

# MC-202

## Listas Ligadas

Lehilton Pedrosa

Universidade Estadual de Campinas

Segundo semestre de 2021

# Vetores

Vetores:

# Vetores

Vetores:

- estão alocados contiguamente na memória

# Vetores

Vetores:

- estão alocados contiguamente na memória
  - pode ser que tenhamos espaço na memória

# Vetores

Vetores:

- estão alocados contiguamente na memória
  - pode ser que tenhamos espaço na memória
  - mas não para alocar um vetor do tamanho desejado

# Vetores

Vetores:

- estão alocados contiguamente na memória
  - pode ser que tenhamos espaço na memória
  - mas não para alocar um vetor do tamanho desejado
- tem um tamanho fixo

# Vetores

Vetores:

- estão alocados contiguamente na memória
  - pode ser que tenhamos espaço na memória
  - mas não para alocar um vetor do tamanho desejado
- tem um tamanho fixo
  - ou alocamos um vetor pequeno e o espaço pode acabar

# Vetores

Vetores:

- estão alocados contiguamente na memória
  - pode ser que tenhamos espaço na memória
  - mas não para alocar um vetor do tamanho desejado
- tem um tamanho fixo
  - ou alocamos um vetor pequeno e o espaço pode acabar
  - ou alocamos um vetor grande e desperdiçamos memória



# Alternativa - Lista Ligada

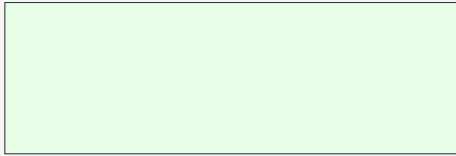


Pilha

## Alternativa - Lista Ligada

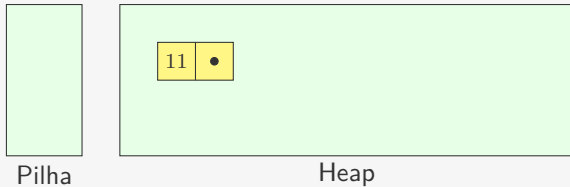


Pilha



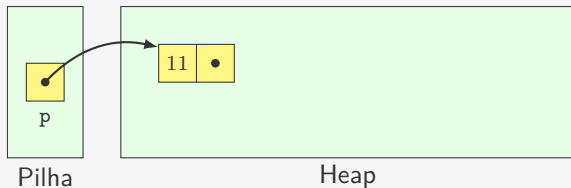
Heap

## Alternativa - Lista Ligada



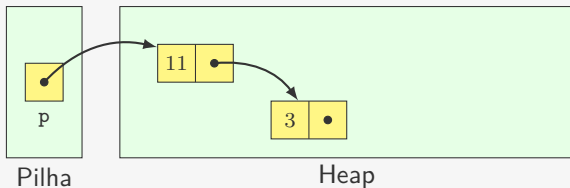
- alocamos memória conforme o necessário

## Alternativa - Lista Ligada



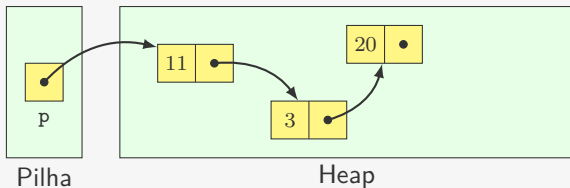
- alocamos memória conforme o necessário
- guardamos um ponteiro para a estrutura em uma variável

## Alternativa - Lista Ligada



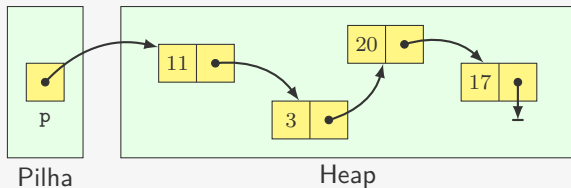
- alocamos memória conforme o necessário
- guardamos um ponteiro para a estrutura em uma variável
- o primeiro **nó** aponta para o segundo

