MC-202 — Unidade 4
Vetores

Rafael C. S. Schouery
rafael@ic.unicamp.br

Universidade Estadual de Campinas

1º semestre/2017

Vetores

Vetores são uma forma nativa do C de estruturar dados

Vetores

Vetores são uma forma nativa do C de estruturar dados

- É uma lista indexada de itens

Vetores

Vetores são uma forma nativa do C de estruturar dados

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

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 uma bloco sequencial de memória

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 uma bloco sequencial de memória

Ele pode ser alocado:

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 uma bloco sequencial de memória

Ele pode ser alocado:

- estaticamente - `int v[100];`

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 uma bloco sequencial de memória

Ele pode ser alocado:

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

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 uma 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 #ifndef VETOR_H
2 #define VETOR_H
```

TAD Vetor - Interface

```
1 #ifndef VETOR_H
2 #define VETOR_H
3
4 typedef struct {
5   int *dados;
6   int n;
7 } vetor;
```

TAD Vetor - Interface

```
1 #ifndef VETOR_H
2 #define VETOR_H
3
4 typedef struct {
5   int *dados;
6   int n;
7 } vetor;
8
9 void iniciar_vetor(vetor **v, int tam);
```

TAD Vetor - Interface

```
 1 #ifndef VETOR_H
 2 #define VETOR_H
 3
 4 typedef struct {
 5   int *dados;
 6   int n;
 7 } vetor;
 8
 9 void iniciar_vetor(vetor **v, int tam);
10
11 void destruir_vetor(vetor **v);
```

TAD Vetor - Interface

```
 1 #ifndef VETOR_H
 2 #define VETOR_H
 3
 4 typedef struct {
 5   int *dados;
 6   int n;
 7 } vetor;
 8
 9 void iniciar_vetor(vetor **v, int tam);
10
11 void destruir_vetor(vetor **v);
12
13 void adicionar_elemento(vetor *v, int x);
```

TAD Vetor - Interface

```
 1 #ifndef VETOR_H
 2 #define VETOR_H
 3
 4 typedef struct {
 5   int *dados;
 6   int n;
 7 } vetor;
 8
 9 void iniciar_vetor(vetor **v, int tam);
10
11 void destruir_vetor(vetor **v);
12
13 void adicionar_elemento(vetor *v, int x);
14
15 void remover_elemento(vetor *v, int x);
```

TAD Vetor - Interface

```
 1 #ifndef VETOR_H
 2 #define VETOR_H
 3
 4 typedef struct {
 5   int *dados;
 6   int n;
 7 } vetor;
 8
 9 void iniciar_vetor(vetor **v, int tam);
10
11 void destruir_vetor(vetor **v);
12
13 void adicionar_elemento(vetor *v, int x);
14
15 void remover_elemento(vetor *v, int x);
16
17 int busca(vetor *v, int x);
```

TAD Vetor - Interface

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

TAD Vetor - Interface

```
 1 #ifndef VETOR_H
 2 #define VETOR_H
 3
 4 typedef struct {
 5   int *dados;
 6   int n;
 7 } vetor;
 8
 9 void iniciar_vetor(vetor **v, int tam);
10
11 void destruir_vetor(vetor **v);
12
13 void adicionar_elemento(vetor *v, int x);
14
15 void remover_elemento(vetor *v, int x);
16
17 int busca(vetor *v, int x);
18
19 void imprime(vetor *v);
20
21 #endif
```

TAD Vetor - Implementação

```c
#include "vetor.h"
#include <stdio.h>
#include <stdlib.h>

void iniciar_vetor(vetor **v, int tam) {
  ...
}

void destruir_vetor(vetor **v) {
  ...
}

void adicionar_elemento(vetor *v, int x) {
  ...
}

int busca(vetor *v, int x) {
  ...
}

void remover_elemento(vetor *v, int i) {
  ...
}

void imprime(vetor *v) {
  ...
}
```

TAD Vetor - Implementação

```c
#include "vetor.h"
#include <stdio.h>
#include <stdlib.h>

void iniciar_vetor(vetor **v, int tam) {
  ...
}

void destruir_vetor(vetor **v) {
  ...
}

void adicionar_elemento(vetor *v, int x) {
  ...
}

int busca(vetor *v, int x) {
  ...
}

void remover_elemento(vetor *v, int i) {
  ...
}

void imprime(vetor *v) {
  ...
}
```

Veremos três implementações diferentes

TAD Vetor - Implementação

