

# Estruturas de Dados

Vetores dinâmicos

## Aula 01

Prof. Felipe A. Louza



- 1 Vetores
- 2 Vetores ordenados
- 3 Vetores dinâmicos
- 4 Referências

1 Vetores

2 Vetores ordenados

3 Vetores dinâmicos

4 Referências

# Vetores

Vetores (ou *arrays*) são uma **forma nativa** em muitas **linguagens de programação** para **estruturar dados**

- Corresponde à uma lista **indexada** de itens
- O número de elementos é fixo e definido quando ele é criado
- Um vetor não aumenta ou diminui de tamanho

0	1	2	3	4	5	6	7	8	9
9	3	7	6	2	5	?	?	?	?

Vetores são usados para **implementar** muitas outras estruturas de dados, como filas, pilhas, grafos, etc.

Em C, um vetor pode ser alocado:

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

A sua grande vantagem é o acesso em tempo  $O(1)$ , a qualquer um dos seus elementos através do índice

`&v[i] = &v[0] + i*sizeof(int)`

0	1	2	3	4	5	6	7	8	9
9	3	7	6	2	5	?	?	?	?

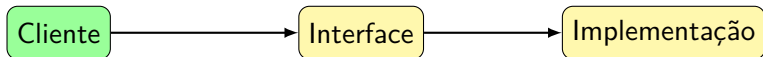
Vamos definir um **Tipo Abstrato de Dados** (TAD) para os nossos vetores:

- Podemos **abstrair a forma** como o vetor é implementado (alocação estática ou dinâmica)
- Vamos definir as **operações disponíveis** para o TAD
- Depois, “apenas” utilizamos o TAD (**cliente**)

Operações do TAD:

- **Criar**, Adicionar, Remover, Buscar e Acessar, **Destruir**

Relembrando:

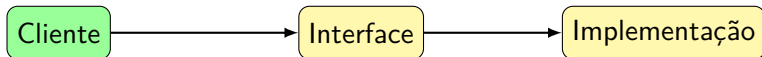


- **Interface:** conjunto de operações de um TAD
  - Consiste dos nomes e definições usadas para executar as operações.
- **Implementação:** conjunto de algoritmos que realizam as operações
  - A implementação é o **único “lugar”** que uma variável é acessada diretamente

---

A interface funciona como um contrato entre o cliente e a implementação.

Relembrando:



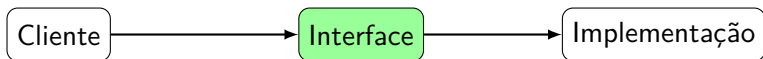
- **Interface**: conjunto de operações de um TAD
  - Consiste dos nomes e definições usadas para executar as operações.
- **Implementação**: conjunto de algoritmos que realizam as operações
  - A implementação é o **único “lugar”** que uma variável é acessada diretamente
- **Cliente**: código que utiliza/chama uma operação
  - O cliente **nunca** acessa a variável diretamente

---

A interface funciona como um contrato entre o cliente e a implementação.

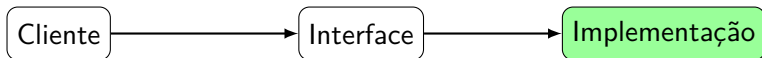


# Vetor



## vetor.h

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



## vetor.c

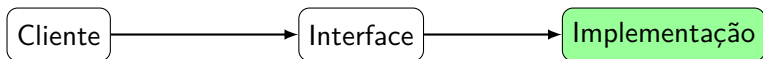
```
1  #include "vetor.h"
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  vetor* criar_vetor(int tam) {
6      ...
7  }
8
9  void destruir_vetor(vetor *v) {
10     ...
11 }
12
13 int acessa(vetor *v, int i) {
14     ...
15 }
```

```
1  int tamanho(vetor *v) {
2      ...
3  }
4
5  void adicionar_elemento(vetor *v, int x) {
6      ...
7  }
8
9  void remover_elemento(vetor *v, int x) {
10     ...
11 }
12
13 int busca(vetor *v, int x) {
14     ...
15 }
```



## vetor.c

```
5  vetor* criar_vetor(int tam) {
6      vetor *v;
7      v = (vetor*) malloc(sizeof(vetor));
8      v->dados = (int*) malloc(tam * sizeof(int));
9      v->n = 0;
10     return v;
11 }
12
13 void destruir_vetor(vetor *v) {
14     free(v->dados);
15     free(v);
16 }
```



`vetor.c`

```
35 int acessar(vetor *v, int i) {  
36     return v->dados[i];  
37 }  
38  
39 int tamanho(vetor *v) {  
40     return v->n;  
41 }
```

Veremos três implementações diferentes para **inserção/remoção/busca**:

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

# Vetor - Inserção

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

- inserimos no final do vetor

```
18 void adicionar_elemento(vetor *v, int x) {  
19     v->dados[v->n] = x;  
20     v->n++;  
21 }
```

Adicionando o valor 5:

0	1	2	3	4	5	6	7	8	9
9	3	7	6	2	?	?	?	?	?

---

Não estamos preocupados em manter nenhum tipo de ordem.

# Vetor - Inserção

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

- inserimos no final do vetor

```
18 void adicionar_elemento(vetor *v, int x) {  
19     v->dados[v->n] = x;  
20     v->n++;  
21 }
```

Adicionando o valor 5:

0	1	2	3	4	5	6	7	8	9
9	3	7	6	2	5	?	?	?	?