## Alternativa - Lista Ligada



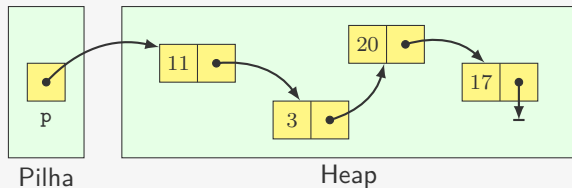
- alocamos memória conforme o necessário
- guardamos um ponteiro para a estrutura em uma variável
- o primeiro **nó** aponta para o segundo
- o segundo nó aponta para o terceiro

## Alternativa - Lista Ligada



- alocamos memória conforme o necessário
- guardamos um ponteiro para a estrutura em uma variável
- o primeiro **nó** aponta para o segundo
- o segundo nó aponta para o terceiro

## Alternativa - Lista Ligada



- alocamos memória conforme o necessário
- guardamos um ponteiro para a estrutura em uma variável
- o primeiro **nó** aponta para o segundo
- o segundo nó aponta para o terceiro
- o último nó aponta para **NULL**



# Listas ligadas

Nó: elemento alocado dinamicamente que contém

# Listas ligadas

Nó: elemento alocado dinamicamente que contém

- um conjunto de dados

# Listas ligadas

Nó: elemento alocado dinamicamente que contém

- um conjunto de dados
- um ponteiro para outro nó

# Listas ligadas

Nó: elemento alocado dinamicamente que contém

- um conjunto de dados
- um ponteiro para outro nó

Lista ligada:

# Listas ligadas

Nó: elemento alocado dinamicamente que contém

- um conjunto de dados
- um ponteiro para outro nó

Lista ligada:

- Conjunto de nós ligados entre si de maneira sequencial

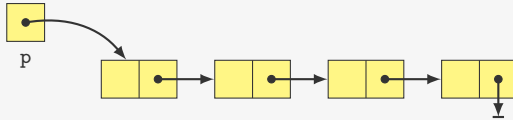
# Listas ligadas

Nó: elemento alocado dinamicamente que contém

- um conjunto de dados
- um ponteiro para outro nó

Lista ligada:

- Conjunto de nós ligados entre si de maneira sequencial



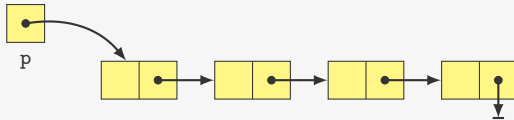
# Listas ligadas

Nó: elemento alocado dinamicamente que contém

- um conjunto de dados
- um ponteiro para outro nó

Lista ligada:

- Conjunto de nós ligados entre si de maneira sequencial



Observações:

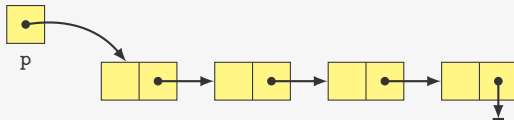
# Listas ligadas

Nó: elemento alocado dinamicamente que contém

- um conjunto de dados
- um ponteiro para outro nó

Lista ligada:

- Conjunto de nós ligados entre si de maneira sequencial



Observações:

- a lista ligada é acessada a partir de uma variável



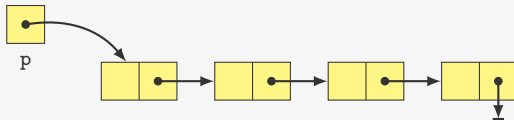
# Listas ligadas

Nó: elemento alocado dinamicamente que contém

- um conjunto de dados
- um ponteiro para outro nó

Lista ligada:

- Conjunto de nós ligados entre si de maneira sequencial



Observações:

- a lista ligada é acessada a partir de uma variável
- um ponteiro pode estar vazio (aponta para **NULL** em C)

# Implementação em C

Definição do Nó:

# Implementação em C

Definição do Nó:

```
1 typedef struct no *p_no;
2
3 struct no {
4     int dado;
5     p_no prox;
6 };
```

