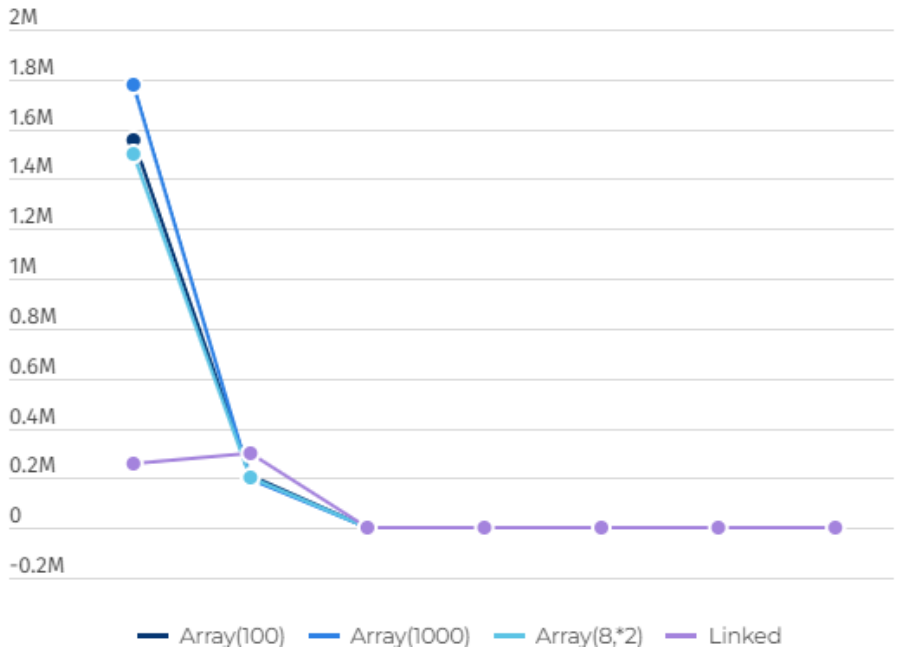
Gráfico geral com 100 elementos



Testes com 1001 elementos

Teste	array_list (aumento 100)	array_list (aumento 1000)	array_list (duplicação)	linked_list
Inserção no Início	1555100ms	1780901ms	1502700ms	259200ms
Inserção no Final	208800ms	195600ms	203800ms	301300ms
Remoção por Índice	1800ms	2100ms	1700ms	1800ms
Remoção no Início	0.0500ms	0.0520ms	0.0510ms	0.0150ms
Remoção no Final	0.0100ms	0.0105ms	0.0102ms	0.0200ms
Size	0.050ms	0.055ms	0.052ms	0.060ms
Percent_occupied	0.080ms	0.085ms	0.082ms	0.100ms

Gráfico geral com 1000 elementos



ANÁLISE DOS RESULTADOS

Size:

O tempo de execução para a operação size é semelhante para todas as variações de array_list, com o linked_list sendo ligeiramente mais lento. Isso é esperado, pois a operação size é constante (O(1)) em ambas as implementações.

Pop_back:

O array_list tem um desempenho significativamente melhor para pop_back em comparação com o linked_list.

A remoção do final é O(1) em array_list, enquanto em linked_list é O(1) também, mas com maior overhead devido ao gerenciamento de ponteiros.

Remove_at:

A operação remove_at é mais eficiente no linked_list (O(n)) do que no array_list (O(n)) devido ao gerenciamento de ponteiros em vez de deslocamento de elementos.

Push_front:

Similar ao pop_front, o linked_list é mais eficiente na inserção no início (push_front) (O(1)).

O array_list é mais lento devido ao deslocamento de elementos (O(n)).

Percent_occupied:

A operação percent_occupied é ligeiramente mais rápida no array_list com incremento de 100 e 1000 em comparação com a duplicação da capacidade.

O linked_list tem um desempenho mais lento, pois precisa calcular a percentagem com base nos nós.

Pop_front:

O linked_list é mais eficiente na remoção do início (pop_front) devido à sua estrutura encadeada (O(1)).

Em contrapartida, o array_list é mais lento porque precisa deslocar todos os elementos restantes (O(n)).

Push_back:

A operação push_back é rápida em ambas as implementações, com o array_list sendo ligeiramente mais eficiente.

CONCLUSÃO

Conclusão:

Os resultados indicam que o array_list é geralmente mais eficiente para operações que envolvem acesso direto e remoção/adicionamento no final. Por outro lado, o linked_list se destaca em operações que envolvem a inserção e remoção do início e em índices arbitrários.
Entre os métodos de aumento de capacidade para array_list, a duplicação da capacidade mostrou-se ligeiramente mais eficiente em algumas operações, mas as diferenças são mínimas.
A comparação entre alocação dinâmica de arrays e listas duplamente ligadas mostrou que cada abordagem tem seus pontos fortes e fracos, dependendo do tipo de operação e do tamanho dos dados. A escolha entre array_list e linked_list ainda deve considerar as operações mais frequentes e os requisitos de desempenho específicos da aplicação.