MC-202 — Unidade 5
Listas Ligadas

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

Universidade Estadual de Campinas

1º semestre/2017

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

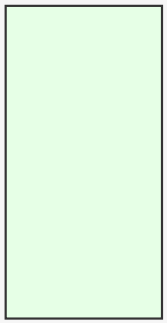
Vetores:

- 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:

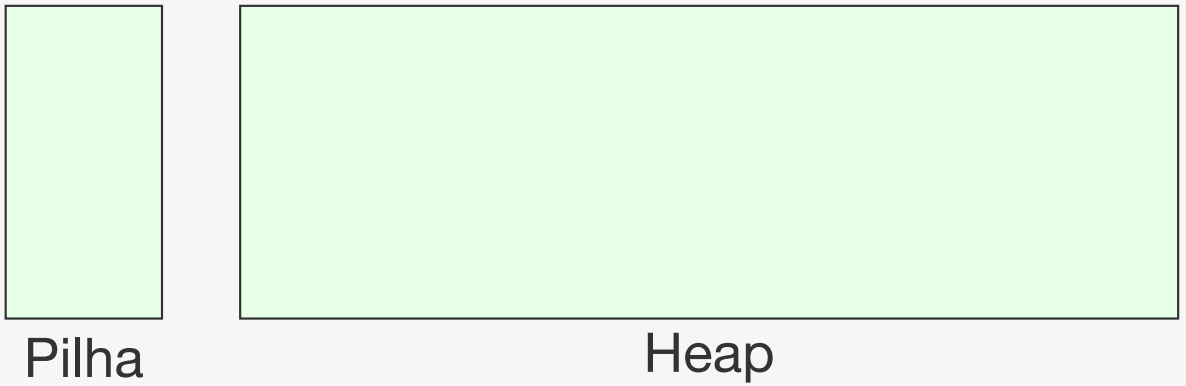
- resolvem o problema do tamanho fixo parcialmente
  - ainda podemos ter um grande desperdício de memória
  - ex: usamos 64GB para armazenar um vetor de 16GB
- inserção/remoção é rápida na maior parte das vezes, mas em algumas operações demora muito
  - ruim para aplicações de “tempo real”

Alternativa - Lista Ligada

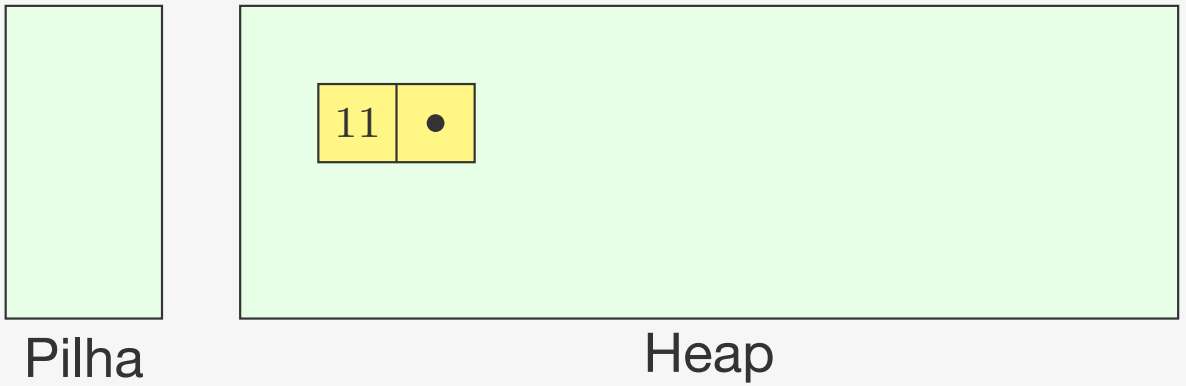


Pilha

Alternativa - Lista Ligada

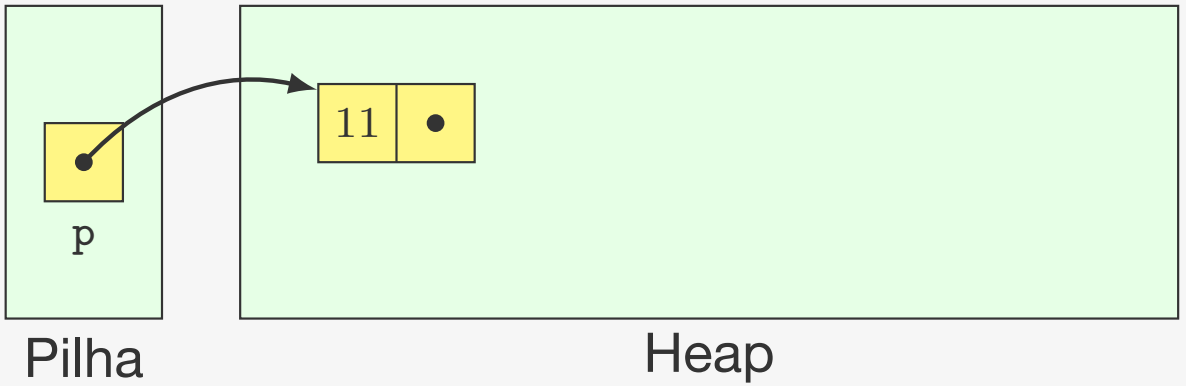


Alternativa - Lista Ligada



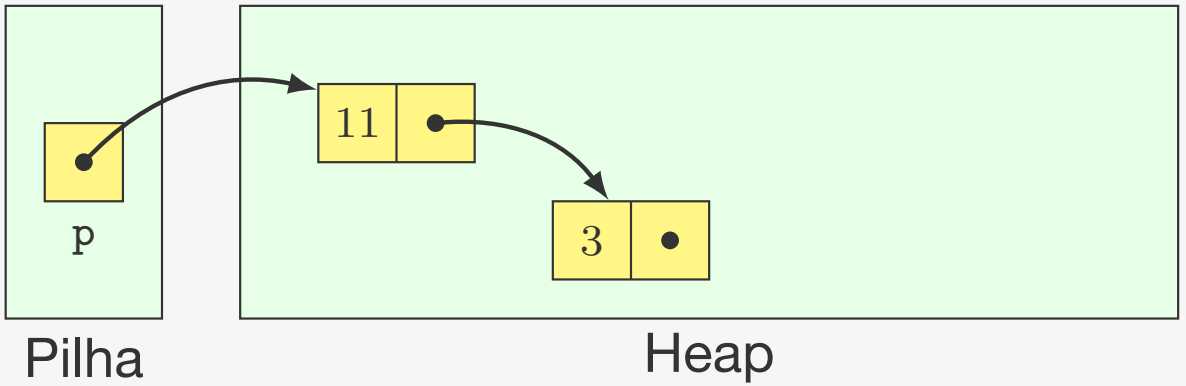
- alocamos memória conforme o necessário