# Implementação em C

Definição do Nó:

```
1 typedef struct no *p_no;  
2  
3 struct no {  
4     int dado;  
5     p_no prox;  
6 };
```

Observações

# Implementação em C

Definição do Nó:

```
1 typedef struct no *p_no;
2
3 struct no {
4     int dado;
5     p_no prox;
6 };
```

Observações

- `p_no` é um apelido de ponteiro para `struct no`

# Implementação em C

Definição do Nó:

```
1 typedef struct no *p_no;
2
3 struct no {
4     int dado;
5     p_no prox;
6 };
```

Observações

- `p_no` é um apelido de ponteiro para `struct no`
- podemos usar `typedef` com ponteiro antes da struct

# Implementação em C

## Definição do Nó:

```
1 typedef struct no *p_no;
2
3 struct no {
4     int dado;
5     p_no prox;
6 };
```

## Observações

- `p_no` é um apelido de ponteiro para `struct no`
- podemos usar `typedef` com ponteiro antes da struct
- assim podemos usar `p_no` dentro da definição do registro

# Implementação em C

## Definição do Nó:

```
1 typedef struct no *p_no;
2
3 struct no {
4     int dado;
5     p_no prox;
6 };
```

## Observações

- `p_no` é um apelido de ponteiro para `struct no`
- podemos usar `typedef` com ponteiro antes da struct
- assim podemos usar `p_no` dentro da definição do registro
- também poderíamos escrever `struct no *`



# Criando e Destruindo

Cria uma lista vazia:

# Criando e Destruindo

Cria uma lista vazia:

```
1 p_no criar_lista() {  
2     return NULL;  
3 }
```

# Criando e Destruindo

Cria uma lista vazia:

```
1 p_no criar_lista() {  
2     return NULL;  
3 }
```

Código no cliente:

# Criando e Destruindo

Cria uma lista vazia:

```
1 p_no criar_lista() {  
2     return NULL;  
3 }
```

Código no cliente:

```
1 p_no lista;  
2 lista = criar_lista();
```

# Criando e Destruindo

Cria uma lista vazia:

```
1 p_no criar_lista() {  
2     return NULL;  
3 }
```

Código no cliente:

```
1 p_no lista;  
2 lista = criar_lista();
```

Destruindo a lista:

# Criando e Destruindo

Cria uma lista vazia:

```
1 p_no criar_lista() {  
2     return NULL;  
3 }
```

Código no cliente:

```
1 p_no lista;  
2 lista = criar_lista();
```

Destruindo a lista:

```
1 void destruir_lista(p_no lista) {  
2     if (lista != NULL) {  
3         destruir_lista(lista->prox);  
4         free(lista);  
5     }  
6 }
```

# Criando e Destruindo

Cria uma lista vazia:

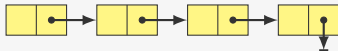
```
1 p_no criar_lista() {  
2     return NULL;  
3 }
```

Código no cliente:

```
1 p_no lista;  
2 lista = criar_lista();
```

Destruindo a lista:

```
1 void destruir_lista(p_no lista) {  
2     if (lista != NULL) {  
3         destruir_lista(lista->prox);  
4         free(lista);  
5     }  
6 }
```



# Criando e Destruindo

Cria uma lista vazia:

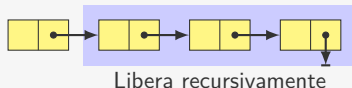
```
1 p_no criar_lista() {  
2     return NULL;  
3 }
```

Código no cliente:

```
1 p_no lista;  
2 lista = criar_lista();
```

Destruindo a lista:

```
1 void destruir_lista(p_no lista) {  
2     if (lista != NULL) {  
3         destruir_lista(lista->prox);  
4         free(lista);  
5     }  
6 }
```





