

# MC-202

## Vetores

Lehilton Pedrosa  
lehilton@ic.unicamp.br

Universidade Estadual de Campinas

Segundo semestre de 2024

# Vetores

Vetores são uma forma nativa do C de **estruturar dados**

- É uma lista **indexada** de itens
- Estão presentes em muitas outras linguagens também

Em C, um vetor é um bloco sequencial de memória

Ele pode ser alocado:

- estaticamente — `int v[100];`
- dinamicamente — `int *v = malloc(100*sizeof(int));`

A sua grande vantagem é o acesso em **tempo constante** a qualquer um dos seus elementos através do índice

# TAD Vetor — Interface

```
1 typedef struct vetor *p_vetor;
2
3 struct vetor {
4     int *dados;
5     int n;
6 };
7
8 p_vetor criar_vetor(int tam);
9
10 void destruir_vetor(p_vetor v);
11
12 void adicionar_elemento(p_vetor v, int x);
13
14 void remover_elemento(p_vetor v, int i);
15
16 int busca(p_vetor v, int x);
17
18 void imprime(p_vetor v);
```

- `p_vetor` é um ponteiro para `struct vetor`
- Precisaremos para mudar os campos de um `struct vetor`
- Código ficará mais limpo usando `p_vetor`
- Vamos sempre manipular via ponteiro

# TAD Vetor — Implementação

```
1 #include "vetor.h"
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 p_vetor criar_vetor(int tam) {
6     ...
7 }
8
9 void destruir_vetor(p_vetor v) {
10    ...
11 }
12
13 void adicionar_elemento(p_vetor v, int x) {
14    ...
15 }
16
17 void remover_elemento(p_vetor v, int x) {
18    ...
19 }
20
21 int busca(p_vetor v, int x) {
22    ...
23 }
24
25 void imprime(p_vetor v) {
26    ...
27 }
```

Veremos três implementações diferentes

- as três fazem as mesmas coisas
- mas levam **tempos** diferentes

# Inicialização/Destruição

Código no cliente:

```
1  p_vetor v;  
2  v = criar_vetor(100);
```

Código em vetor.c:

```
1  p_vetor criar_vetor(int tam) {  
2    p_vetor v;  
3    v = malloc(sizeof(struct vetor));  
4    v->dados = malloc(tam * sizeof(int));  
5    v->n = 0;  
6    return v;  
7 }
```

Código no cliente:

```
1  destruir_vetor(v);
```

Código em vetor.c:

```
1  void destruir_vetor(p_vetor v) {  
2    free(v->dados);  
3    free(v);  
4 }
```

## Inserção e Remoção

Não estamos preocupados em manter nenhum tipo de ordem

Inserção em  $O(1)$  (tempo constante):

- inserimos no final do vetor

```
1 void adicionar_elemento(p_vetor v, int x) {
2     v->dados[v->n] = x;
3     v->n++;
4 }
```

Remoção em  $O(1)$ :

- trocamos o elemento a ser removido com o último
- diminuimos  $n$

Removendo elemento na posição 2:

9	3	5	6	2	5	?	?	?	?
---	---	---	---	---	---	---	---	---	---

```
1 void remover_elemento(p_vetor v, int i) {
2     v->dados[i] = v->dados[v->n - 1];
3     v->n--;
4 }
```

# Busca

Busca sequencial em  $O(n)$  (linear)

```
1 int busca(p_vetor v, int x) {
2     int i;
3     for (i = 0; i < v->n; i++)
4         if (v->dados[i] == x)
5             return i;
6     return -1;
7 }
```

Não podemos fazer busca binária, já que o vetor não está ordenado...