Alternativa - Lista Ligada



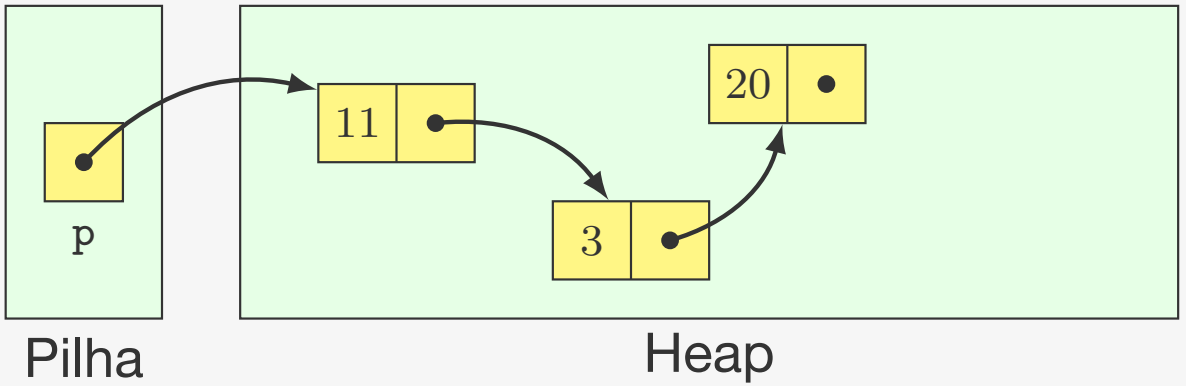
- alocamos memória conforme o necessário
- guardamos um ponteiro para a estrutura em uma variável

Alternativa - Lista Ligada



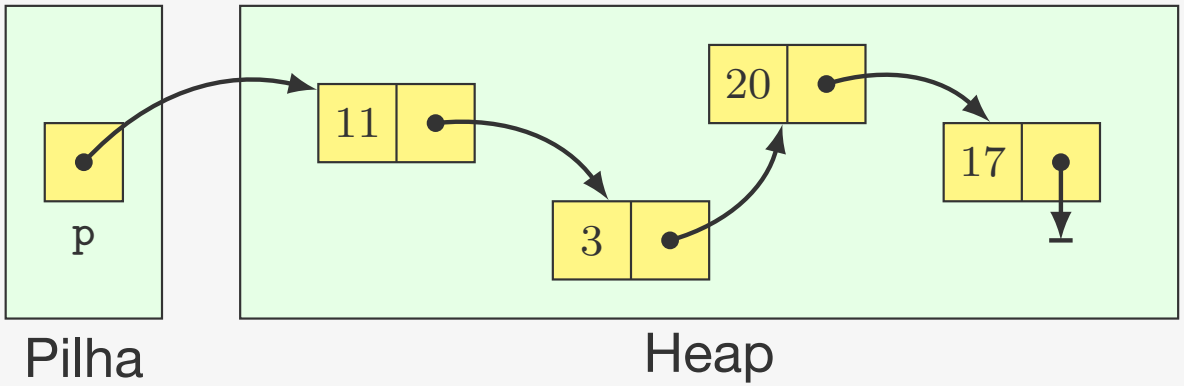
- alocamos memória conforme o necessário
- guardamos um ponteiro para a estrutura em uma variável
- o primeiro nó aponta para o segundo

Alternativa - Lista Ligada



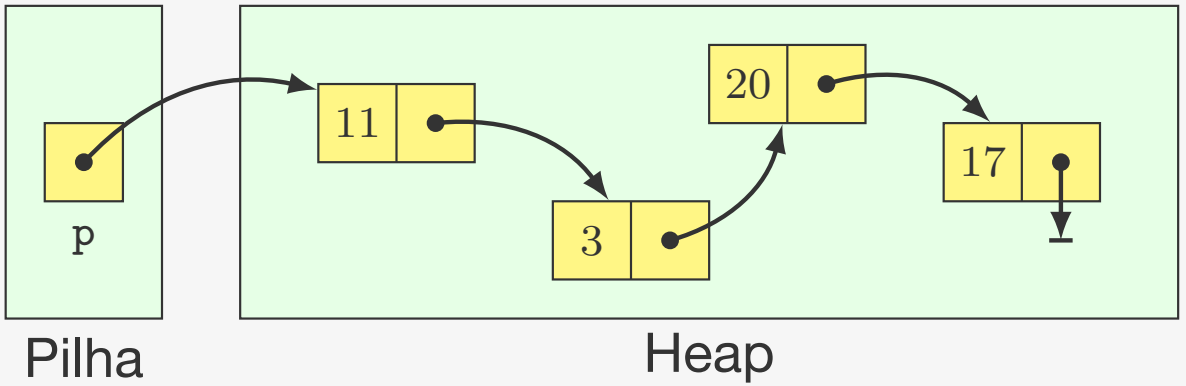
- alocamos memória conforme o necessário
- guardamos um ponteiro para a estrutura em uma variável
- o primeiro nó aponta para o segundo
- o segundo nó aponta para o terceiro

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

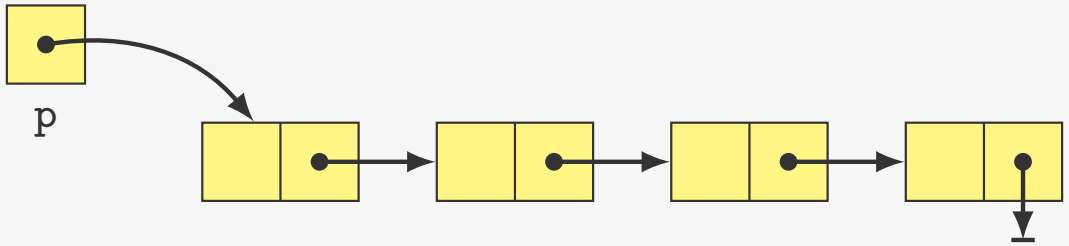
Listas ligadas

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



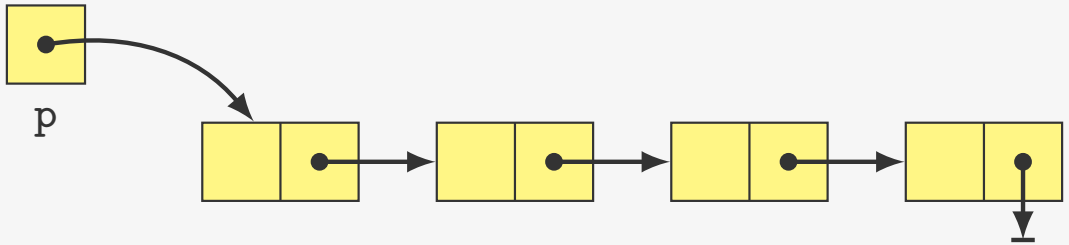
Listas ligadas

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:

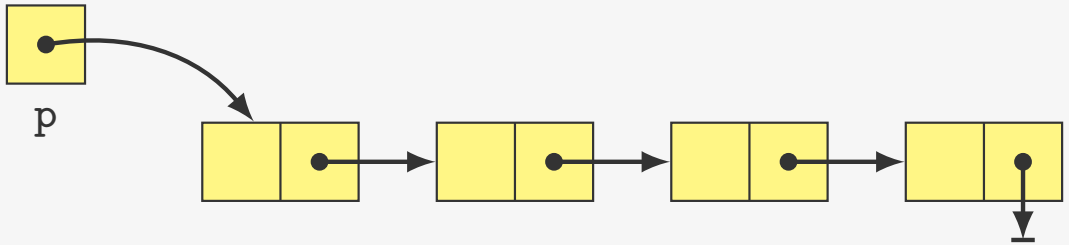
Listas ligadas

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

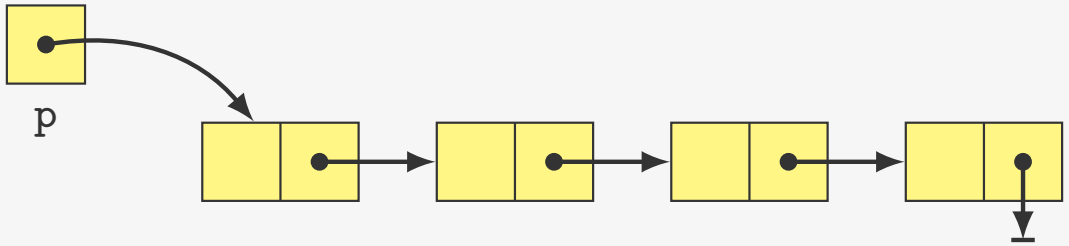
Listas ligadas

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;
```

Implementação em C

Definição do Nó:

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

Observações

Implementação em C

Definição do Nó:

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

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;
```

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

Algumas operações

Inicializa a lista:

Algumas operações

Inicializa a lista:

```
1 void iniciar_lista(No **lista) {
2   *lista = NULL;
3 }
```

Algumas operações

Inicializa a lista:

```
1 void iniciar_lista(No **lista) {
2   *lista = NULL;
3 }
```

Adiciona elemento (no começo da lista):

Algumas operações

Inicializa a lista:

```
1 void iniciar_lista(No **lista) {
2   *lista = NULL;
3 }
```

Adiciona elemento (no começo da lista):

```
1 void adicionar_elemento(No **lista, int x) {
2   No *novo;
3   novo = malloc(sizeof(No));
4   novo->dado = x;
5   novo->prox = *lista;
6   *lista = novo;
7 }
```

Algumas operações

Inicializa a lista:

```
1 void iniciar_lista(No **lista) {
2   *lista = NULL;
3 }
```

Adiciona elemento (no começo da lista):

```
1 void adicionar_elemento(No **lista, int x) {
2   No *novo;
3   novo = malloc(sizeof(No));
4   novo->dado = x;
5   novo->prox = *lista;
6   *lista = novo;
7 }
```

- O ponteiro da lista é passado por referência

Algumas operações

Inicializa a lista:

```
1 void iniciar_lista(No **lista) {
2   *lista = NULL;
3 }
```

Adiciona elemento (no começo da lista):

```
1 void adicionar_elemento(No **lista, int x) {
2   No *novo;
3   novo = malloc(sizeof(No));
4   novo->dado = x;
5   novo->prox = *lista;
6   *lista = novo;
7 }
```

- O ponteiro da lista é passado por referência
- A inserção ocorre em $O(1)$

Algumas operações

Inicializa a lista:

```
1 void iniciar_lista(No **lista) {
2   *lista = NULL;
3 }
```

Adiciona elemento (no começo da lista):

```
1 void adicionar_elemento(No **lista, int x) {
2   No *novo;
3   novo = malloc(sizeof(No));
4   novo->dado = x;
5   novo->prox = *lista;
6   *lista = novo;
7 }
```

- O ponteiro da lista é passado por referência
- A inserção ocorre em $O(1)$
- Deveria verificar se `malloc` não devolve `NULL`

Algumas operações

Inicializa a lista:

```
1 void iniciar_lista(No **lista) {
2   *lista = NULL;
3 }
```

Adiciona elemento (no começo da lista):

```
1 void adicionar_elemento(No **lista, int x) {
2   No *novo;
3   novo = malloc(sizeof(No));
4   novo->dado = x;
5   novo->prox = *lista;
6   *lista = novo;
7 }
```

- O ponteiro da lista é passado por referência
- A inserção ocorre em $O(1)$
- Deveria verificar se `malloc` não devolve `NULL`
  - Teria acabado a memória

Algumas operações

Inicializa a lista:

```
1 void iniciar_lista(No **lista) {
2   *lista = NULL;
3 }
```

Adiciona elemento (no começo da lista):

```
1 void adicionar_elemento(No **lista, int x) {
2   No *novo;
3   novo = malloc(sizeof(No));
4   novo->dado = x;
5   novo->prox = *lista;
6   *lista = novo;
7 }
```

- O ponteiro da lista é passado por referência
- A inserção ocorre em $\mathrm{O}(1)$
- 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(No *lista) {
2   No *atual;
3   for (atual = lista; atual != NULL; atual = atual->prox)
4     printf("%d\n", atual->dado);
5 }
```

Impressão

Impressão iterativa:

```
1 void imprime(No *lista) {
2   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(No *lista) {
2   No *atual;
3   for (atual = lista; atual != NULL; atual = atual->prox)
4     printf("%d\n", atual->dado);
5 }
```

Impressão recursiva:

```
1 void imprime_recursivo(No *lista) {
2   if(lista != NULL){
3     printf("%d\n", lista->dado);
4     imprime(lista->prox);
5   }
6 }
```

Impressão

Impressão iterativa:

```
1 void imprime(No *lista) {
2   No *atual;
3   for (atual = lista; atual != NULL; atual = atual->prox)
4     printf("%d\n", atual->dado);
5 }
```

Impressão recursiva:

```
1 void imprime_recursivo(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(No *lista) {
2   No *atual;
3   for (atual = lista; atual != NULL; atual = atual->prox)
4     printf("%d\n", atual->dado);
5 }
```

Impressão recursiva:

```
1 void imprime_recursivo(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(No *lista) {
2   No *atual;
3   for (atual = lista; atual != NULL; atual = atual->prox)
4     printf("%d\n", atual->dado);
5 }
```

Impressão recursiva:

```
1 void imprime_recursivo(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

Destruir lista - versão recursiva

```
 1 void destruir_lista_rec(No *lista) {
 2   if(lista != NULL) {
 3     destruir_lista_rec(lista->prox);
 4     free(lista);
 5   }
 6 }
 7
 8 void destruir_lista(No **lista) {
 9   destruir_lista_rec(*lista);
10   *lista = NULL;
11 }
```

Destruir lista - versão recursiva

```
 1 void destruir_lista_rec(No *lista) {
 2   if(lista != NULL) {
 3     destruir_lista_rec(lista->prox);
 4     free(lista);
 5   }
 6 }
 7
 8 void destruir_lista(No **lista) {
 9   destruir_lista_rec(*lista);
10   *lista = NULL;
11 }
```