# Criando e Destruindo

Cria uma lista vazia:

```
1 p_no criar_lista() {  
2     return NULL;  
3 }
```

Código no cliente:

```
1 p_no lista;  
2 lista = criar_lista();
```

Destruindo a lista:

```
1 void destruir_lista(p_no lista) {  
2     if (lista != NULL) {  
3         destruir_lista(lista->prox);  
4         free(lista);  
5     }  
6 }
```



# Criando e Destruindo

Cria uma lista vazia:

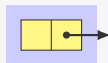
```
1 p_no criar_lista() {  
2     return NULL;  
3 }
```

Código no cliente:

```
1 p_no lista;  
2 lista = criar_lista();
```

Destruindo a lista:

```
1 void destruir_lista(p_no lista) {  
2     if (lista != NULL) {  
3         destruir_lista(lista->prox);  
4         free(lista);  
5     }  
6 }
```



Libera

# Criando e Destruindo

Cria uma lista vazia:

```
1 p_no criar_lista() {  
2     return NULL;  
3 }
```

Código no cliente:

```
1 p_no lista;  
2 lista = criar_lista();
```

Destruindo a lista:

```
1 void destruir_lista(p_no lista) {  
2     if (lista != NULL) {  
3         destruir_lista(lista->prox);  
4         free(lista);  
5     }  
6 }
```

# Criando e Destruindo

Cria uma lista vazia:

```
1 p_no criar_lista() {  
2     return NULL;  
3 }
```

Código no cliente:

```
1 p_no lista;  
2 lista = criar_lista();
```

Destruindo a lista:

```
1 void destruir_lista(p_no lista) {  
2     if (lista != NULL) {  
3         destruir_lista(lista->prox);  
4         free(lista);  
5     }  
6 }
```

Exercício: faça uma versão iterativa de `destruir_lista`

## Adicionando elementos

A função devolve uma “nova” lista

## Adicionando elementos

A função devolve uma “nova” lista

- É a lista antiga com o elemento novo adicionado

## Adicionando elementos

A função devolve uma “nova” lista

- É a lista antiga com o elemento novo adicionado
- Evita ter que passar o ponteiro por referência

## Adicionando elementos

A função devolve uma “nova” lista

- É a lista antiga com o elemento novo adicionado
- Evita ter que passar o ponteiro por referência

```
1 p_no adicionar_elemento(p_no lista, int x) {
2     p_no novo;
3     novo = malloc(sizeof(struct no));
4     novo->dado = x;
5     novo->prox = lista;
6     return novo;
7 }
```



## Adicionando elementos

A função devolve uma “nova” lista

- É a lista antiga com o elemento novo adicionado
- Evita ter que passar o ponteiro por referência

```
1 p_no adicionar_elemento(p_no lista, int x) {
2     p_no novo;
3     novo = malloc(sizeof(struct no));
4     novo->dado = x;
5     novo->prox = lista;
6     return novo;
7 }
```

Código no cliente:

## Adicionando elementos

A função devolve uma “nova” lista

- É a lista antiga com o elemento novo adicionado
- Evita ter que passar o ponteiro por referência

```
1 p_no adicionar_elemento(p_no lista, int x) {
2     p_no novo;
3     novo = malloc(sizeof(struct no));
4     novo->dado = x;
5     novo->prox = lista;
6     return novo;
7 }
```

Código no cliente:

```
1 lista = adicionar_elemento(lista, num);
```

## Adicionando elementos

A função devolve uma “nova” lista

- É a lista antiga com o elemento novo adicionado
- Evita ter que passar o ponteiro por referência

```
1 p_no adicionar_elemento(p_no lista, int x) {
2     p_no novo;
3     novo = malloc(sizeof(struct no));
4     novo->dado = x;
5     novo->prox = lista;
6     return novo;
7 }
```

Código no cliente:

```
1 lista = adicionar_elemento(lista, num);
```

- A inserção ocorre em  $O(1)$

## Adicionando elementos

A função devolve uma “nova” lista

- É a lista antiga com o elemento novo adicionado
- Evita ter que passar o ponteiro por referência

```
1 p_no adicionar_elemento(p_no lista, int x) {
2     p_no novo;
3     novo = malloc(sizeof(struct no));
4     novo->dado = x;
5     novo->prox = lista;
6     return novo;
7 }
```