---

Não estamos preocupados em manter nenhum tipo de ordem.

# Vetor - Remoção

Remoção em  $O(1)$ : ??

Removendo elemento na posição 2:

0	1	2	3	4	5	6	7	8	9
9	3	7	6	2	5	?	?	?	?

---

Não estamos preocupados em manter nenhum tipo de ordem.

# Vetor - Remoção

Remoção em  $O(1)$ :

- trocamos o elemento a ser removido com o último

Removendo elemento na posição 2:

0	1	2	3	4	5	6	7	8	9
9	3	7	6	2	5	?	?	?	?

---

Não estamos preocupados em manter nenhum tipo de ordem.



# Vetor - Remoção

Remoção em  $O(1)$ :

- trocamos o elemento a ser removido com o último

Removendo elemento na posição 2:

0	1		3	4	5	6	7	8	9
9	3	7	6	2	5	?	?	?	?

---

Não estamos preocupados em manter nenhum tipo de ordem.

# Vetor - Remoção

Remoção em  $O(1)$ :

- trocamos o elemento a ser removido com o último

Removendo elemento na posição 2:

0	1		3	4	5	6	7	8	9
9	3	5	6	2	5	?	?	?	?

---

Não estamos preocupados em manter nenhum tipo de ordem.

# Vetor - Remoção

Remoção em  $O(1)$ :

- trocamos o elemento a ser removido com o último

Removendo elemento na posição 2:

0	1	2	3	4	5	6	7	8	9
9	3	5	6	2	5	?	?	?	?

---

Não estamos preocupados em manter nenhum tipo de ordem.

# Vetor - Remoção

Remoção em  $O(1)$ :

- trocamos o elemento a ser removido com o último

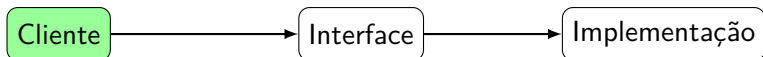
Removendo elemento na posição 2:

0	1	2	3	4	5	6	7	8	9
9	3	5	6	2	5	?	?	?	?

```
30 void remover_elemento(vetor *v, int i) {  
31     v->dados[i] = v->dados[v->n - 1];  
32     v->n--;  
33 }
```

---

Não estamos preocupados em manter nenhum tipo de ordem.



## exemplo1.c

```
1  #include <stdio.h>
2  #include "vetor.h"
3
4  int main() {
5      vetor *v;
6      v = criar_vetor(100);
7      int i, num;
8      scanf("%d", &num);
9      for(i=1; i<=num; i++) adicionar_elemento(v, i);
10     while(tamanho(v)!=0){
11         printf("%d -> ", acessar(v, 0));
12         remover_elemento(v, 0);
13     }
14     printf("FIM\n");
15     destruir_vetor(v);
16     return 0;
17 }
```

# Como compilar?

Teremos três arquivos diferentes:

- `exemplo1.c` contém a função `main`
- `vetor.c` contém a implementação
- `vetor.h` contém a interface

Vamos compilar por partes:

- `gcc -Wall -Werror -c exemplo1.c`
  - vai gerar o arquivo compilado `exemplo1.o`
- `gcc -Wall -Werror -c vetor.c`
  - vai gerar o arquivo compilado `vetor.o`
- `gcc exemplo1.o vetor.o -o exemplo1`
  - faz a linkagem, gerando o executável `exemplo1`

# Makefile

É mais fácil usar um **Makefile** para compilar

```
1 CFLAGS= -Wall -Werror
2
3 all: exemplo1
4
5 exemplo1: exemplo1.o vetor.o
6         gcc exemplo1.o vetor.o -o exemplo1
7
8 vetor.o: vetor.c vetor.h
9         gcc $(CFLAGS) -c vetor.c
10
11 exemplo1.o: exemplo1.c vetor.h
12         gcc $(CFLAGS) -c exemplo1.c
```

O comando **make** recompila apenas o que for necessário!

- Regra: o **nome do alvo** especifica o arquivo a ser gerado. As **dependencias** são os arquivos necessários como pré-requisitos<sup>1</sup>.

---

<sup>1</sup>O **make** irá procurar resolver dependencias executando outras regras.

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

0	1	2	3	4	5	6	7	8	9
9	3	7	6	2	5	?	?	?	?

```
23 int busca(vetor *v, int x) {  
24     int i;  
25     for (i = 0; i < v->n; i++)  
26         if (v->dados[i] == x) return i;  
27     return -1;  
28 }
```

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



Se as buscas no vetor forem  **muito frequentes**:

- Mais vantajoso realizar uma  **busca binária**
- Poderíamos ordenar o vetor antes e realizar a busca binárias

Porém, ordenar custa:

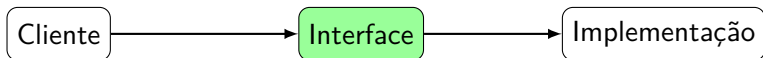
- $O(n^2)$  usando InsertionSort, SelectionSort ou BubbleSort
- $O(n \lg n)$  usando MergeSort ou QuickSort (tempo esperado)

Só vale a pena se não tivermos que ordenar sempre

Outra opção: podemos manter o vetor ordenado

- 1 Vetores
- 2 Vetores ordenados
- 3 Vetores dinâmicos
- 4 Referências

# Vetor Ordenado - Interface



Vamos (re)implementar as seguintes funções

`vetor_ordenado.h`

```
7 //Funções
8 vetor* criar_vetor(int tam);
9 void destruir_vetor(vetor *v);
10
11 void adicionar_elemento(vetor *v, int x);
12 void remover_elemento(vetor *v, int i);
13 int busca(vetor *v, int x);
14
15 int acessar(vetor *v, int i);
16 int tamanho(vetor *v);
```