Exercício: faça uma versão iterativa do destruir lista

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   No *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   No *lista;
8   iniciar_lista(&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    No *lista;
8    iniciar_lista(&lista);
9    /*lê números positivos e armazena na lista*/
10   do {
11     scanf("%d", &num);
12     if (num > 0)
13       adicionar_elemento(&lista, num);
14   } while (num > 0);
```

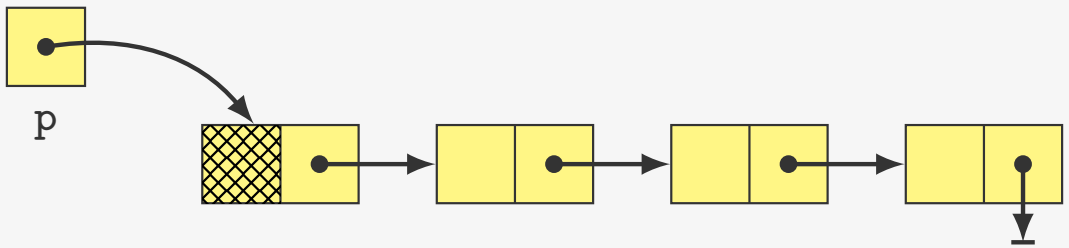
Exemplo - lendo números positivos

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "lista_ligada.h"
4
5  int main() {
6    int num;
7    No *lista;
8    iniciar_lista(&lista);
9    /*lê números positivos e armazena na lista*/
10   do {
11     scanf("%d", &num);
12     if (num > 0)
13       adicionar_elemento(&lista, num);
14   } while (num > 0);
15   imprime(lista); /*(em ordem reversa de inserção)*/
```

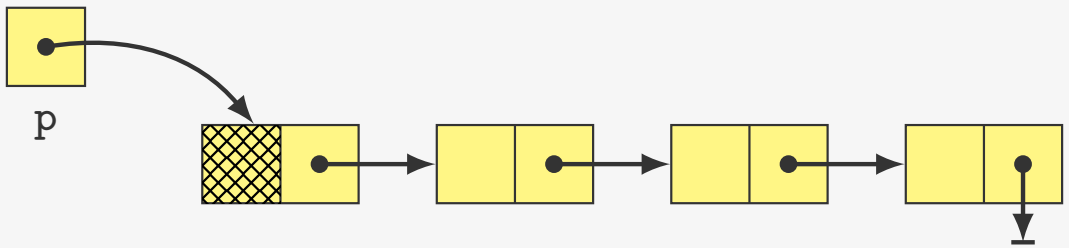
Exemplo - lendo números positivos

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "lista_ligada.h"
4
5  int main() {
6    int num;
7    No *lista;
8    iniciar_lista(&lista);
9    /*lê números positivos e armazena na lista*/
10   do {
11     scanf("%d", &num);
12     if (num > 0)
13       adicionar_elemento(&lista, num);
14   } while (num > 0);
15   imprime(lista); /*(em ordem reversa de inserção)*/
16   destruir_lista(&lista);
17   return 0;
18 }
```

Lista com cabeça (nó *dummy*)

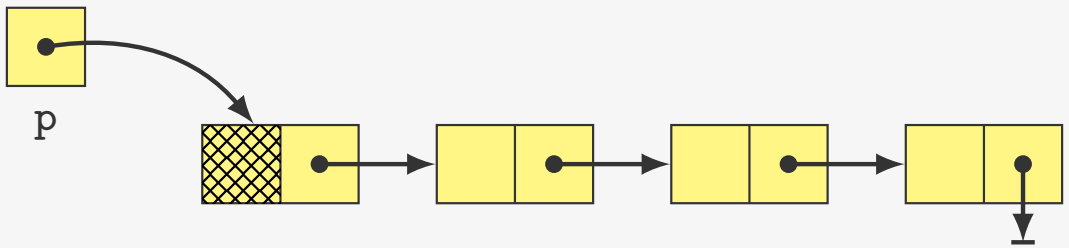


Lista com cabeça (nó *dummy*)



Diferenças para a versão sem cabeça:

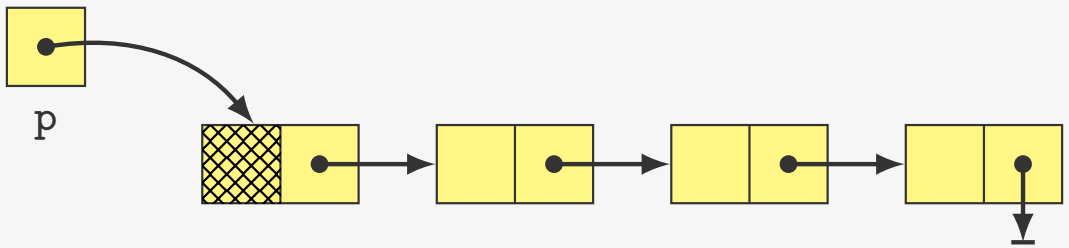
Lista com cabeça (nó *dummy*)



Diferenças para a versão sem cabeça:

- lista sempre aponta para o nó *dummy*

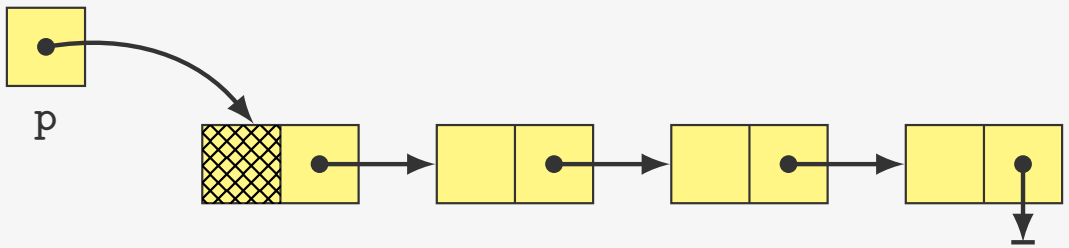
Lista com cabeça (nó *dummy*)



Diferenças para a versão sem cabeça:

- lista sempre aponta para o nó *dummy*
- evita passagem por referência

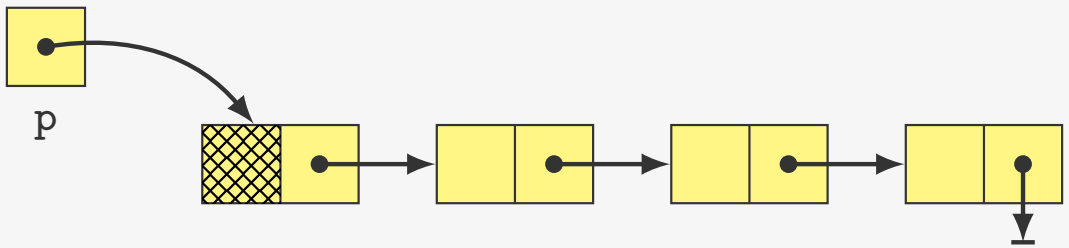
Lista com cabeça (nó *dummy*)



Diferenças para a versão sem cabeça:

- lista sempre aponta para o nó *dummy*
- evita passagem por referência
- mesmo procedimento para adicionar elemento em qualquer posição

Lista com cabeça (nó *dummy*)



Diferenças para a versão sem cabeça:

- lista sempre aponta para o nó *dummy*
- evita passagem por referência
- mesmo procedimento para adicionar elemento em qualquer posição
- ao percorrer tem que ignorar cabeça

Lista com cabeça (nó *dummy*) – operações

Iniciar lista:

Lista com cabeça (nó *dummy*) – operações

Iniciar lista:

```
1 No *iniciar_lista() {
2   No *dummy;
3   dummy = malloc(sizeof(No));
4   dummy->prox = NULL;
5   return dummy;
6 }
```

Lista com cabeça (nó *dummy*) – operações

Iniciar lista:

```
1 No *iniciar_lista() {
2   No *dummy;
3   dummy = malloc(sizeof(No));
4   dummy->prox = NULL;
5   return dummy;
6 }
```

Adicionar elemento depois:

Lista com cabeça (nó *dummy*) – operações

Iniciar lista:

```
1 No *iniciar_lista() {
2   No *dummy;
3   dummy = malloc(sizeof(No));
4   dummy->prox = NULL;
5   return dummy;
6 }
```

Adicionar elemento depois:

```
1 void adicionar_elemento_depois(No *anterior, int x) {
2   No *novo;
3   novo = malloc(sizeof(No));
4   novo->dado = x;
5   novo->prox = anterior->prox;
6   anterior->prox = novo;
7 }
```

Exemplo de aplicação: polinômio

Um polinômio é uma expressão da seguinte forma

Exemplo de aplicação: polinômio

Um polinômio é uma expressão da seguinte forma

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

Exemplo de aplicação: polinômio

Um polinômio é uma expressão da seguinte forma

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

Representação através de lista ligada:

Exemplo de aplicação: polinômio

Um polinômio é uma expressão da seguinte forma

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

Representação através de lista ligada:

```
1 typedef struct Termo {
2     double coeficiente;
3     int expoente;
4     struct Termo *prox;
5 } Termo;
```

Exemplo de aplicação: polinômio

Um polinômio é uma expressão da seguinte forma

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

Representação através de lista ligada:

```
1 typedef struct Termo {
2     double coeficiente;
3     int expoente;
4     struct Termo *prox;
5 } Termo;
```

Exemplo: $2x^5 + x^3 + 2x^2 + 2$

Exemplo de aplicação: polinômio

Um polinômio é uma expressão da seguinte forma

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

Representação através de lista ligada:

```
1 typedef struct Termo {
2     double coeficiente;
3     int expoente;
4     struct Termo *prox;
5 } Termo;
```

Exemplo: $2x^5 + x^3 + 2x^2 + 2$

p

Exemplo de aplicação: polinômio

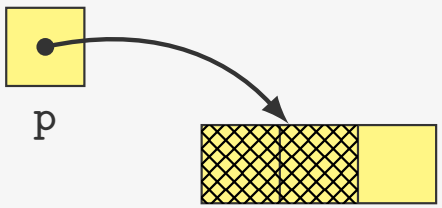
Um polinômio é uma expressão da seguinte forma

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

Representação através de lista ligada:

```
1 typedef struct Termo {
2     double coeficiente;
3     int expoente;
4     struct Termo *prox;
5 } Termo;
```

Exemplo: $2x^5 + x^3 + 2x^2 + 2$



Exemplo de aplicação: polinômio

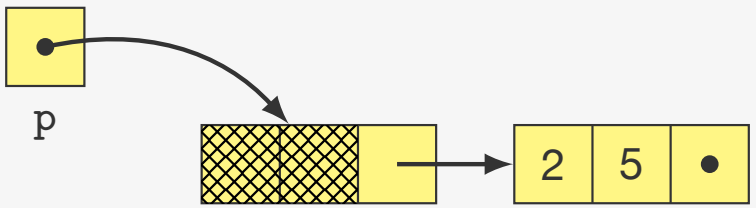
Um polinômio é uma expressão da seguinte forma

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

Representação através de lista ligada:

```
1 typedef struct Termo {
2     double coeficiente;
3     int expoente;
4     struct Termo *prox;
5 } Termo;
```

Exemplo: $2x^5 + x^3 + 2x^2 + 2$



Exemplo de aplicação: polinômio

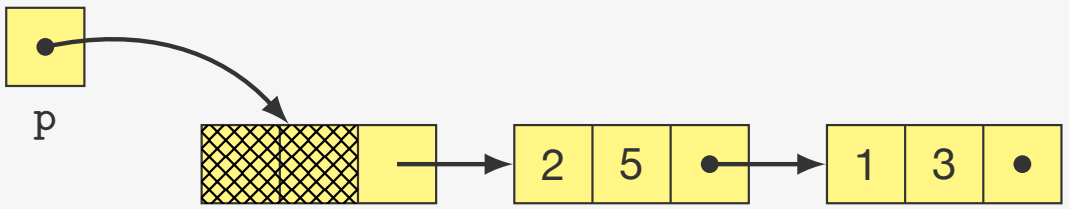
Um polinômio é uma expressão da seguinte forma

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

Representação através de lista ligada:

```
1 typedef struct Termo {
2     double coeficiente;
3     int expoente;
4     struct Termo *prox;
5 } Termo;
```

Exemplo: $2x^5 + x^3 + 2x^2 + 2$



Exemplo de aplicação: polinômio

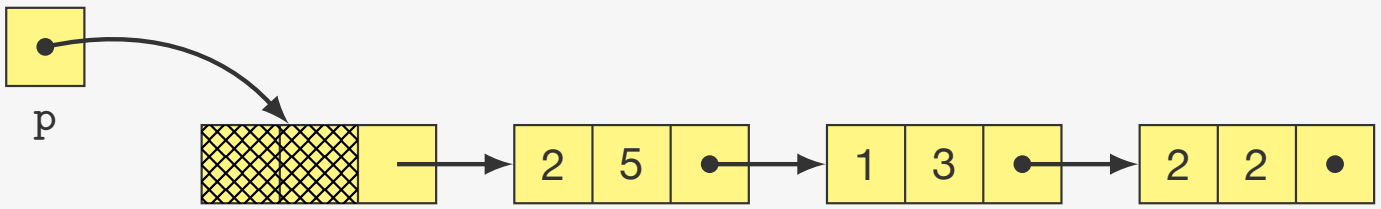
Um polinômio é uma expressão da seguinte forma

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

Representação através de lista ligada:

