

Estruturas de Dados

Listas Ligadas

Aula 02

Prof. Felipe A. Louza



- 1 Listas ligadas
- 2 Vetores vs. Listas ligadas
- 3 Lista com nó cabeça (nó dummy)
- 4 Referências

- 1 Listas ligadas
- 2 Vetores vs. Listas ligadas
- 3 Lista com nó cabeça (nó dummy)
- 4 Referências

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 **desperdiçamos memória** ou o **espaço pode acabar**

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

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 **desperdiçamos memória** ou o **espaço pode acabar**

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

Vetores dinâmicos:

- resolvem parcialmente o problema do **tamanho fixo**
 - ex: usamos **64GB** para armazenar um vetor de **16GB**
- inserção/remoção é rápida na maior parte das vezes, mas algumas operações podem **demorar muito**
 - ruim para aplicações de “tempo real”

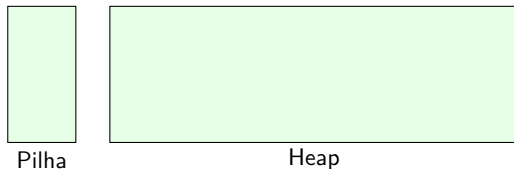
Alternativa - Lista Ligada



Pilha

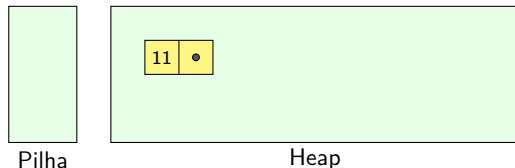
Pilha: variáveis locais criadas na execução de uma função (removidas no final)
Heap: variáveis criadas por alocação dinâmica

Alternativa - Lista Ligada



Pilha: variáveis locais criadas na execução de uma função (removidas no final)
Heap: variáveis criadas por alocação dinâmica

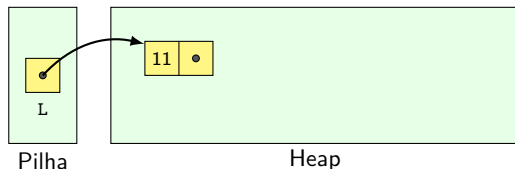
Alternativa - Lista Ligada



- alocamos memória **conforme o necessário**

Pilha: **variáveis locais** criadas na execução de uma função (removidas no final)
Heap: variáveis criadas por **alocação dinâmica**

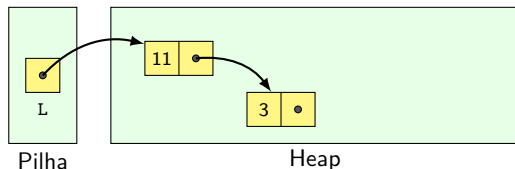
Alternativa - Lista Ligada



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

Pilha: **variáveis locais** criadas na execução de uma função (removidas no final)
Heap: variáveis criadas por **alocação dinâmica**

Alternativa - Lista Ligada

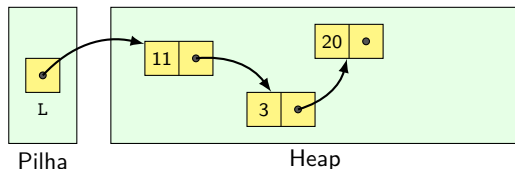


- alocamos memória **conforme o necessário**
- guardamos um **ponteiro** para a estrutura em uma variável
- o primeiro **nó** aponta para o segundo

Pilha: **variáveis locais** criadas na execução de uma função (removidas no final)

Heap: variáveis criadas por **alocação dinâmica**

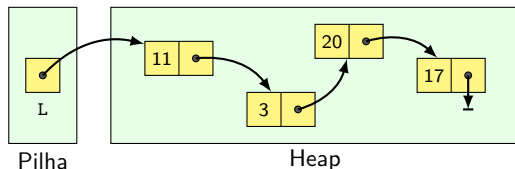
Alternativa - Lista Ligada



- 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

Pilha: **variáveis locais** criadas na execução de uma função (removidas no final)
Heap: variáveis criadas por **alocação dinâmica**

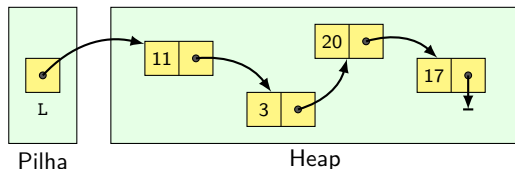
Alternativa - Lista Ligada



- 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

Pilha: variáveis locais criadas na execução de uma função (removidas no final)
Heap: variáveis criadas por alocação dinâmica

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**

Pilha: **variáveis locais** criadas na execução de uma função (removidas no final)