```c
#include "vetor.h"
#include <stdio.h>
#include <stdlib.h>

void iniciar_vetor(vetor **v, int tam) {
  ...
}

void destruir_vetor(vetor **v) {
  ...
}

void adicionar_elemento(vetor *v, int x) {
  ...
}

int busca(vetor *v, int x) {
  ...
}

void remover_elemento(vetor *v, int i) {
  ...
}

void imprime(vetor *v) {
  ...
}
```

Veremos três implementações diferentes

- as três fazem as mesmas coisas

TAD Vetor - Implementação

```c
#include "vetor.h"
#include <stdio.h>
#include <stdlib.h>

void iniciar_vetor(vetor **v, int tam) {
  ...
}

void destruir_vetor(vetor **v) {
  ...
}

void adicionar_elemento(vetor *v, int x) {
  ...
}

int busca(vetor *v, int x) {
  ...
}

void remover_elemento(vetor *v, int i) {
  ...
}

void imprime(vetor *v) {
  ...
}
```

Veremos três implementações diferentes

- as três fazem as mesmas coisas
- mas levam tempo diferente

Inicialização/Destruição

Código no cliente:

Inicialização/Destruição

Código no cliente:

```
1   vetor *v;
2   iniciar_vetor(&v, 100);
```

Inicialização/Destruição

Código no cliente:

```
1   vetor *v;
2   iniciar_vetor(&v, 100);
```

Código em vetor.c:

Inicialização/Destruição

Código no cliente:

```
1   vetor *v;
2   iniciar_vetor(&v, 100);
```

Código em vetor.c:

```
1 void iniciar_vetor(vetor **v, int tam) {
2   *v = malloc(sizeof(vetor));
3   (*v)->dados = malloc(tam * sizeof(int));
4   (*v)->n = 0;
5 }
```

Inicialização/Destruição

Código no cliente:

```
1   vetor *v;
2   iniciar_vetor(&v, 100);
```

Código em vetor.c:

```
1 void iniciar_vetor(vetor **v, int tam) {
2   *v = malloc(sizeof(vetor));
3   (*v)->dados = malloc(tam * sizeof(int));
4   (*v)->n = 0;
5 }
```

Código no cliente:

Inicialização/Destruição

Código no cliente:

```
1   vetor *v;
2   iniciar_vetor(&v, 100);
```

Código em vetor.c:

```
1 void iniciar_vetor(vetor **v, int tam) {
2   *v = malloc(sizeof(vetor));
3   (*v)->dados = malloc(tam * sizeof(int));
4   (*v)->n = 0;
5 }
```

Código no cliente:

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

Inicialização/Destruição

Código no cliente:

```
1   vetor *v;
2   iniciar_vetor(&v, 100);
```

Código em vetor.c:

```
1 void iniciar_vetor(vetor **v, int tam) {
2   *v = malloc(sizeof(vetor));
3   (*v)->dados = malloc(tam * sizeof(int));
4   (*v)->n = 0;
5 }
```

Código no cliente:

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

Código em vetor.c:

Inicialização/Destruição

Código no cliente:

```
1   vetor *v;
2   iniciar_vetor(&v, 100);
```

Código em vetor.c:

```
1 void iniciar_vetor(vetor **v, int tam) {
2   *v = malloc(sizeof(vetor));
3   (*v)->dados = malloc(tam * sizeof(int));
4   (*v)->n = 0;
5 }
```

Código no cliente:

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

Código em vetor.c:

```
1 void destruir_vetor(vetor **v) {
2   free((*v)->dados);
3   free(*v);
4   *v = NULL;
5 }
```

Inserção e Remoção

Inserção em $\mathrm{O}(1)$ (tempo constante):

Inserção e Remoção

Inserção em $\mathrm{O}(1)$ (tempo constante):

- inserimos no final do vetor

Inserção e Remoção

Inserção em $O(1)$ (tempo constante):

- inserimos no final do vetor

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

Inserção e Remoção

Inserção em $O(1)$ (tempo constante):

- inserimos no final do vetor

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

Remoção em $O(1)$:

Inserção e Remoção

Inserção em $O(1)$ (tempo constante):

- inserimos no final do vetor

```
1 void adicionar_elemento(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

Inserção e Remoção

Inserção em $O(1)$ (tempo constante):

- inserimos no final do vetor

```
1 void adicionar_elemento(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
- diminuímos n

Inserção e Remoção

Inserção em $O(1)$ (tempo constante):

- inserimos no final do vetor

```
1 void adicionar_elemento(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
- diminuímos n

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

Busca

Busca sequencial em $\mathrm{O}(n)$ (linear)

Busca

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

```
1 int busca(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 }
```

Busca

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

```
1 int busca(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:

Buscas frequentes

Se as buscas no vetor forem muito frequentes:

- mais vantajoso realizar uma busca binária

Buscas frequentes

Se as buscas no vetor forem muito frequentes:

- mais vantajoso realizar uma busca binária

Poderíamos ordenar o vetor antes e realizar a busca binárias

Buscas frequentes

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:

Buscas frequentes

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:

- $\mathrm{O}(n^2)$ usando InsertionSort, SelectionSort ou BubbleSort

Buscas frequentes

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:

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

Buscas frequentes

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:

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

Só vale a pena se não tivermos que ordenar sempre

Buscas frequentes

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:

- $\mathrm{O}(n^2)$ usando InsertionSort, SelectionSort ou BubbleSort
- $\mathrm{O}(n \lg n)$ 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 e Remoção - Vetor Ordenado

Para adicionar um elemento precisamos:

Inserção e Remoção - Vetor Ordenado

Para adicionar um elemento precisamos:

- encontrar sua posição correta

Inserção e Remoção - Vetor Ordenado

Para adicionar um elemento precisamos:

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

Inserção e Remoção - Vetor Ordenado

Para adicionar um elemento precisamos:

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

Inserção e Remoção - Vetor Ordenado

Para adicionar um elemento precisamos:

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

Inserção e Remoção - Vetor Ordenado

Para adicionar um elemento precisamos:

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

Inserção e Remoção - Vetor Ordenado

Para adicionar um elemento precisamos:

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

```
1 void adicionar_elemento(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 }
```

Inserção e Remoção - Vetor Ordenado

Para adicionar um elemento precisamos:

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

```
1 void adicionar_elemento(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 }
```

Inserção e Remoção - Vetor Ordenado

Para adicionar um elemento precisamos:

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

```
1 void adicionar_elemento(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 }
```

$O(n)$

Inserção e Remoção - Vetor Ordenado

Para adicionar um elemento precisamos:

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

```
1 void adicionar_elemento(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 }
```

$O(n)$

Para remover um elemento precisamos:

Inserção e Remoção - Vetor Ordenado

Para adicionar um elemento precisamos:

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

```
1 void adicionar_elemento(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 }
```

$O(n)$

Para remover um elemento precisamos:

- deslocar os elementos para a esquerda

Inserção e Remoção - Vetor Ordenado

Para adicionar um elemento precisamos:

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

```
1 void adicionar_elemento(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 }
```

$O(n)$

Para remover um elemento precisamos:

- deslocar os elementos para a esquerda

Inserção e Remoção - Vetor Ordenado

Para adicionar um elemento precisamos:

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