Código no cliente:

```
1 lista = adicionar_elemento(lista, num);
```

- A inserção ocorre em  $O(1)$
- Deveria verificar se `malloc` não devolve `NULL`

## Adicionando elementos

A função devolve uma “nova” lista

- É a lista antiga com o elemento novo adicionado
- Evita ter que passar o ponteiro por referência

```
1 p_no adicionar_elemento(p_no lista, int x) {
2     p_no novo;
3     novo = malloc(sizeof(struct no));
4     novo->dado = x;
5     novo->prox = lista;
6     return novo;
7 }
```

Código no cliente:

```
1 lista = adicionar_elemento(lista, num);
```

- A inserção ocorre em  $O(1)$
- Deveria verificar se `malloc` não devolve `NULL`
  - Teria acabado a memória

## Adicionando elementos

A função devolve uma “nova” lista

- É a lista antiga com o elemento novo adicionado
- Evita ter que passar o ponteiro por referência

```
1 p_no adicionar_elemento(p_no lista, int x) {
2     p_no novo;
3     novo = malloc(sizeof(struct no));
4     novo->dado = x;
5     novo->prox = lista;
6     return novo;
7 }
```

Código no cliente:

```
1 lista = adicionar_elemento(lista, num);
```

- A inserção ocorre em  $O(1)$
- Deveria verificar se `malloc` não devolve `NULL`
  - Teria acabado a memória
  - Será omitido, mas precisa ser tratado na prática

# Impressão

Impressão iterativa:

# Impressão

Impressão iterativa:

```
1 void imprime(p_no lista) {  
2     p_no atual;  
3     for (atual = lista; atual != NULL; atual = atual->prox)  
4         printf("%d\n", atual->dado);  
5 }
```



# Impressão

Impressão iterativa:

```
1 void imprime(p_no lista) {  
2     p_no atual;  
3     for (atual = lista; atual != NULL; atual = atual->prox)  
4         printf("%d\n", atual->dado);  
5 }
```

Impressão recursiva:

# Impressão

## Impressão iterativa:

```
1 void imprime(p_no lista) {
2     p_no atual;
3     for (atual = lista; atual != NULL; atual = atual->prox)
4         printf("%d\n", atual->dado);
5 }
```

## Impressão recursiva:

```
1 void imprime_recursivo(p_no lista) {
2     if (lista != NULL){
3         printf("%d\n", lista->dado);
4         imprime_recursivo(lista->prox);
5     }
6 }
```

# Impressão

Impressão iterativa:

```
1 void imprime(p_no lista) {
2     p_no atual;
3     for (atual = lista; atual != NULL; atual = atual->prox)
4         printf("%d\n", atual->dado);
5 }
```

Impressão recursiva:

```
1 void imprime_recursivo(p_no lista) {
2     if (lista != NULL){
3         printf("%d\n", lista->dado);
4         imprime_recursivo(lista->prox);
5     }
6 }
```

Algoritmos recursivos para lista ligada são, em geral, mais elegantes e simples

# Impressão

Impressão iterativa:

```
1 void imprime(p_no lista) {
2     p_no atual;
3     for (atual = lista; atual != NULL; atual = atual->prox)
4         printf("%d\n", atual->dado);
5 }
```

Impressão recursiva:

```
1 void imprime_recursivo(p_no lista) {
2     if (lista != NULL){
3         printf("%d\n", lista->dado);
4         imprime_recursivo(lista->prox);
5     }
6 }
```

Algoritmos recursivos para lista ligada são, em geral, mais elegantes e simples

- Porém, os iterativos costumam ser mais rápidos

# Impressão

Impressão iterativa:

```
1 void imprime(p_no lista) {
2     p_no atual;
3     for (atual = lista; atual != NULL; atual = atual->prox)
4         printf("%d\n", atual->dado);
5 }
```