```
1 typedef struct Termo {
2     double coeficiente;
3     int expoente;
4     struct Termo *prox;
5 } Termo;
```

Exemplo: $2x^5 + x^3 + 2x^2 + 2$



Exemplo de aplicação: polinômio

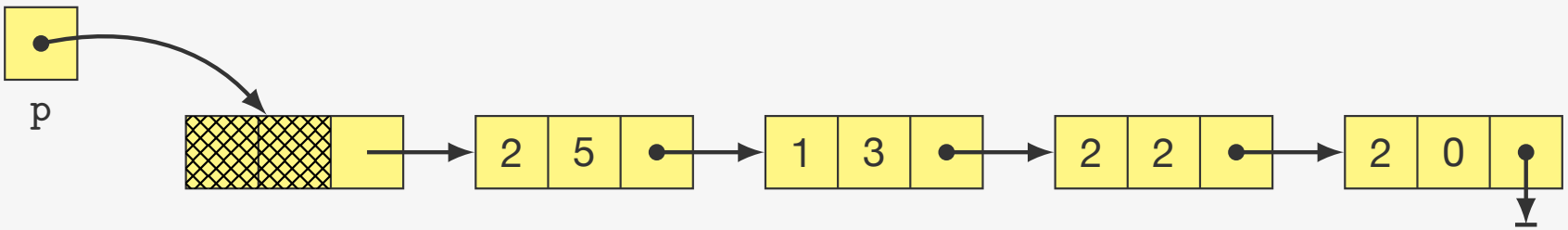
Um polinômio é uma expressão da seguinte forma

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

Representação através de lista ligada:

```
1 typedef struct Termo {
2     double coeficiente;
3     int expoente;
4     struct Termo *prox;
5 } Termo;
```

Exemplo: $2x^5 + x^3 + 2x^2 + 2$

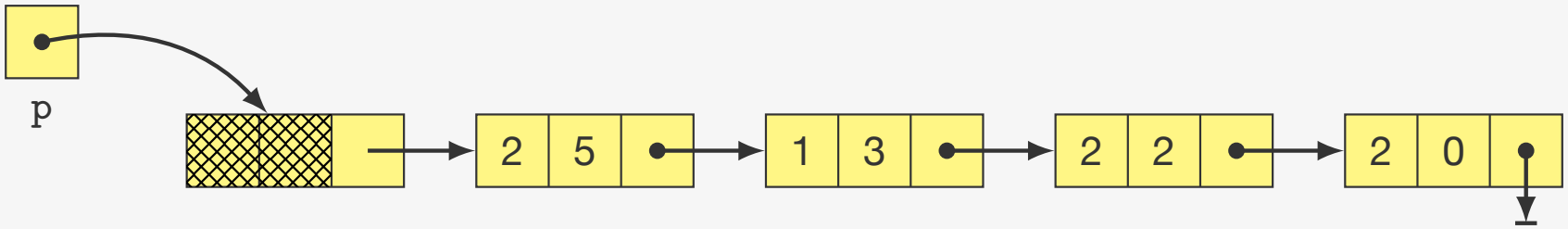


Somando polinômios

Dados dois polinômios p e q, como obter o polinômio r correspondente à soma $p + q$?

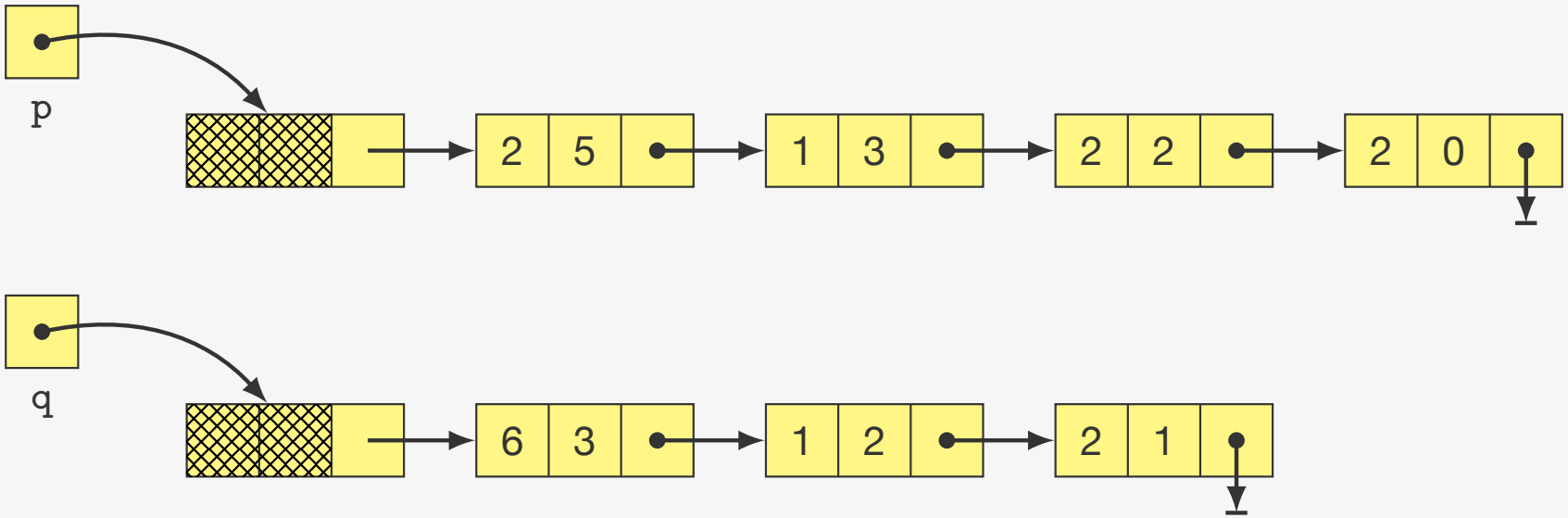
Somando polinômios

Dados dois polinômios p e q, como obter o polinômio r correspondente à soma $p + q$?



Somando polinômios

Dados dois polinômios p e q, como obter o polinômio r correspondente à soma $p + q$?



Lendo um polinômio

```
1 Termo *le_polinomio() {
```

Lendo um polinômio

```
1 Termo *le_polinomio() {
2   int i, n, expoente;
3   double coeficiente;
```

Lendo um polinômio

```
1 Termo *le_polinomio() {
2   int i, n, expoente;
3   double coeficiente;
4   Termo *atual, *p = iniciar_lista();
```

Lendo um polinômio

```
1 Termo *le_polinomio() {
2   int i, n, expoente;
3   double coeficiente;
4   Termo *atual, *p = iniciar_lista();
5   atual = p;
```

Lendo um polinômio

```
1 Termo *le_polinomio() {
2   int i, n, expoente;
3   double coeficiente;
4   Termo *atual, *p = iniciar_lista();
5   atual = p;
6   scanf("%d", &n);
```

Lendo um polinômio

```
1 Termo *le_polinomio() {
2   int i, n, expoente;
3   double coeficiente;
4   Termo *atual, *p = iniciar_lista();
5   atual = p;
6   scanf("%d", &n);
7   for (i = 0; i < n; i++) {
```

Lendo um polinômio

```
1 Termo *le_polinomio() {
2   int i, n, expoente;
3   double coeficiente;
4   Termo *atual, *p = iniciar_lista();
5   atual = p;
6   scanf("%d", &n);
7   for (i = 0; i < n; i++) {
8     scanf("%lf %d", &coeficiente, &expoente);
9     adicionar_elemento_depois(atual, coeficiente, expoente);
```

Lendo um polinômio