# Buscas frequentes

Se as buscas no vetor forem muito frequentes:

- é mais vantajoso realizar uma busca binária

Poderíamos ordenar o vetor antes e realizar a busca binárias

Porém, ordenar custa:

- $O(n^2)$  usando InsertionSort, SelectionSort ou BubbleSort
- $O(n \lg n)$  usando outros algoritmos que veremos no curso

Só vale a pena se não tivermos que ordenar sempre

Outra opção: podemos manter o vetor ordenado



## Inserção — Vetor Ordenado

Para adicionar um elemento precisamos:

- encontrar sua posição correta
- deslocar os elementos para a direita
- inserir na posição correta

Inserindo 4:

2	3	4	5	6	7	9	?	?	?
---	---	---	---	---	---	---	---	---	---

```
1 void adicionar_elemento(p_vetor v, int x) {
2     int i;
3     for (i = v->n - 1; i >= 0 && v->dados[i] > x; i--)
4         v->dados[i + 1] = v->dados[i];
5     v->dados[i + 1] = x;
6     v->n++;
7 }
```

Tempo:  $O(n)$

## Remoção — Vetor Ordenado

Para remover um elemento precisamos:

- deslocar os elementos para a esquerda

Removendo elemento na posição **3**:

2	3	4	6	7	9	9	?	?	?
---	---	---	---	---	---	---	---	---	---

```
1 void remover_elemento(p_vetor v, int i) {
2   for(; i < v->n - 1; i++)
3     v->dados[i] = v->dados[i + 1];
4   v->n--;
5 }
```

Tempo:  $O(n)$

# Busca — Vetor Ordenado

Realizamos busca binária em  $O(\lg n)$

```
1 int busca_binaria(int *dados, int l, int r, int x) {
2     int m = (l+r)/2;
3     if (l > r)
4         return -1;
5     if (dados[m] == x)
6         return m;
7     else if (dados[m] < x)
8         return busca_binaria(dados, m + 1, r, x);
9     else
10        return busca_binaria(dados, l, m - 1, x);
11 }
12
13 int busca(p_vetor v, int x) {
14     return busca_binaria(v->dados, 0, v->n - 1, x);
15 }
```

A função **busca** é a que o cliente usará

- O cliente não precisa saber que usamos busca binária
- nem como chamá-la (não precisa estar no .h)

## Vetores Não Ordenados vs. Ordenados

	Não Ordenados	Ordenados
Inserção	$O(1)$	$O(n)$
Remoção	$O(1)$	$O(n)$
Busca	$O(n)$	$O(\lg n)$

Se temos muitas inserções e remoções e poucas buscas:

- Usamos vetores não ordenados
- Podemos considerar ordenar o vetor antes de buscar

Se temos poucas inserções e remoções e muitas buscas:

- Usamos vetores ordenados

E se as três operações forem frequentes?

- Existem estruturas de dados para as quais as três operações custam  $O(\lg n)$

# Vetores Dinâmicos

Os vetores que vimos até agora tem um grande problema — eles têm espaço limitado

Na hora de inicializar é necessário saber o tamanho máximo que o vetor terá durante o seu tempo de vida

- Isso nem sempre é possível
- Pode levar a um grande desperdício de memória

Uma opção é criar um vetor que aumenta e diminuí de tamanho de acordo com a quantidade de dados armazenada

# Mudança na struct

Vamos realizar uma mudança na **struct** que define o **vetor**

```
1 typedef struct vetor *p_vetor;  
2  
3 struct vetor {  
4     int *dados;  
5     int n;  
6     int alocado;  
7 };
```

O campo **alocado** indica com qual tamanho o vetor foi alocado

Porém, o campo **n** indica quantas posições estão de fato sendo usadas

# Alterações na Inicialização

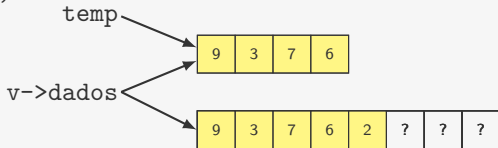
Basta armazenar no campo `alocado` qual o tamanho inicial

```
1 p_vetor criar_vetor(int tam) {
2     p_vetor v;
3     v = malloc(sizeof(struct vetor));
4     v->dados = malloc(tam * sizeof(int));
5     v->n = 0;
6     v->alocado = tam;
7     return v;
8 }
```

## Alterações na Inserção

Se, ao inserir o elemento, iremos estourar o vetor, dobramos o seu tamanho

```
1 void adicionar_elemento(p_vetor v, int x) {
2     int i, *temp;
3     if (v->n == v->alocado) {
4         temp = v->dados;
5         v->alocado *= 2;
6         v->dados = malloc(v->alocado * sizeof(int));
7         for (i = 0; i < v->n; i++)
8             v->dados[i] = temp[i];
9         free(temp);
10    }
11    v->dados[v->n] = x;
12    v->n++;
13 }
```