Impressão recursiva:

```
1 void imprime_recursivo(p_no lista) {
2     if (lista != NULL){
3         printf("%d\n", lista->dado);
4         imprime_recursivo(lista->prox);
5     }
6 }
```

Algoritmos recursivos para lista ligada são, em geral, mais elegantes e simples

- Porém, os iterativos costumam ser mais rápidos
- Não arcam com o overhead da recursão

## Exemplo - lendo números positivos

## Exemplo - lendo números positivos

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "lista_ligada.h"
4
5 int main() {
6     int num;
```

## Exemplo - lendo números positivos

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "lista_ligada.h"
4
5 int main() {
6     int num;
7     p_no lista;
8     lista = criar_lista();
```



## Exemplo - lendo números positivos

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "lista_ligada.h"
4
5 int main() {
6     int num;
7     p_no lista;
8     lista = criar_lista();
9     /*lê números positivos e armazena na lista*/
10    do {
11        scanf("%d", &num);
12        if (num > 0)
13            lista = adicionar_elemento(lista, num);
14    } while (num > 0);
```

## Exemplo - lendo números positivos

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "lista_ligada.h"
4
5 int main() {
6     int num;
7     p_no lista;
8     lista = criar_lista();
9     /*lê números positivos e armazena na lista*/
10    do {
11        scanf("%d", &num);
12        if (num > 0)
13            lista = adicionar_elemento(lista, num);
14    } while (num > 0);
15    imprime(lista); /*(em ordem reversa de inserção)*/
```