```
1  Termo *le_polinomio() {
2    int i, n, expoente;
3    double coeficiente;
4    Termo *atual, *p = iniciar_lista();
5    atual = p;
6    scanf("%d", &n);
7    for (i = 0; i < n; i++) {
8      scanf("%lf %d", &coeficiente, &expoente);
9      adicionar_elemento_depois(atual, coeficiente, expoente);
10     atual = atual->prox;
11   }
```

Lendo um polinômio

```
1  Termo *le_polinomio() {
2    int i, n, expoente;
3    double coeficiente;
4    Termo *atual, *p = iniciar_lista();
5    atual = p;
6    scanf("%d", &n);
7    for (i = 0; i < n; i++) {
8      scanf("%lf %d", &coeficiente, &expoente);
9      adicionar_elemento_depois(atual, coeficiente, expoente);
10     atual = atual->prox;
11   }
12   return p;
13 }
```

Somando polinômios

```
1 Termo *soma_polinomios(Termo *p, Termo *q) {
```

Somando polinômios

```
1 Termo *soma_polinomios(Termo *p, Termo *q) {
2   Termo *r = iniciar_lista();
```

Somando polinômios

```
1 Termo *soma_polinomios(Termo *p, Termo *q) {
2   Termo *r = iniciar_lista();
3   r->prox = soma_polinomios_rec(p->prox,q->prox);
```

Somando polinômios

```
1 Termo *soma_polinomios(Termo *p, Termo *q) {
2   Termo *r = iniciar_lista();
3   r->prox = soma_polinomios_rec(p->prox,q->prox);
4   return r;
5 }
```

Somando polinômios

```
1 Termo *soma_polinomios(Termo *p, Termo *q) {
2   Termo *r = iniciar_lista();
3   r->prox = soma_polinomios_rec(p->prox,q->prox);
4   return r;
5 }
6
7 Termo *soma_polinomios_rec(Termo *p, Termo *q) {
```

Somando polinômios

```
1  Termo *soma_polinomios(Termo *p, Termo *q) {
2    Termo *r = iniciar_lista();
3    r->prox = soma_polinomios_rec(p->prox,q->prox);
4    return r;
5  }
6
7  Termo *soma_polinomios_rec(Termo *p, Termo *q) {
8    Termo *r;
9    if(p == NULL && q == NULL)
10     return NULL;
```

Somando polinômios

```
1  Termo *soma_polinomios(Termo *p, Termo *q) {
2    Termo *r = iniciar_lista();
3    r->prox = soma_polinomios_rec(p->prox,q->prox);
4    return r;
5  }
6
7  Termo *soma_polinomios_rec(Termo *p, Termo *q) {
8    Termo *r;
9    if(p == NULL && q == NULL)
10     return NULL;
11   r = malloc(sizeof(Termo));
```

Somando polinômios

```
1  Termo *soma_polinomios(Termo *p, Termo *q) {
2    Termo *r = iniciar_lista();
3    r->prox = soma_polinomios_rec(p->prox,q->prox);
4    return r;
5  }
6
7  Termo *soma_polinomios_rec(Termo *p, Termo *q) {
8    Termo *r;
9    if(p == NULL && q == NULL)
10     return NULL;
11   r = malloc(sizeof(Termo));
12   if (q == NULL || p->expoente > q->expoente) {
13     r->coeficiente = p->coeficiente;
14     r->expoente = p->expoente;
15     r->prox = soma_polinomios_rec(p->prox, q);
```

Somando polinômios

```
1  Termo *soma_polinomios(Termo *p, Termo *q) {
2    Termo *r = iniciar_lista();
3    r->prox = soma_polinomios_rec(p->prox,q->prox);
4    return r;
5  }
6
7  Termo *soma_polinomios_rec(Termo *p, Termo *q) {
8    Termo *r;
9    if(p == NULL && q == NULL)
10     return NULL;
11   r = malloc(sizeof(Termo));
12   if (q == NULL || p->expoente > q->expoente) {
13     r->coeficiente = p->coeficiente;
14     r->expoente = p->expoente;
15     r->prox = soma_polinomios_rec(p->prox, q);
16   } else if (p == NULL || q->expoente > p->expoente) {
17     r->coeficiente = q->coeficiente;
18     r->expoente = q->expoente;
19     r->prox = soma_polinomios_rec(p, q->prox);
```

Somando polinômios

```
1  Termo *soma_polinomios(Termo *p, Termo *q) {
2    Termo *r = iniciar_lista();
3    r->prox = soma_polinomios_rec(p->prox,q->prox);
4    return r;
5  }
6
7  Termo *soma_polinomios_rec(Termo *p, Termo *q) {
8    Termo *r;
9    if(p == NULL && q == NULL)
10     return NULL;
11   r = malloc(sizeof(Termo));
12   if (q == NULL || p->expoente > q->expoente) {
13     r->coeficiente = p->coeficiente;
14     r->expoente = p->expoente;
15     r->prox = soma_polinomios_rec(p->prox, q);
16   } else if (p == NULL || q->expoente > p->expoente) {
17     r->coeficiente = q->coeficiente;
18     r->expoente = q->expoente;
19     r->prox = soma_polinomios_rec(p, q->prox);
20   } else {
21     r->coeficiente = p->coeficiente + q->coeficiente;
22     r->expoente = p->expoente;
23     r->prox = soma_polinomios_rec(p->prox, q->prox);
24   }
```

Somando polinômios

```
1  Termo *soma_polinomios(Termo *p, Termo *q) {
2    Termo *r = iniciar_lista();
3    r->prox = soma_polinomios_rec(p->prox,q->prox);
4    return r;
5  }
6
7  Termo *soma_polinomios_rec(Termo *p, Termo *q) {
8    Termo *r;
9    if(p == NULL && q == NULL)
10     return NULL;
11   r = malloc(sizeof(Termo));
12   if (q == NULL || p->expoente > q->expoente) {
13     r->coeficiente = p->coeficiente;
14     r->expoente = p->expoente;
15     r->prox = soma_polinomios_rec(p->prox, q);
16   } else if (p == NULL || q->expoente > p->expoente) {
17     r->coeficiente = q->coeficiente;
18     r->expoente = q->expoente;
19     r->prox = soma_polinomios_rec(p, q->prox);
20   } else {
21     r->coeficiente = p->coeficiente + q->coeficiente;
22     r->expoente = p->expoente;
23     r->prox = soma_polinomios_rec(p->prox, q->prox);
24   }
25   return r;
26 }
```

Cliente

```
1 int main() {
2   Termo *p, *q, *r;
```

Cliente

```
1 int main() {
2   Termo *p, *q, *r;
3   p = le_polinomio();
4   imprime_polinomio(p);
5   printf("\n");
```

Cliente

```
1 int main() {
2   Termo *p, *q, *r;
3   p = le_polinomio();
4   imprime_polinomio(p);
5   printf("\n");
6   q = le_polinomio();
7   imprime_polinomio(q);
8   printf("\n");
```

Cliente

```
1 int main() {
2   Termo *p, *q, *r;
3   p = le_polinomio();
4   imprime_polinomio(p);
5   printf("\n");
6   q = le_polinomio();
7   imprime_polinomio(q);
8   printf("\n");
9   r = soma_polinomios(p,q);
```

Cliente