# Vetor Ordenado - Inserção

Para adicionar um **novo elemento** precisamos:

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

Adicionando o valor **4**:

0	1	2	3	4	5	6	7	8	9
2	3	5	6	7	9	?	?	?	?

```
18 void adicionar_elemento(vetor *v, int x) {  
19     int i;  
20     for (i = v->n - 1; i >= 0 && v->dados[i] > x; i--)  
21         v->dados[i+1] = v->dados[i];  
22     v->dados[i+1] = x;  
23     v->n++;  
24 }
```

# Vetor Ordenado - Inserção

Para adicionar um **novo elemento** precisamos:

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

Adicionando o valor **4**:

0	1	2	3	4		6	7	8	9
2	3	5	6	7	9	?	?	?	?

```
18 void adicionar_elemento(vetor *v, int x) {  
19     int i;  
20     for (i = v->n - 1; i >= 0 && v->dados[i] > x; i--)  
21         v->dados[i+1] = v->dados[i];  
22     v->dados[i+1] = x;  
23     v->n++;  
24 }
```

# Vetor Ordenado - Inserção

Para adicionar um **novo elemento** precisamos:

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

Adicionando o valor **4**:

0	1	2	3	4		6	7	8	9
2	3	5	6	7	9	9	?	?	?

```
18 void adicionar_elemento(vetor *v, int x) {  
19     int i;  
20     for (i = v->n - 1; i >= 0 && v->dados[i] > x; i--)  
21         v->dados[i+1] = v->dados[i];  
22     v->dados[i+1] = x;  
23     v->n++;  
24 }
```

# Vetor Ordenado - Inserção

Para adicionar um **novo elemento** precisamos:

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

Adicionando o valor **4**:

0	1	2	3		5	6	7	8	9
2	3	5	6	7	9	9	?	?	?

```
18 void adicionar_elemento(vetor *v, int x) {  
19     int i;  
20     for (i = v->n - 1; i >= 0 && v->dados[i] > x; i--)  
21         v->dados[i+1] = v->dados[i];  
22     v->dados[i+1] = x;  
23     v->n++;  
24 }
```

# Vetor Ordenado - Inserção

Para adicionar um **novo elemento** precisamos:

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

Adicionando o valor **4**:

0	1	2	3		5	6	7	8	9
2	3	5	6	7	7	9	?	?	?

```
18 void adicionar_elemento(vetor *v, int x) {
19     int i;
20     for (i = v->n - 1; i >= 0 && v->dados[i] > x; i--)
21         v->dados[i+1] = v->dados[i];
22     v->dados[i+1] = x;
23     v->n++;
24 }
```



# Vetor Ordenado - Inserção

Para adicionar um **novo elemento** precisamos:

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

Adicionando o valor **4**:

0	1	2		4	5	6	7	8	9
2	3	5	6	7	7	9	?	?	?

```
18 void adicionar_elemento(vetor *v, int x) {
19     int i;
20     for (i = v->n - 1; i >= 0 && v->dados[i] > x; i--)
21         v->dados[i+1] = v->dados[i];
22     v->dados[i+1] = x;
23     v->n++;
24 }
```

# Vetor Ordenado - Inserção

Para adicionar um **novo elemento** precisamos:

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

Adicionando o valor **4**:

0	1	2		4	5	6	7	8	9
2	3	5	6	6	7	9	?	?	?

```
18 void adicionar_elemento(vetor *v, int x) {
19     int i;
20     for (i = v->n - 1; i >= 0 && v->dados[i] > x; i--)
21         v->dados[i+1] = v->dados[i];
22     v->dados[i+1] = x;
23     v->n++;
24 }
```

# Vetor Ordenado - Inserção

Para adicionar um **novo elemento** precisamos:

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

Adicionando o valor **4**:

0	1		3	4	5	6	7	8	9
2	3	5	6	6	7	9	?	?	?

```
18 void adicionar_elemento(vetor *v, int x) {  
19     int i;  
20     for (i = v->n - 1; i >= 0 && v->dados[i] > x; i--)  
21         v->dados[i+1] = v->dados[i];  
22     v->dados[i+1] = x;  
23     v->n++;  
24 }
```

# Vetor Ordenado - Inserção

Para adicionar um **novo elemento** precisamos:

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

Adicionando o valor **4**:

0	1		3	4	5	6	7	8	9
2	3	5	5	6	7	9	?	?	?

```
18 void adicionar_elemento(vetor *v, int x) {
19     int i;
20     for (i = v->n - 1; i >= 0 && v->dados[i] > x; i--)
21         v->dados[i+1] = v->dados[i];
22     v->dados[i+1] = x;
23     v->n++;
24 }
```

# Vetor Ordenado - Inserção

Para adicionar um **novo elemento** precisamos:

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

Adicionando o valor **4**:

0	1		3	4	5	6	7	8	9
2	3	4	5	6	7	9	?	?	?

```
18 void adicionar_elemento(vetor *v, int x) {  
19     int i;  
20     for (i = v->n - 1; i >= 0 && v->dados[i] > x; i--)  
21         v->dados[i+1] = v->dados[i];  
22     v->dados[i+1] = x;  
23     v->n++;  
24 }
```