## Exemplo - lendo números positivos

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "lista_ligada.h"
4
5 int main() {
6     int num;
7     p_no lista;
8     lista = criar_lista();
9     /*lê números positivos e armazena na lista*/
10    do {
11        scanf("%d", &num);
12        if (num > 0)
13            lista = adicionar_elemento(lista, num);
14    } while (num > 0);
15    imprime(lista); /*(em ordem reversa de inserção)*/
16    destruir_lista(lista);
17    return 0;
18 }
```

## Comparando vetores e listas ligadas

- Acesso a posição  $k$ :

## Comparando vetores e listas ligadas

- Acesso a posição  $k$ :
  - Vetor:  $O(1)$

## Comparando vetores e listas ligadas

- Acesso a posição  $k$ :
  - Vetor:  $O(1)$
  - Lista:  $O(k)$  (precisa percorrer a lista)

## Comparando vetores e listas ligadas

- Acesso a posição  $k$ :
  - Vetor:  $O(1)$
  - Lista:  $O(k)$  (precisa percorrer a lista)
- Inserção na posição  $0$ :

## Comparando vetores e listas ligadas

- Acesso a posição  $k$ :
  - Vetor:  $O(1)$
  - Lista:  $O(k)$  (precisa percorrer a lista)
- Inserção na posição  $0$ :
  - Vetor:  $O(n)$  (precisa mover itens para a direita)



## Comparando vetores e listas ligadas

- Acesso a posição  $k$ :
  - Vetor:  $O(1)$
  - Lista:  $O(k)$  (precisa percorrer a lista)
- Inserção na posição  $0$ :
  - Vetor:  $O(n)$  (precisa mover itens para a direita)
  - Lista:  $O(1)$

## Comparando vetores e listas ligadas

- Acesso a posição  $k$ :
  - Vetor:  $O(1)$
  - Lista:  $O(k)$  (precisa percorrer a lista)
- Inserção na posição  $0$ :
  - Vetor:  $O(n)$  (precisa mover itens para a direita)
  - Lista:  $O(1)$
- Remoção da posição  $0$ :

## Comparando vetores e listas ligadas

- Acesso a posição  $k$ :
  - Vetor:  $O(1)$
  - Lista:  $O(k)$  (precisa percorrer a lista)
- Inserção na posição  $0$ :
  - Vetor:  $O(n)$  (precisa mover itens para a direita)
  - Lista:  $O(1)$
- Remoção da posição  $0$ :
  - Vetor:  $O(n)$  (precisa mover itens para a esquerda)

## Comparando vetores e listas ligadas

- Acesso a posição  $k$ :
  - Vetor:  $O(1)$
  - Lista:  $O(k)$  (precisa percorrer a lista)
- Inserção na posição  $0$ :
  - Vetor:  $O(n)$  (precisa mover itens para a direita)
  - Lista:  $O(1)$
- Remoção da posição  $0$ :
  - Vetor:  $O(n)$  (precisa mover itens para a esquerda)
  - Lista:  $O(1)$

## Comparando vetores e listas ligadas

- Acesso a posição  $k$ :
  - Vetor:  $O(1)$
  - Lista:  $O(k)$  (precisa percorrer a lista)
- Inserção na posição 0:
  - Vetor:  $O(n)$  (precisa mover itens para a direita)
  - Lista:  $O(1)$
- Remoção da posição 0:
  - Vetor:  $O(n)$  (precisa mover itens para a esquerda)
  - Lista:  $O(1)$
- Uso de espaço:

## Comparando vetores e listas ligadas

- Acesso a posição  $k$ :
  - Vetor:  $O(1)$
  - Lista:  $O(k)$  (precisa percorrer a lista)
- Inserção na posição 0:
  - Vetor:  $O(n)$  (precisa mover itens para a direita)
  - Lista:  $O(1)$
- Remoção da posição 0:
  - Vetor:  $O(n)$  (precisa mover itens para a esquerda)
  - Lista:  $O(1)$
- Uso de espaço:
  - Vetor: provavelmente desperdiçará memória

## Comparando vetores e listas ligadas

- Acesso a posição  $k$ :
  - Vetor:  $O(1)$
  - Lista:  $O(k)$  (precisa percorrer a lista)
- Inserção na posição 0:
  - Vetor:  $O(n)$  (precisa mover itens para a direita)
  - Lista:  $O(1)$
- Remoção da posição 0:
  - Vetor:  $O(n)$  (precisa mover itens para a esquerda)
  - Lista:  $O(1)$
- Uso de espaço:
  - Vetor: provavelmente desperdiçará memória
  - Lista: não desperdiça memória, mas cada elemento consome mais memória por causa do ponteiro

## Comparando vetores e listas ligadas

- Acesso a posição  $k$ :
  - Vetor:  $O(1)$
  - Lista:  $O(k)$  (precisa percorrer a lista)
- Inserção na posição  $0$ :
  - Vetor:  $O(n)$  (precisa mover itens para a direita)
  - Lista:  $O(1)$
- Remoção da posição  $0$ :
  - Vetor:  $O(n)$  (precisa mover itens para a esquerda)
  - Lista:  $O(1)$
- Uso de espaço:
  - Vetor: provavelmente desperdiçará memória
  - Lista: não desperdiça memória, mas cada elemento consome mais memória por causa do ponteiro

Qual é melhor?



# Comparando vetores e listas ligadas

- Acesso a posição  $k$ :
  - Vetor:  $O(1)$
  - Lista:  $O(k)$  (precisa percorrer a lista)
- Inserção na posição  $0$ :
  - Vetor:  $O(n)$  (precisa mover itens para a direita)
  - Lista:  $O(1)$
- Remoção da posição  $0$ :
  - Vetor:  $O(n)$  (precisa mover itens para a esquerda)
  - Lista:  $O(1)$
- Uso de espaço:
  - Vetor: provavelmente desperdiçará memória
  - Lista: não desperdiça memória, mas cada elemento consome mais memória por causa do ponteiro

Qual é melhor?

- depende do problema, do algoritmo e da implementação

## Exercício - Busca e Remoção

Faça uma função que busca um elemento **x** em uma lista ligada, devolvendo o ponteiro para o nó encontrado ou **NULL** se o elemento não estiver na lista.

Faça uma função que remove a primeira ocorrência (se existir) de um elemento **x** de uma lista ligada dada.

Faça uma função que remove todas as ocorrências de um elemento **x** de uma lista ligada dada.