MC-202 — Unidade 6
Listas Ligadas

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

Universidade Estadual de Campinas

1º semestre/2018

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

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

Vetores dinâmicos:

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

Vetores dinâmicos:

- resolvem o problema do tamanho fixo parcialmente

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

Vetores dinâmicos:

- resolvem o problema do tamanho fixo parcialmente
  - ainda podemos ter um grande desperdício de memória

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

Vetores dinâmicos:

- resolvem o problema do tamanho fixo parcialmente
  - ainda podemos ter um grande desperdício de memória
  - ex: usamos 64GB para armazenar um vetor de 16GB

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

Vetores dinâmicos:

- resolvem o problema do tamanho fixo parcialmente
  - ainda podemos ter um grande desperdício de memória
  - ex: usamos 64GB para armazenar um vetor de 16GB
- inserção/remoção é rápida na maior parte das vezes, mas em algumas operações demora muito

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

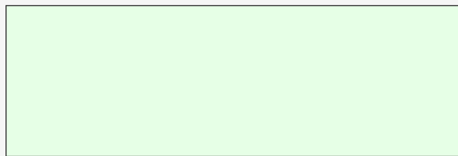
Vetores dinâmicos:

- resolvem o problema do tamanho fixo parcialmente
  - ainda podemos ter um grande desperdício de memória
  - ex: usamos 64GB para armazenar um vetor de 16GB
- inserção/remoção é rápida na maior parte das vezes, mas em algumas operações demora muito
  - ruim para aplicações de “tempo real”

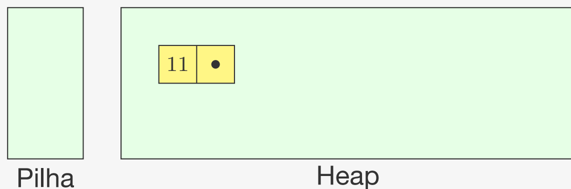
Alternativa - Lista Ligada

Pilha

Alternativa - Lista Ligada

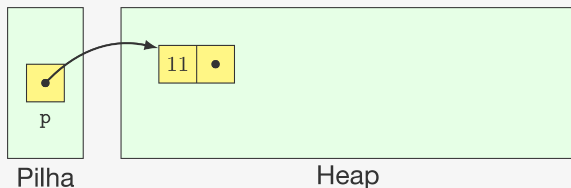


Alternativa - Lista Ligada



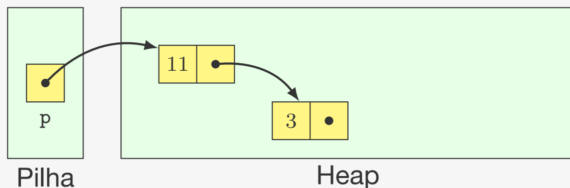
- alocamos memória conforme o necessário

Alternativa - Lista Ligada



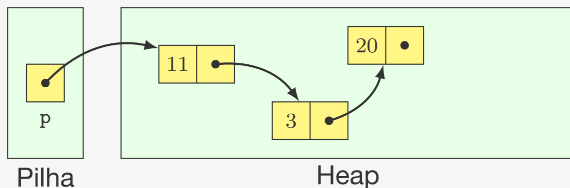
- alocamos memória conforme o necessário
- guardamos um ponteiro para a estrutura em uma variável

Alternativa - Lista Ligada



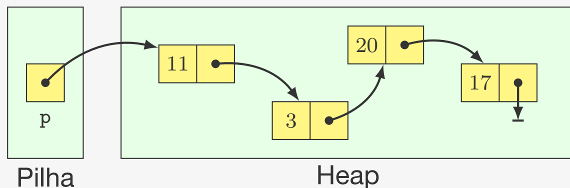
- alocamos memória conforme o necessário
- guardamos um ponteiro para a estrutura em uma variável
- o primeiro nó aponta para o segundo