# Vetor Ordenado - Inserção

Para adicionar um **novo elemento** precisamos:

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

Adicionando o valor **4**:

0	1	2	3	4	5	6	7	8	9
2	3	4	5	6	7	9	?	?	?

```
18 void adicionar_elemento(vetor *v, int x) {  
19     int i;  
20     for (i = v->n - 1; i >= 0 && v->dados[i] > x; i--)  
21         v->dados[i+1] = v->dados[i];  
22     v->dados[i+1] = x;  
23     v->n++;  
24 }
```

# Vetor Ordenado - Inserção

Para adicionar um **novo elemento** precisamos:

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

Adicionando o valor **4**:

0	1	2	3	4	5	6	7	8	9
2	3	4	5	6	7	9	?	?	?

```
18 void adicionar_elemento(vetor *v, int x) {  
19     int i;  
20     for (i = v->n - 1; i >= 0 && v->dados[i] > x; i--)  
21         v->dados[i+1] = v->dados[i];  
22     v->dados[i+1] = x;  
23     v->n++;  
24 }
```

# Vetor Ordenado - Inserção

Para adicionar um **novo elemento** precisamos:

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

Adicionando o valor **4**:

0	1	2	3	4	5	6	7	8	9
2	3	4	5	6	7	9	?	?	?

```
18 void adicionar_elemento(vetor *v, int x) {  
19     int i;  
20     for (i = v->n - 1; i >= 0 && v->dados[i] > x; i--)  
21         v->dados[i+1] = v->dados[i];  
22     v->dados[i+1] = x;  
23     v->n++;  
24 }
```

Tempo:  $O(n)$



# Vetor Ordenado - Remoção

Para remover um elemento precisamos:

- 1 deslocar os elementos para a esquerda

Removendo elemento na posição 3:

0	1	2	3	4	5	6	7	8	9
2	3	4	5	6	7	9	?	?	?

```
26 void remover_elemento(vetor *v, int i) {  
27     for(; i < v->n - 1; i++)  
28         v->dados[i] = v->dados[i+1];  
29     (v->n)--;  
30 }
```

# Vetor Ordenado - Remoção

Para remover um elemento precisamos:

- 1 deslocar os elementos para a esquerda

Removendo elemento na posição 3:

0	1	2		4	5	6	7	8	9
2	3	4	5	6	7	9	?	?	?

```
26 void remover_elemento(vetor *v, int i) {  
27     for(; i < v->n - 1; i++)  
28         v->dados[i] = v->dados[i+1];  
29     (v->n)--;  
30 }
```

# Vetor Ordenado - Remoção

Para remover um elemento precisamos:

- 1 deslocar os elementos para a esquerda

Removendo elemento na posição 3:

0	1	2		4	5	6	7	8	9
2	3	4	6	6	7	9	?	?	?

```
26 void remover_elemento(vetor *v, int i) {  
27     for(; i < v->n - 1; i++)  
28         v->dados[i] = v->dados[i+1];  
29     (v->n)--;  
30 }
```

# Vetor Ordenado - Remoção

Para remover um elemento precisamos:

- 1 deslocar os elementos para a esquerda

Removendo elemento na posição 3:

0	1	2	3		5	6	7	8	9
2	3	4	6	6	7	9	?	?	?

```
26 void remover_elemento(vetor *v, int i) {  
27     for(; i < v->n - 1; i++)  
28         v->dados[i] = v->dados[i+1];  
29     (v->n)--;  
30 }
```

# Vetor Ordenado - Remoção

Para remover um elemento precisamos:

- 1 deslocar os elementos para a esquerda

Removendo elemento na posição 3:

0	1	2	3		5	6	7	8	9
2	3	4	6	7	7	9	?	?	?

```
26 void remover_elemento(vetor *v, int i) {  
27     for(; i < v->n - 1; i++)  
28         v->dados[i] = v->dados[i+1];  
29     (v->n)--;  
30 }
```

# Vetor Ordenado - Remoção

Para remover um elemento precisamos:

- 1 deslocar os elementos para a esquerda

Removendo elemento na posição 3:

0	1	2	3	4		6	7	8	9
2	3	4	6	7	7	9	?	?	?

```
26 void remover_elemento(vetor *v, int i) {  
27     for(; i < v->n - 1; i++)  
28         v->dados[i] = v->dados[i+1];  
29     (v->n)--;  
30 }
```

# Vetor Ordenado - Remoção

Para remover um elemento precisamos:

- 1 deslocar os elementos para a esquerda

Removendo elemento na posição 3:

0	1	2	3	4		6	7	8	9
2	3	4	6	7	9	9	?	?	?

```
26 void remover_elemento(vetor *v, int i) {  
27     for(; i < v->n - 1; i++)  
28         v->dados[i] = v->dados[i+1];  
29     (v->n)--;  
30 }
```

# Vetor Ordenado - Remoção

Para remover um elemento precisamos:

- 1 deslocar os elementos para a esquerda

Removendo elemento na posição 3:

0	1	2	3	4	5	6	7	8	9
2	3	4	6	7	9	9	?	?	?

```
26 void remover_elemento(vetor *v, int i) {  
27     for(; i < v->n - 1; i++)  
28         v->dados[i] = v->dados[i+1];  
29     (v->n)--;  
30 }
```



# Vetor Ordenado - Remoção

Para remover um elemento precisamos:

- 1 deslocar os elementos para a esquerda

Removendo elemento na posição 3:

0	1	2	3	4	5	6	7	8	9
2	3	4	6	7	9	9	?	?	?

```
26 void remover_elemento(vetor *v, int i) {  
27     for(; i < v->n - 1; i++)  
28         v->dados[i] = v->dados[i+1];  
29     (v->n)--;  
30 }
```

Tempo:  $O(n)$

# Vetor Ordenado - Busca

Agora podemos fazer buscas em  $O(\lg n)$

```
32 int busca_binaria(int *dados, int l, int r, int x) {  
33     int m = (l+r)/2;  
34     if (l > r) return -1;  
35     if (dados[m] == x) return m;  
36     else if (dados[m] < x) return busca_binaria(dados, m + 1, r, x);  
37     else return busca_binaria(dados, l, m - 1, x);  
38 }  
39  
40 int busca(vetor *v, int x) {  
41     return busca_binaria(v->dados, 0, v->n - 1, x);  
42 }
```

# Vetor Ordenado - Busca

Agora podemos fazer buscas em  $O(\lg n)$

```
32 int busca_binaria(int *dados, int l, int r, int x) {  
33     int m = (l+r)/2;  
34     if (l > r) return -1;  
35     if (dados[m] == x) return m;  
36     else if (dados[m] < x) return busca_binaria(dados, m + 1, r, x);  
37     else return busca_binaria(dados, l, m - 1, x);  
38 }  
39  
40 int busca(vetor *v, int x) {  
41     return busca_binaria(v->dados, 0, v->n - 1, x);  
42 }
```

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

- O cliente não precisa saber que usamos busca binária ←encapsulamento
- nem como chamá-la

# 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

Se temos poucas inserções e remoções e muitas buscas:

- Usamos vetores ordenados

E se as três operações forem frequentes?

- Existem outras EDs para as quais as três operações custam  $O(\lg n)$

- 1 Vetores
- 2 Vetores ordenados
- 3 Vetores dinâmicos**
- 4 Referências

Os **principal problema** dos vetores é o seu espaço limitado.

Quando **inicializamos** é necessário saber o **tamanho máximo** que o vetor pode ter durante o seu tempo de vida

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

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

# Vetores Dinâmicos - Interface

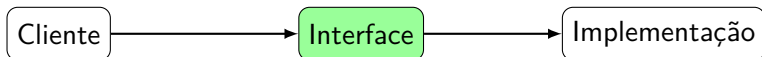
Vamos realizar uma mudança na `struct` que define o Vetor:

`vetor_dinamico.h`

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

- O campo `alocado` indica com qual tamanho o vetor foi alocado
- Enquanto que o campo `n` indica quantas posições estão de fato sendo usadas

# Vetores Dinâmicos - Interface



Vamos (re)implementar as seguintes funções

`vetor_dinamico.h`

```
8 //Funções
9 vetor *criar_vetor(int tam);
10 void destruir_vetor(vetor *v);
11
12 void adicionar_elemento(vetor *v, int x);
13 void remover_elemento(vetor *v, int i);
14 int busca(vetor *v, int x);
15
16 int acessar(vetor *v, int i);
17 int tamanho(vetor *v);
18 int alocado(vetor *v);
```



# Vetores Dinâmicos - Criação

Na função `cria_vetor()` armazenamos no campo `alocado` qual o tamanho inicial

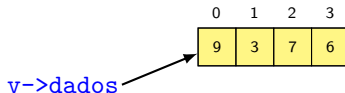
`vetor.c`

```
5  vetor* cria_vetor(int tam) {  
6      vetor *v;  
7      v = (vetor*) malloc(sizeof(vetor));  
8      v->dados = (int*) malloc(tam * sizeof(int));  
9      v->n = 0;  
10     v->alocado = tam;  
11     return v;  
12 }
```

# Vetores Dinâmicos - Inserção

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

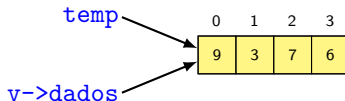
```
19 void adicionar_elemento(vetor *v, int x) {  
20     int i, *temp;  
21     if (v->n == v->alocado) {  
22         temp = v->dados;  
23         v->alocado *= 2;  
24         v->dados = (int*) malloc(v->alocado * sizeof(int));  
25         for (i = 0; i < v->n; i++)  
26             v->dados[i] = temp[i];  
27         free(temp);  
28     }  
29     v->dados[v->n] = x;  
30     v->n++;  
31 }
```



# Vetores Dinâmicos - Inserção

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

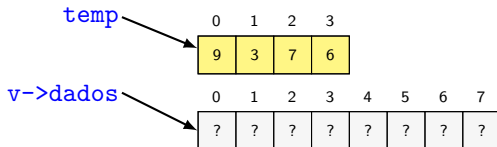
```
19 void adicionar_elemento(vetor *v, int x) {  
20     int i, *temp;  
21     if (v->n == v->alocado) {  
22         temp = v->dados;  
23         v->alocado *= 2;  
24         v->dados = (int*) malloc(v->alocado * sizeof(int));  
25         for (i = 0; i < v->n; i++)  
26             v->dados[i] = temp[i];  
27         free(temp);  
28     }  
29     v->dados[v->n] = x;  
30     v->n++;  
31 }
```