```
1 void adicionar_elemento(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 }
```

$O(n)$

Para remover um elemento precisamos:

- deslocar os elementos para a esquerda

Inserção e Remoção - Vetor Ordenado

Para adicionar um elemento precisamos:

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

```
1 void adicionar_elemento(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 }
```

$O(n)$

Para remover um elemento precisamos:

- deslocar os elementos para a esquerda

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

Inserção e Remoção - Vetor Ordenado

Para adicionar um elemento precisamos:

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

```
1 void adicionar_elemento(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 }
```

$O(n)$

Para remover um elemento precisamos:

- deslocar os elementos para a esquerda

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

Inserção e Remoção - Vetor Ordenado

Para adicionar um elemento precisamos:

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

```
1 void adicionar_elemento(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 }
```

$O(n)$

Para remover um elemento precisamos:

- deslocar os elementos para a esquerda

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

$O(n)$

Busca - Vetor Ordenado

Realizamos busca binária em $\mathrm{O}(\lg n)$

Busca - Vetor Ordenado

Realizamos busca binária em $\mathrm{O}(\lg n)$

```
 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 }
```

Busca - Vetor Ordenado

Realizamos busca binária em $\mathrm{O}(\lg n)$

```
 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 }
```

Busca - Vetor Ordenado

Realizamos busca binária em $\mathrm{O}(\lg n)$

```
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(vetor *v, int x) {
14   return busca_binaria(v->dados, 0, v->n - 1, x);
15 }
```

Vetores Não Ordenados vs. Ordenados

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

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:

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

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

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:

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

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?

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 $\mathrm{O}(\lg n)$

Vetores Dinâmicos

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

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

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

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

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`

Mudança na `struct`

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

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

Mudança na `struct`

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

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

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

Mudança na `struct`

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

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

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

Alterações na Inicialização

Basta armazenar no campo `alocado` qual o tamanho inicial

```
1 void iniciar_vetor(vetor **v, int tam) {
2   *v = malloc(sizeof(vetor));
3   (*v)->dados = malloc(tam * sizeof(int));
4   (*v)->n = 0;
5   (*v)->alocado = tam;
6 }
```

Alterações na Inserção

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

Alterações na Inserção

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

```
1 void adicionar_elemento(vetor *v, int x) {
2   int i, *temp;
```

Alterações na Inserção

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

```
1 void adicionar_elemento(vetor *v, int x) {
2   int i, *temp;
```

Alterações na Inserção

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

```
1 void adicionar_elemento(vetor *v, int x) {
2   int i, *temp;
3   if (v->n == v->alocado) {
```

Alterações na Inserção

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

```
1 void adicionar_elemento(vetor *v, int x) {
2   int i, *temp;
3   if (v->n == v->alocado) {
4     temp = v->dados;
```

Alterações na Inserção

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

```
1 void adicionar_elemento(vetor *v, int x) {
2   int i, *temp;
3   if (v->n == v->alocado) {
4     temp = v->dados;
5     v->alocado *= 2;
```

Alterações na Inserção

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

```
1 void adicionar_elemento(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));
```

Alterações na Inserção

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

```
1 void adicionar_elemento(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];
```

Alterações na Inserção

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

```
1 void adicionar_elemento(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);
```

Alterações na Inserção

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

```
1  void adicionar_elemento(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   }
```

Alterações na Inserção

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

```
1  void adicionar_elemento(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:

Alterações na Inserção

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

```
1  void adicionar_elemento(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

Alterações na Inserção

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

```
1  void adicionar_elemento(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 $O(i)$

Tempo para inserir n elementos

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

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

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 $O(n)$

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

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 $O(1)$

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:

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

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

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

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

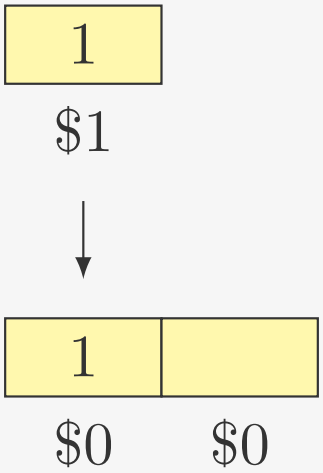
Dessa forma, nunca ficamos devendo

Simulação da contabilidade

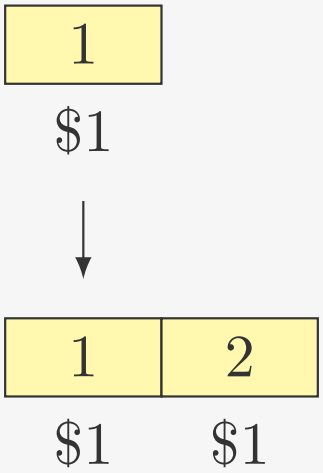
$0

Simulação da contabilidade

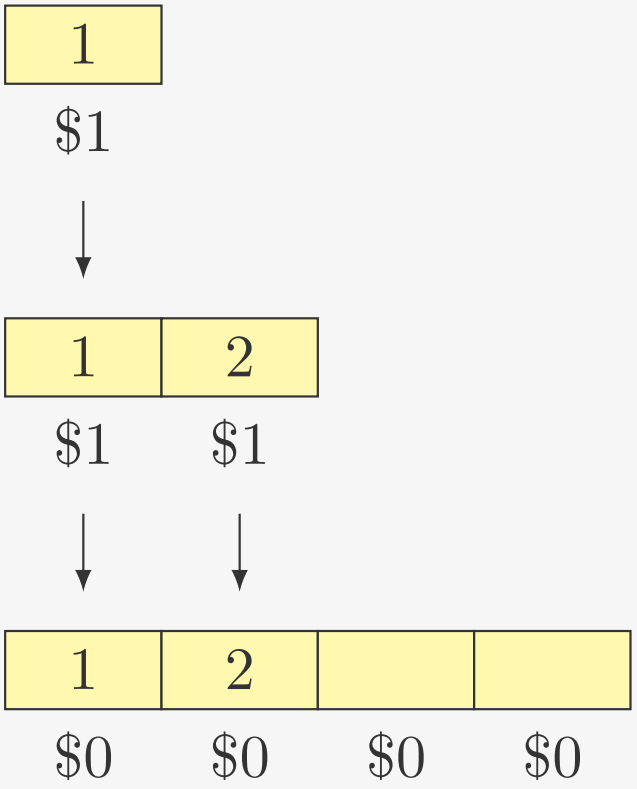
Simulação da contabilidade



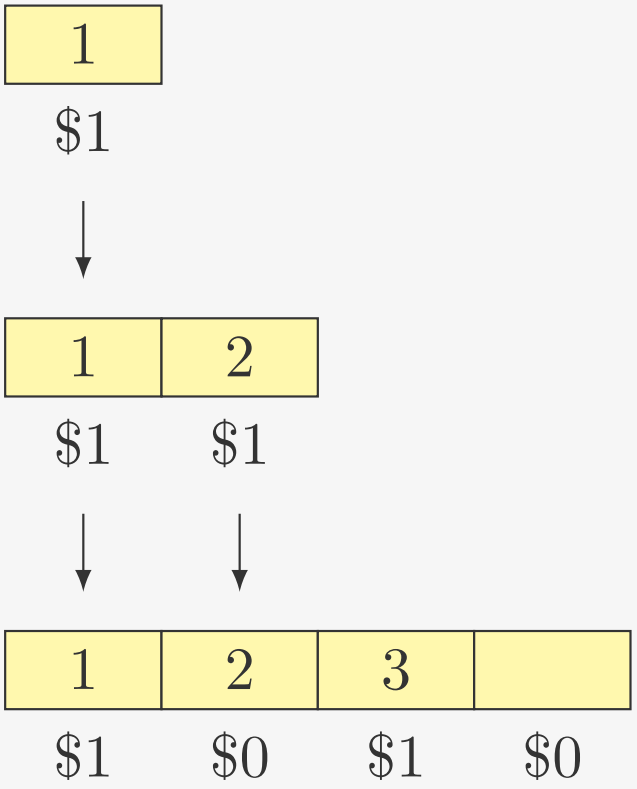
Simulação da contabilidade



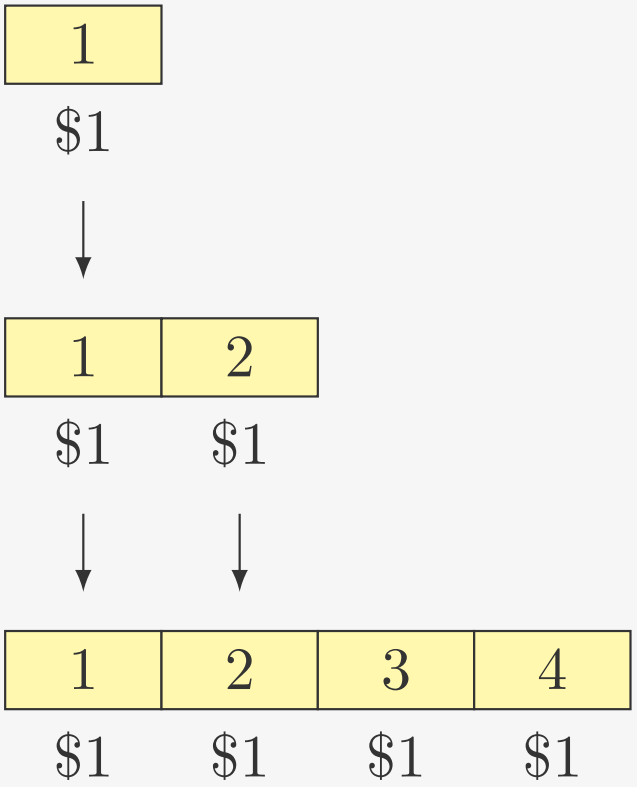
Simulação da contabilidade



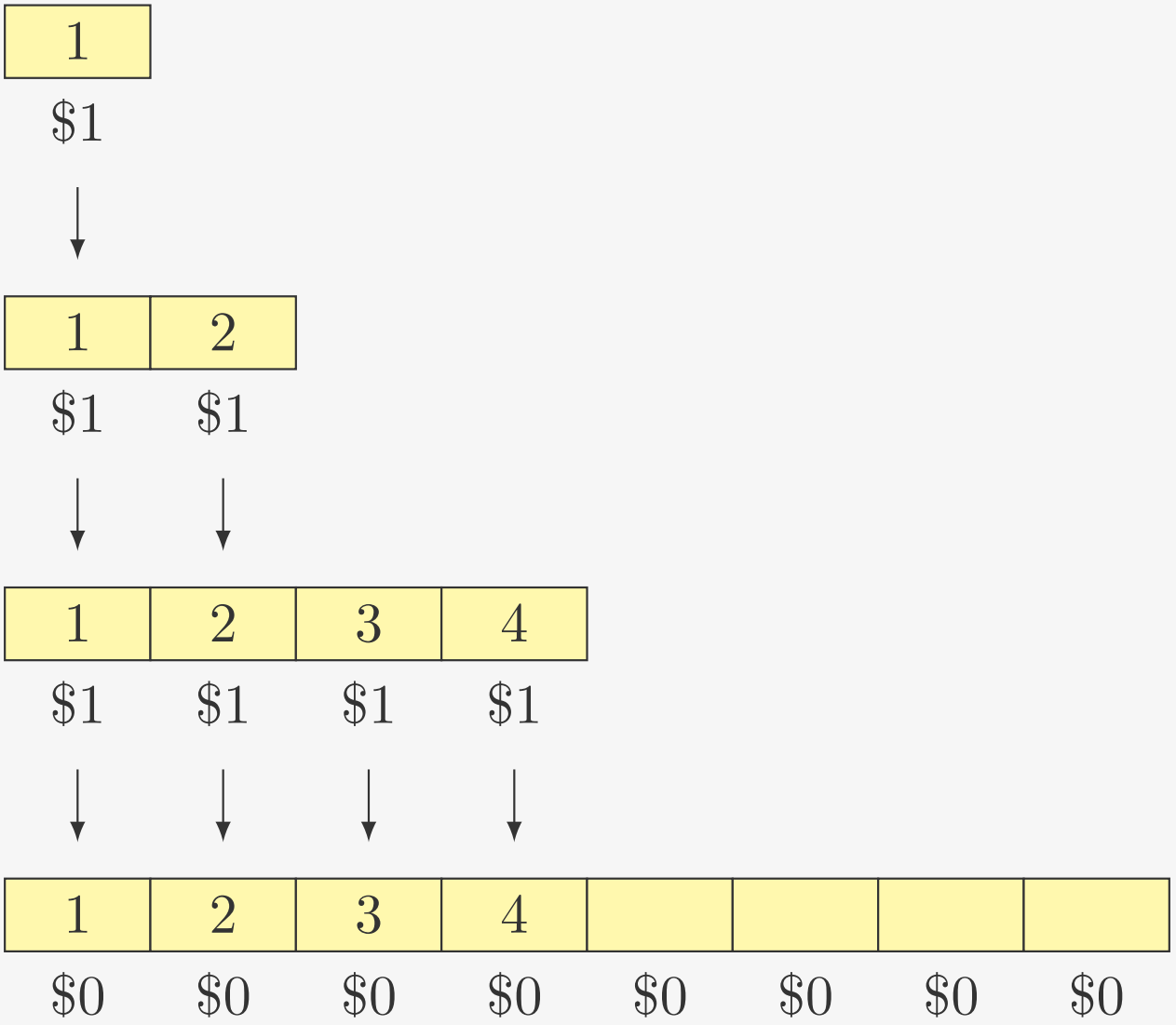
Simulação da contabilidade



Simulação da contabilidade



Simulação da contabilidade



Alterações na Remoção

Para não desperdiçar espaço, diminuímos o vetor ao remover

Alterações na Remoção

Para não desperdiçar espaço, diminuímos o vetor ao remover

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