```
 1 int main() {
 2   Termo *p, *q, *r;
 3   p = le_polinomio();
 4   imprime_polinomio(p);
 5   printf("\n");
 6   q = le_polinomio();
 7   imprime_polinomio(q);
 8   printf("\n");
 9   r = soma_polinomios(p,q);
10   imprime_polinomio(r);
11   return 0;
12 }
```

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 k:

Comparando vetores e listas ligadas

- Acesso a posição k:
  - Vetor: $\mathrm{O}(1)$
  - Lista: $\mathrm{O}(k)$ (precisa percorrer a lista)
- Inserção na posição k:
  - Vetor: $\mathrm{O}(n-k)$ (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 k:
  - Vetor: $\mathrm{O}(n-k)$ (precisa mover itens para a direita)
  - Lista: $\mathrm{O}(1)$ (se já tiver o ponteiro para o nó $k-1$)

Comparando vetores e listas ligadas

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

Comparando vetores e listas ligadas

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

Comparando vetores e listas ligadas

- Acesso a posição k:
  - Vetor: $\mathrm{O}(1)$
  - Lista: $\mathrm{O}(k)$ (precisa percorrer a lista)
- Inserção na posição k:
  - Vetor: $\mathrm{O}(n-k)$ (precisa mover itens para a direita)
  - Lista: $\mathrm{O}(1)$ (se já tiver o ponteiro para o nó $k-1$)
- Remoção da posição k:
  - Vetor: $\mathrm{O}(n-k)$ (precisa mover itens para a esquerda)
  - Lista: $\mathrm{O}(1)$ (se já tiver o ponteiro para o nó $k-1$)
- 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 k:
  - Vetor: $\mathrm{O}(n-k)$ (precisa mover itens para a direita)
  - Lista: $\mathrm{O}(1)$ (se já tiver o ponteiro para o nó $k-1$)
- Remoção da posição k:
  - Vetor: $\mathrm{O}(n-k)$ (precisa mover itens para a esquerda)
  - Lista: $\mathrm{O}(1)$ (se já tiver o ponteiro para o nó $k-1$)
- Uso de espaço:
  - Vetor: provavelmente desperdiçará memória

Comparando vetores e listas ligadas

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

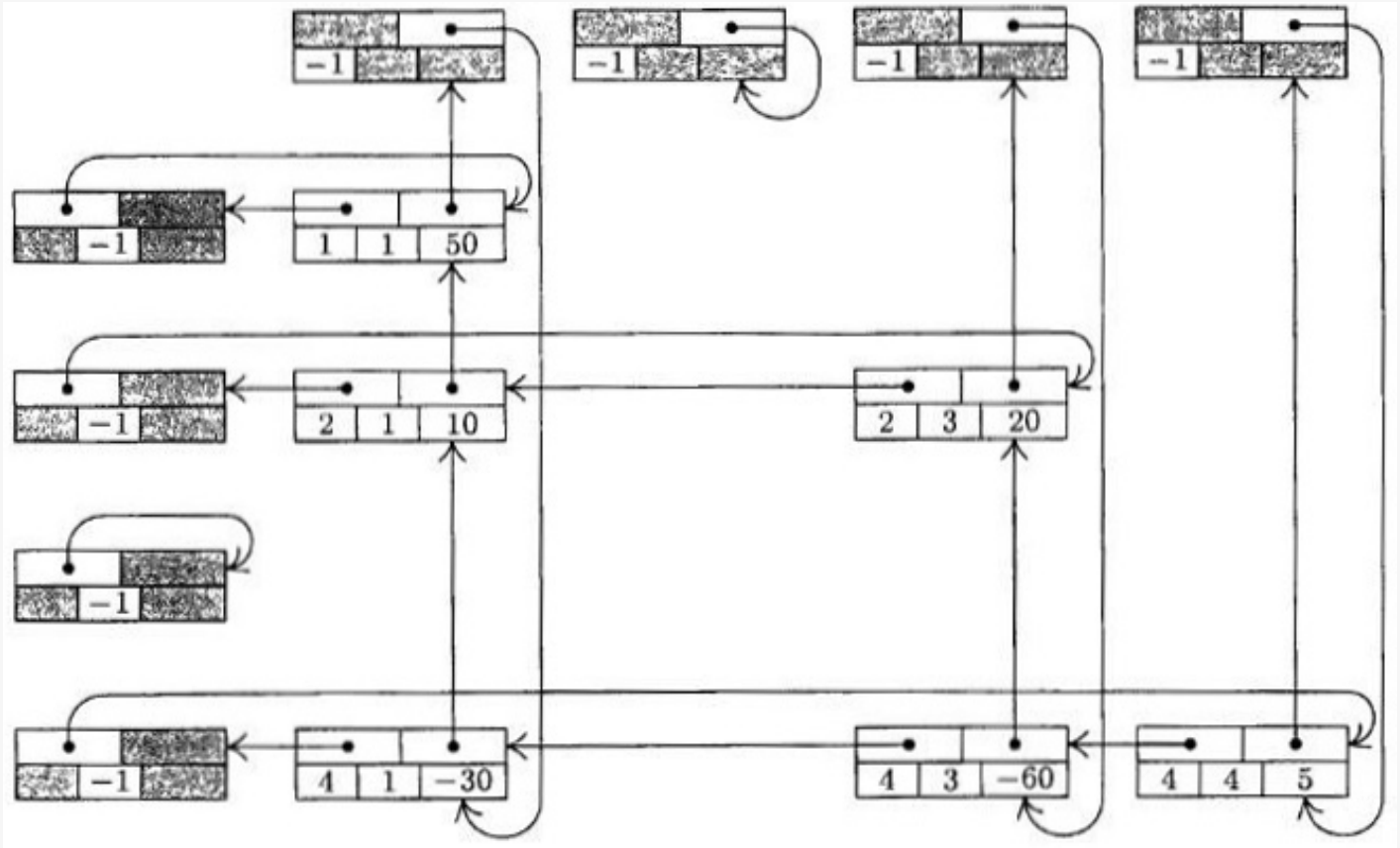
Exercício 2 - Matriz esparsa

Uma matriz $n \times m$ é dita esparsa quando o número de elementos não-nulos é "pequeno" comparado ao número total de elementos nm. Nessa situação, pode ser vantajoso utilizar listas ligadas para representar uma matriz, já que os algoritmos podem supor que todos os elementos não percorridos são nulos. Por exemplo, a matriz a seguir é representada pelas listas "ortogonais" desenhadas no próximo slide:

$$\begin{pmatrix} 50 & 0 & 0 & 0 \\ 10 & 0 & 20 & 0 \\ 0 & 0 & 0 & 0 \\ -30 & 0 & -60 & 5 \end{pmatrix}$$

Defina um novo tipo de nó (`struct`) correspondente a um elemento da matriz esparsa desenhada no próximo slide. Qual a diferença dessa estrutura para as estruturas vistas em sala? (por exemplo, para onde aponta os nós *dummies*?)

Listas ortogonais[1]



[1] Imagem do livro The Art of Computer Programming - I, Knuth.