# Vetores Dinâmicos - Inserção

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

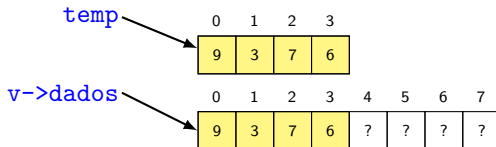
```
19 void adicionar_elemento(vetor *v, int x) {
20     int i, *temp;
21     if (v->n == v->alocado) {
22         temp = v->dados;
23         v->alocado *= 2;
24         v->dados = (int*) malloc(v->alocado * sizeof(int));
25         for (i = 0; i < v->n; i++)
26             v->dados[i] = temp[i];
27         free(temp);
28     }
29     v->dados[v->n] = x;
30     v->n++;
31 }
```



# Vetores Dinâmicos - Inserção

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

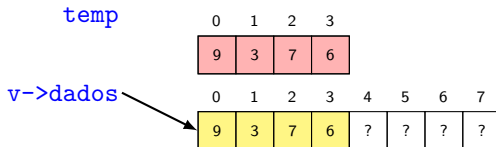
```
19 void adicionar_elemento(vetor *v, int x) {  
20     int i, *temp;  
21     if (v->n == v->alocado) {  
22         temp = v->dados;  
23         v->alocado *= 2;  
24         v->dados = (int*) malloc(v->alocado * sizeof(int));  
25         for (i = 0; i < v->n; i++)  
26             v->dados[i] = temp[i];  
27         free(temp);  
28     }  
29     v->dados[v->n] = x;  
30     v->n++;  
31 }
```



# Vetores Dinâmicos - Inserção

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

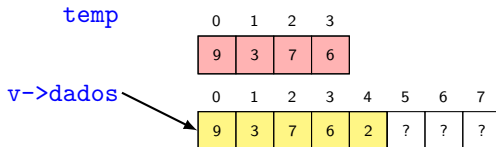
```
19 void adicionar_elemento(vetor *v, int x) {  
20     int i, *temp;  
21     if (v->n == v->alocado) {  
22         temp = v->dados;  
23         v->alocado *= 2;  
24         v->dados = (int*) malloc(v->alocado * sizeof(int));  
25         for (i = 0; i < v->n; i++)  
26             v->dados[i] = temp[i];  
27         free(temp);  
28     }  
29     v->dados[v->n] = x;  
30     v->n++;  
31 }
```



# Vetores Dinâmicos - Inserção

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

```
19 void adicionar_elemento(vetor *v, int x) {  
20     int i, *temp;  
21     if (v->n == v->alocado) {  
22         temp = v->dados;  
23         v->alocado *= 2;  
24         v->dados = (int*) malloc(v->alocado * sizeof(int));  
25         for (i = 0; i < v->n; i++)  
26             v->dados[i] = temp[i];  
27         free(temp);  
28     }  
29     v->dados[v->n] = x;  
30     v->n++;  
31 }
```



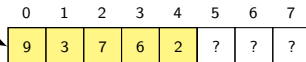
# Vetores Dinâmicos - Inserção

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

```
19 void adicionar_elemento(vetor *v, int x) {  
20     int i, *temp;  
21     if (v->n == v->alocado) {  
22         temp = v->dados;  
23         v->alocado *= 2;  
24         v->dados = (int*) malloc(v->alocado * sizeof(int));  
25         for (i = 0; i < v->n; i++)  
26             v->dados[i] = temp[i];  
27         free(temp);  
28     }  
29     v->dados[v->n] = x;  
30     v->n++;  
31 }
```

temp

v->dados



0	1	2	3	4	5	6	7
9	3	7	6	2	?	?	?

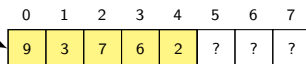


# Vetores Dinâmicos - Inserção

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

```
19 void adicionar_elemento(vetor *v, int x) {  
20     int i, *temp;  
21     if (v->n == v->alocado) {  
22         temp = v->dados;  
23         v->alocado *= 2;  
24         v->dados = (int*) malloc(v->alocado * sizeof(int));  
25         for (i = 0; i < v->n; i++)  
26             v->dados[i] = temp[i];  
27         free(temp);  
28     }  
29     v->dados[v->n] = x;  
30     v->n++;  
31 }
```

**v->dados**



0	1	2	3	4	5	6	7
9	3	7	6	2	?	?	?

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 (no pior caso) pode demorar tempo  $O(i)$

- Então inserir  $n$  elementos demora tempo  $\sum_{i=1}^n i = O(n^2)$ ?

Essa análise não é justa (nem realista)

- Na verdade, precisamos de uma análise de custo amortizado

# Tempo para inserir $n$ elementos

Inserir o  $i$ -ésimo elemento (no pior caso) pode demorar tempo  $O(i)$

- Então inserir  $n$  elementos demora tempo  $\sum_{i=1}^n i = O(n^2)$ ?

Essa análise não é justa (nem realista)

- Na verdade, precisamos de uma análise de custo amortizado

## Análise amortizada:

- Quando o tempo de cada execução depende das anteriores
- Tipicamente, cada execução lenta é precedida por muitas execuções rápidas
- Não confundida com custo médio (conjunto de execuções independentes)

# Tempo para inserir $n$ elementos

A política de redimensionamento tem **impacto no desempenho**

- Na prática, o tamanho do vetor **cresce muito rápido**:  
 $1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2^{11}, 2^{12}, 2^{13}, 2^{14}, 2^{15}, \dots$
- Com isso, faremos **poucas operações** de redimensionamento
- Para inserir  $n$  elementos  $\approx \lceil \lg(n) \rceil$  **redimensionamentos**

---

$2^{\lfloor \lg(n) \rfloor}$  é a maior potência de 2 menor do que  $n$ .