Heap: variáveis criadas por **alocação dinâmica**

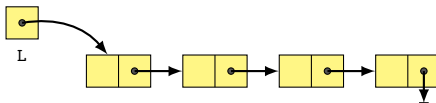
Listas ligadas

Nó: elemento **alocado dinamicamente** que contém

- um ou mais campos de informação (**dados**) e
- um **ponteiro** para o próximo nó

Lista ligada:

- Conjunto de nós ligados entre si de maneira sequencial



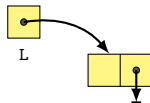
Observação:

- a lista ligada é acessada a partir de **L**
- Os nós **estão tipicamente espalhados** pela memória

Listas ligadas

Definição do Nó:

```
5 typedef struct no {  
6     int dado;  
7     struct no *prox;  
8 } No;
```



Observações:

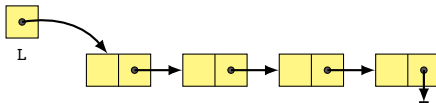
- `typedef` define um apelido `No` para o tipo `struct no`
- deve-se usar `struct no` dentro do registro, porque o apelido ainda não existe
- `No *L;` é um ponteiro para o tipo `No`

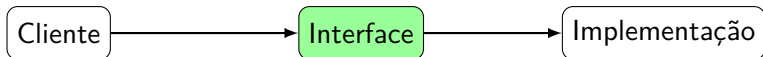
Listas ligadas

Endereço de uma lista ligada:

- O endereço de uma lista é o endereço de sua **primeira célula** (armazenado em **L**).
- A lista está vazia (não tem nenhum nó) quando **L == NULL**.

```
1 No *L = NULL;
```





lista_ligada.h

```
4 //Dados
5 typedef struct no {
6     int dado;
7     struct no *prox;
8 } No;
9
10 //Funções
11 No* criar_lista();
12 void destruir_lista(No **L);
13
14 void imprimir_lista(No *L);
15
```

```
16 //Adicionar
17 void adicionar_inicio(No **L, int x);
18 void adicionar_final(No **L, int x);
19
20 //Remover
21 void remover_inicio(No **L);
22 void remover_final(No **L);
23 void remover_valor(No **L, int x);
24
25 //Buscar
26 int buscar_valor(No *L, int x);
27
```

lista_ligada.h

```
1  #ifndef LISTA_LIGADA_H
2  #define LISTA_LIGADA_H
3
4  //Dados
5  ...
6
7  //Funções
8  ...
9
10 #endif
```

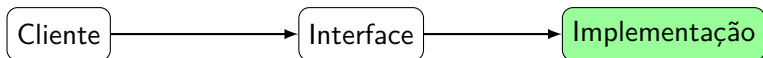
Para evitar **problemas de compilação** com redefinição de tipos/funções¹:

- ❶ **#ifndef** verifica se **LISTA_LIGADA_H** foi definida
- ❷ Inserimos o código da interface entre o **#define** e o **#endif**

Sempre que **lista_ligada.h** for importada, a interface é inserida apenas se **LISTA_LIGADA_H** ainda não foi definida

¹Caso algum outro TAD em uso também utilize **lista_ligada.h**.

Lista ligada - Criar lista



`lista_ligada.c`

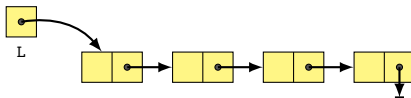
```
5 No* criar_lista() {  
6     return NULL;  
7 }
```

- Vamos apenas retornar o valor `NULL`
- Código do cliente: `No *L = criar_lista();`

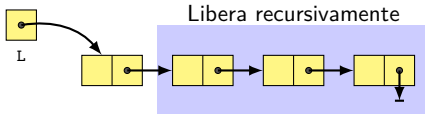
Lista ligada - Destruir lista

lista_ligada.c

```
9 void destruir_iterativo(No **p) { //versão iterativa
10     No *q;
11     while(*p != NULL){
12         q = *p;
13         *p = (*p)->prox;
14         free(q);
15     }
16 }
```



```
18 void destruir_recursivo(No *p) { //versão recursiva
19     if (p != NULL) {
20         destruir_recursivo(p->prox);
21         free(p);
22     }
23 }
```



Lista ligada - Destruir lista

lista_ligada.c

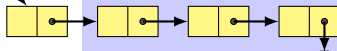
```
9 void destruir_iterativo(No **p) { //versão iterativa
10     No *q;
11     while(*p != NULL){
12         q = *p;
13         *p = (*p)->prox;
14         free(q);
15     }
16 }
```



```
18 void destruir_recursivo(No *p) { //versão recursiva
19     if (p != NULL) {
20         destruir_recursivo(p->prox);
21         free(p);
22     }
23 }
```