Tempo para inserir o  $i$ -ésimo elemento:

- $O(1)$  se não precisou aumentar o vetor
- $O(i)$  se precisou aumentar o vetor



## Tempo para inserir $n$ elementos

Inserir o  $i$ -ésimo elemento pode demorar tempo  $O(i)$

Então inserir  $n$  elementos demora tempo  $O(n^2)$

- custo **amortizado** por elemento  $O(n)$

Essa análise não é justa

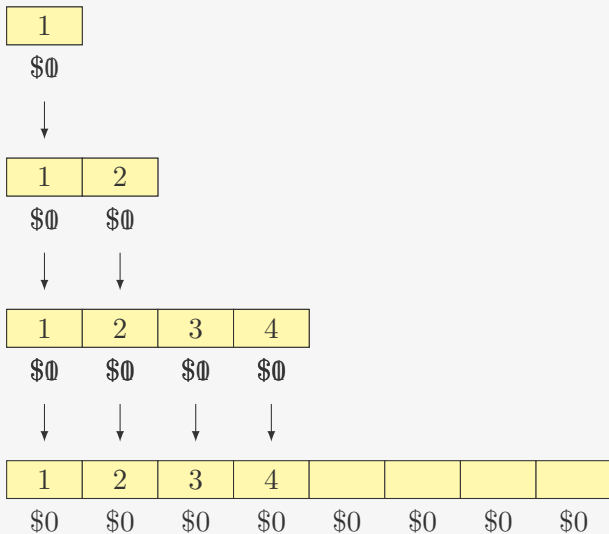
- Na verdade, o custo amortizado é  $O(1)$

Ao invés de “cobrar”  $i$  da  $i$ -ésima inserção, cobre **3** de  $i$ :

- **1** para pagar a inserção atual
- **1** para pagar a sua cópia para um novo vetor
- **1** para pagar a cópia de um outro elemento para um novo vetor

Dessa forma, nunca ficamos devendo!

# Simulação da contabilidade



# Alterações na Remoção

Para não desperdiçar espaço, diminuámos o vetor ao remover

Reduzimos o vetor pela metade quando ele estiver  $1/4$  cheio

- Melhor do que quando estiver  $1/2$  cheio — por que?

Custo amortizado de  $O(1)$

Implementação: Exercício

# Vetores Dinâmicos — Conclusão

Vetores dinâmicos:

- Inserção e Remoção em  $O(1)$  (amortizado)
- Desperdiçam no máximo  $3n$  de espaço

Úteis se você não sabe o tamanho do vetor

- mas pode trazer um *overhead* desnecessário

Algumas operações de inserção e remoção podem demorar muito (mas acontecem poucas vezes)

## Exercício — Busca Binária Iterativa

Implemente uma versão da busca binária que não utiliza recursão.

# Solução

```
1 int busca_binaria(int *dados, int l, int r, int x) {
2     int m = (l+r)/2;
3     while (l <= r) {
4         if (dados[m] == x)
5             return m;
6         else if (dados[m] < x)
7             l = m + 1;
8         else
9             r = m - 1;
10        m = (l+r)/2;
11    }
12    return -1;
13 }
```

## Exercício — Vetor Dinâmico Ordenado

Implemente a inserção de um vetor dinâmico ordenado.

- Quanto tempo pode demorar a inserção do  $i$ -ésimo elemento?
- Qual é o tempo amortizado da inserção?

# Solução

```
1 int busca_binaria(int *dados, int l, int r, int x) {
2     int m = (l+r)/2;
3     while (l <= r) {
4         if (dados[m] == x)
5             return m;
6         else if (dados[m] < x)
7             l = m + 1;
8         else
9             r = m - 1;
10        m = (l+r)/2;
11    }
12    return -1;
13 }
```

Quanto tempo pode demorar a inserção do  $i$ -ésimo elemento?

- Se precisar aumentar o vetor  $O(i)$
- Se não precisar aumentar o vetor, ainda é  $O(i)$

Qual é o tempo amortizado da inserção?

- $O(n)$



Dúvidas?