# Tempo para inserir $n$ elementos

A política de redimensionamento tem **impacto no desempenho**

- Na prática, o tamanho do vetor **cresce muito rápido**:  
 $1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2^{11}, 2^{12}, 2^{13}, 2^{14}, 2^{15}, \dots$
- Com isso, faremos **poucas operações** de redimensionamento
- Para inserir  $n$  elementos  $\approx \lceil \lg(n) \rceil$  **redimensionamentos**

Número **total de cópias**:

- $1 + 2 + 4 + 8 + 16 + 32 + \dots + 2^{\lfloor \lg(n) \rfloor} =$

---

$2^{\lfloor \lg(n) \rfloor}$  é a maior potência de 2 menor do que  $n$ .

# Tempo para inserir $n$ elementos

A política de redimensionamento tem **impacto no desempenho**

- Na prática, o tamanho do vetor **cresce muito rápido**:  
 $1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2^{11}, 2^{12}, 2^{13}, 2^{14}, 2^{15}, \dots$
- Com isso, faremos **poucas operações** de redimensionamento
- Para inserir  $n$  elementos  $\approx \lceil \lg(n) \rceil$  **redimensionamentos**

Número **total de cópias**:

$$1 + 2 + 4 + 8 + 16 + 32 + \dots + 2^{\lfloor \lg(n) \rfloor} = \underbrace{2^{\lceil \lg(n) \rceil}}_{c \cdot n} - 1 = O(n)$$

---

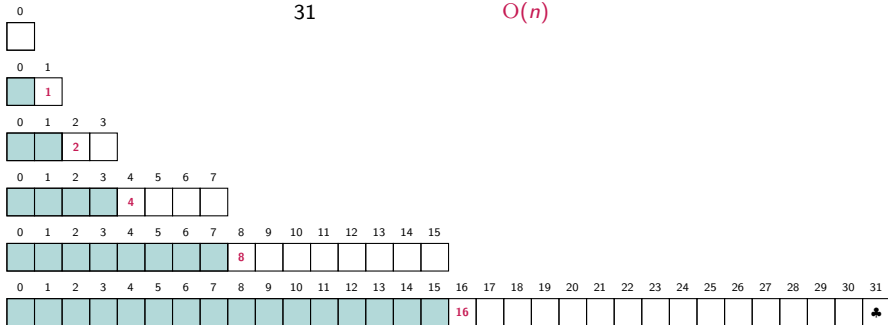
$2^{\lfloor \lg(n) \rfloor}$  é a maior potência de 2 menor do que  $n$ .

## Tempo para inserir $n$ elementos

Exemplo:  $n = 17$

$$\lceil \lg(17) \rceil = 5 \text{ redimensionamentos}$$

$$\underbrace{1 + 2 + 4 + 8 + 16}_{31} = \underbrace{2^{\lceil \lg(17) \rceil}}_{O(n)} - 1 \text{ cópias}$$



# Tempo para inserir $n$ elementos

Vamos assumir que o **último elemento inserido** causou um redimensionamento

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
																16															♣

Quantas operações foram executadas?

- $n$  operações de escrita:  $v \rightarrow \text{dados}[v \rightarrow n] = x;$
- $n$  cópias do último redimensionamento
- $n - 1$  cópias cópias anteriores:

$$1 + 2 + \cdots + \frac{n}{4} + \frac{n}{2} = n - 1$$

- Menos de  $3n$ , no total!

$$\frac{3n}{n} \approx \underline{O(1) \text{ por elemento}}$$



# Tempo para inserir $n$ elementos

Então, a **sobrecarga** para redimensionar causa apenas um **aumento constante (pequeno)** no número de operações por inserção

- O que é **bastante eficiente**!

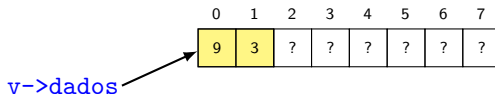
Essas contas valem para uma política de redimensionamento como PG de razão 2

- Para outras políticas, as contas e a conclusão sobre o desempenho podem ser diferentes

# Vetores Dinâmicos - Remoção

Para não desperdiçar espaço, diminuimos o vetor ao remover

```
33 void remover_elemento(vetor *v, int i) {
34     int j, *temp;
35     for(; i < v->n - 1; i++)
36         v->dados[i] = v->dados[i+1];
37     (v->n)--;
38     if (v->n < v->alocado/4 && v->alocado >= 4) {
39         temp = v->dados;
40         v->alocado /= 2;
41         v->dados = malloc(v->alocado * sizeof(int));
42         for (j = 0; j < v->n; j++)
43             v->dados[j] = temp[j];
44         free(temp);
45     }
46 }
47
```

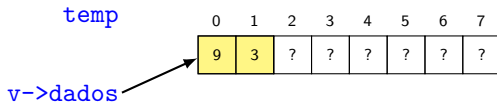


Reduzimos o vetor pela metade quando ele estiver  $1/4$  cheio

# Vetores Dinâmicos - Remoção

Para não desperdiçar espaço, diminuimos o vetor ao remover

```
33 void remover_elemento(vetor *v, int i) {
34     int j, *temp;
35     for(; i < v->n - 1; i++)
36         v->dados[i] = v->dados[i+1];
37     (v->n)--;
38     if (v->n < v->alocado/4 && v->alocado >= 4) {
39         temp = v->dados;
40         v->alocado /= 2;
41         v->dados = malloc(v->alocado * sizeof(int));
42         for (j = 0; j < v->n; j++)
43             v->dados[j] = temp[j];
44         free(temp);
45     }
46 }
47
```