Alternativa - Lista Ligada



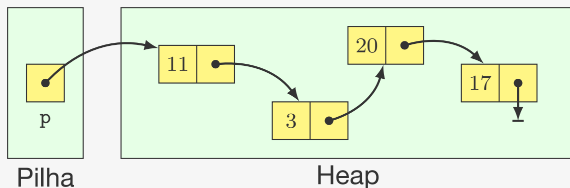
- alocamos memória conforme o necessário
- guardamos um ponteiro para a estrutura em uma variável
- o primeiro nó aponta para o segundo
- o segundo nó aponta para o terceiro

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

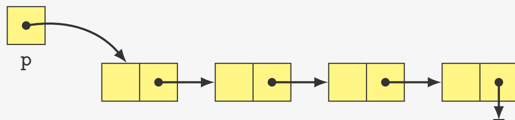
Listas ligadas

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



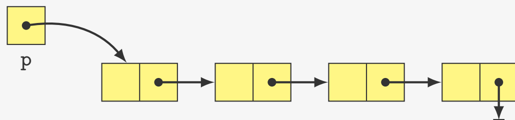
Listas ligadas

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:

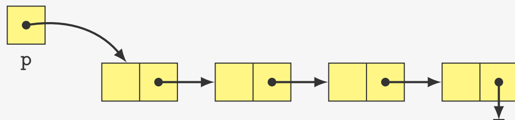
Listas ligadas

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

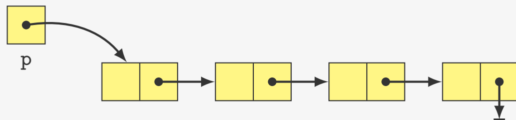
Listas ligadas

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

Implementação em C

Definição do Nó:

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

Observações

Implementação em C

Definição do Nó:

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

Observações

- `typedef` define um apelido `No` para o tipo `struct No`

Implementação em C

Definição do Nó:

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

Observações

- `typedef` define um apelido `No` para o tipo `struct No`
- deve-se usar `struct No` dentro do registro, porque o apelido ainda não existe

Implementação em C

Definição do Nó:

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

Observações

- `typedef` define um apelido `No` para o tipo `struct No`
- deve-se usar `struct No` dentro do registro, porque o apelido ainda não existe
- os nomes do `struct` e do `typedef` podem ser distintos

Implementação em C

Definição do Nó:

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

Observações

- `typedef` define um apelido `No` para o tipo `struct No`
- deve-se usar `struct No` dentro do registro, porque o apelido ainda não existe
- os nomes do `struct` e do `typedef` podem ser distintos
- `p_no` é um ponteiro para um `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:

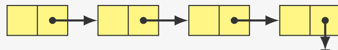
```
1 p_no criar_lista() {
2   return NULL;
3 }
```

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:

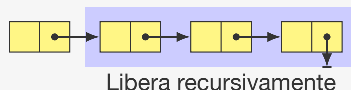
```
1 p_no criar_lista() {
2   return NULL;
3 }
```

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:

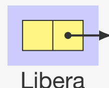
```
p_no criar_lista() {
  return NULL;
}
```

Código no cliente:

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

Destruindo a lista:

```
void destruir_lista(p_no lista) {
  if (lista != NULL) {
    destruir_lista(lista->prox);
    free(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 }
```

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(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(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(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

```c
p_no adicionar_elemento(p_no lista, int x) {
  p_no novo;
  novo = malloc(sizeof(No));
  novo->dado = x;
  novo->prox = lista;
  return novo;
}
```

Código no cliente:

```c
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(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(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(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(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(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(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(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: $\mathrm{O}(1)$

Comparando vetores e listas ligadas

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

Comparando vetores e listas ligadas

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

Comparando vetores e listas ligadas

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

Comparando vetores e listas ligadas

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

Comparando vetores e listas ligadas

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

Comparando vetores e listas ligadas

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

Comparando vetores e listas ligadas

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