Alterações na Remoção

Para não desperdiçar espaço, diminuímos o vetor ao remover

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

- Melhor do que quando estiver $1/2$ cheio

Alterações na Remoção

Para não desperdiçar espaço, diminuímos o vetor ao remover

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

- Melhor do que quando estiver $1/2$ cheio

Custo amortizado de $\mathrm{O}(1)$

Alterações na Remoção

Para não desperdiçar espaço, diminuímos o vetor ao remover

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

- Melhor do que quando estiver $1/2$ cheio

Custo amortizado de $\mathrm{O}(1)$

Implementação: Exercício

Vetores Dinâmicos - Conclusão

Vetores dinâmicos:

Vetores Dinâmicos - Conclusão

Vetores dinâmicos:

- Inserção e Remoção em $O(1)$ (amortizado)

Vetores Dinâmicos - Conclusão

Vetores dinâmicos:

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

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

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

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)

Exercícios

1. Faça uma função recursiva com protótipo `void move_esquerda(vetor *v, int i)` que move todo o conteúdo do vetor da posição `i` para frente uma posição para esquerda. Altere a remoção do TAD vetor para utilizar essa função.
2. Faça uma versão iterativa da função `busca_binaria`.
3. Pesquise sobre a função `realloc` e implemente a inserção e da remoção do vetor dinâmico usando tal função.

Exercícios

5. Faça a implementação de um Tipo Abstrato de Dados que armazena um conjunto de números inteiros.
   - Lembre-se que conjuntos não podem conter elementos repetidos.
   - Estabeleça a interface do seu TAD pensando em operações que o usuário poderia realizar.
   - Projete a melhor implementação possível para o mesmo considerando a eficiência dos algoritmos envolvidos.
   - Talvez você precise fazer hipóteses sobre o uso do TAD.