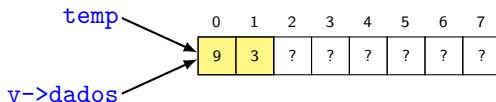


Reduzimos o vetor pela metade quando ele estiver  $1/4$  cheio

# Vetores Dinâmicos - Remoção

Para não desperdiçar espaço, diminuimos o vetor ao remover

```
33 void remover_elemento(vetor *v, int i) {  
34     int j, *temp;  
35     for(; i < v->n - 1; i++)  
36         v->dados[i] = v->dados[i+1];  
37     (v->n)--;  
38     if (v->n < v->alocado/4 && v->alocado >= 4) {  
39         temp = v->dados;  
40         v->alocado /= 2;  
41         v->dados = malloc(v->alocado * sizeof(int));  
42         for (j = 0; j < v->n; j++)  
43             v->dados[j] = temp[j];  
44         free(temp);  
45     }  
46 }  
47
```

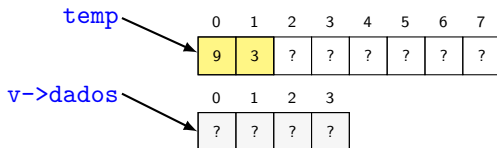


Reduzimos o vetor pela metade quando ele estiver 1/4 cheio

# Vetores Dinâmicos - Remoção

Para não desperdiçar espaço, diminuimos o vetor ao remover

```
33 void remover_elemento(vetor *v, int i) {
34     int j, *temp;
35     for(; i < v->n - 1; i++)
36         v->dados[i] = v->dados[i+1];
37     (v->n)--;
38     if (v->n < v->alocado/4 && v->alocado >= 4) {
39         temp = v->dados;
40         v->alocado /= 2;
41         v->dados = malloc(v->alocado * sizeof(int));
42         for (j = 0; j < v->n; j++)
43             v->dados[j] = temp[j];
44         free(temp);
45     }
46 }
47
```

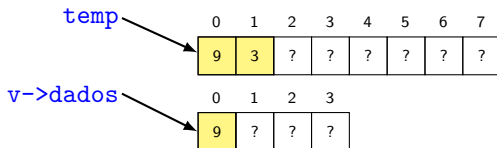


Reduzimos o vetor pela metade quando ele estiver  $1/4$  cheio

# Vetores Dinâmicos - Remoção

Para não desperdiçar espaço, diminuimos o vetor ao remover

```
33 void remover_elemento(vetor *v, int i) {  
34     int j, *temp;  
35     for(; i < v->n - 1; i++)  
36         v->dados[i] = v->dados[i+1];  
37     (v->n)--;  
38     if (v->n < v->alocado/4 && v->alocado >= 4) {  
39         temp = v->dados;  
40         v->alocado /= 2;  
41         v->dados = malloc(v->alocado * sizeof(int));  
42         for (j = 0; j < v->n; j++)  
43             v->dados[j] = temp[j];  
44         free(temp);  
45     }  
46 }  
47
```

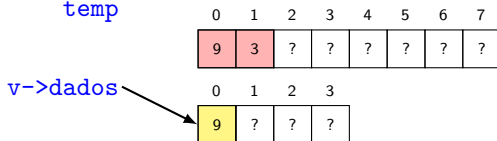


Reduzimos o vetor pela metade quando ele estiver  $1/4$  cheio

# Vetores Dinâmicos - Remoção

Para não desperdiçar espaço, diminuimos o vetor ao remover

```
33 void remover_elemento(vetor *v, int i) {
34     int j, *temp;
35     for(; i < v->n - 1; i++)
36         v->dados[i] = v->dados[i+1];
37     (v->n)--;
38     if (v->n < v->alocado/4 && v->alocado >= 4) {
39         temp = v->dados;
40         v->alocado /= 2;
41         v->dados = malloc(v->alocado * sizeof(int));
42         for (j = 0; j < v->n; j++)
43             v->dados[j] = temp[j];
44         free(temp);
45     }
46 }
47
```



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

- Desperdício máximo:  $3n$  de espaço

# Vetores Dinâmicos - Remoção

Para não desperdiçar espaço, diminuimos o vetor ao remover

```
33 void remover_elemento(vetor *v, int i) {  
34     int j, *temp;  
35     for(; i < v->n - 1; i++)  
36         v->dados[i] = v->dados[i+1];  
37     (v->n)--;  
38     if (v->n < v->alocado/4 && v->alocado >= 4) {  
39         temp = v->dados;  
40         v->alocado /= 2;  
41         v->dados = malloc(v->alocado * sizeof(int));  
42         for (j = 0; j < v->n; j++)           temp  
43             v->dados[j] = temp[j];  
44         free(temp);  
45     }  
46 }  
47
```



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

- Desperdício máximo:  $3n$  de espaço
- Custo amortizado de  $O(1)$



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)

Dúvidas?

- 1 Vetores
- 2 Vetores ordenados
- 3 Vetores dinâmicos
- 4 Referências**

- ① Materiais adaptados dos slides do Prof. Rafael C. S. Schouery, da Universidade Estadual de Campinas.
- ② Feofiloff, Paulo. Algoritmos em linguagem C. Elsevier Brasil, 2009.