Libera recursivamente



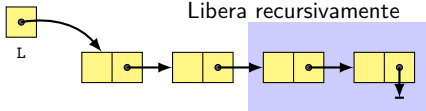
Lista ligada - Destruir lista

lista_ligada.c

```
9 void destruir_iterativo(No **p) { //versão iterativa
10     No *q;
11     while(*p != NULL){
12         q = *p;
13         *p = (*p)->prox;
14         free(q);
15     }
16 }
```



```
18 void destruir_recursivo(No *p) { //versão recursiva
19     if (p != NULL) {
20         destruir_recursivo(p->prox);
21         free(p);
22     }
23 }
```



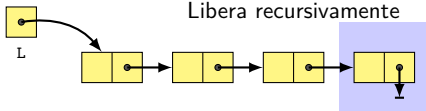
Lista ligada - Destruir lista

lista_ligada.c

```
9 void destruir_iterativo(No **p) { //versão iterativa
10     No *q;
11     while(*p != NULL){
12         q = *p;
13         *p = (*p)->prox;
14         free(q);
15     }
16 }
```



```
18 void destruir_recursivo(No *p) { //versão recursiva
19     if (p != NULL) {
20         destruir_recursivo(p->prox);
21         free(p);
22     }
23 }
```



Lista ligada - Destruir lista

lista_ligada.c

```
9 void destruir_iterativo(No **p) { //versão iterativa
10     No *q;
11     while(*p != NULL){
12         q = *p;
13         *p = (*p)->prox;
14         free(q);
15     }
16 }
```



```
18 void destruir_recursivo(No *p) { //versão recursiva
19     if (p != NULL) {
20         destruir_recursivo(p->prox);
21         free(p);
22     }
23 }
```



Libera recursivamente



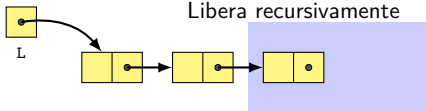
Lista ligada - Destruir lista

lista_ligada.c

```
9 void destruir_iterativo(No **p) { //versão iterativa
10     No *q;
11     while(*p != NULL){
12         q = *p;
13         *p = (*p)->prox;
14         free(q);
15     }
16 }
```



```
18 void destruir_recursivo(No *p) { //versão recursiva
19     if (p != NULL) {
20         destruir_recursivo(p->prox);
21         free(p);
22     }
23 }
```



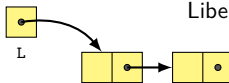
Lista ligada - Destruir lista

lista_ligada.c

```
9 void destruir_iterativo(No **p) { //versão iterativa
10     No *q;
11     while(*p != NULL){
12         q = *p;
13         *p = (*p)->prox;
14         free(q);
15     }
16 }
```



```
18 void destruir_recursivo(No *p) { //versão recursiva
19     if (p != NULL) {
20         destruir_recursivo(p->prox);
21         free(p);
22     }
23 }
```



Libera recursivamente




Lista ligada - Destruir lista

lista_ligada.c

```

9 void destruir_iterativo(No **p) { //versão iterativa
10     No *q;
11     while(*p != NULL){
12         q = *p;
13         *p = (*p)->prox;
14         free(q);
15     }
16 }

```




The diagram shows a yellow square node labeled 'L' below it. Inside the square is a black dot representing a pointer, with an arrow pointing to the right towards the text 'I'.



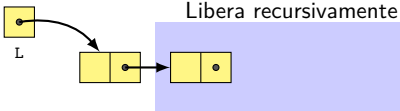
```

18 void destruir_recurso(No *p) { //versão recursiva
19     if (p != NULL) {
20         destruir_recurso(p->prox);
21         free(p);
22     }
23 }

```



The diagram shows a pointer 'L' pointing to a node. The node's 'prox' field points to the next node, which is part of a larger structure labeled 'L'.



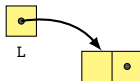
Lista ligada - Destruir lista

lista_ligada.c

```
9 void destruir_iterativo(No **p) { //versão iterativa
10     No *q;
11     while(*p != NULL){
12         q = *p;
13         *p = (*p)->prox;
14         free(q);
15     }
16 }
```



```
18 void destruir_recursivo(No *p) { //versão recursiva
19     if (p != NULL) {
20         destruir_recursivo(p->prox);
21         free(p);
22     }
23 }
```



Libera recursivamente

Lista ligada - Destruir lista

lista_ligada.c

```
9 void destruir_iterativo(No **p) { //versão iterativa
10     No *q;
11     while(*p != NULL){
12         q = *p;
13         *p = (*p)->prox;
14         free(q);
15     }
16 }
```



```
18 void destruir_recursivo(No *p) { //versão recursiva
19     if (p != NULL) {
20         destruir_recursivo(p->prox);
21         free(p);
22     }
23 }
```



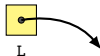
Lista ligada - Destruir lista

lista_ligada.c

```
9 void destruir_iterativo(No **p) { //versão iterativa
10     No *q;
11     while(*p != NULL){
12         q = *p;
13         *p = (*p)->prox;
14         free(q);
15     }
16 }
```



```
18 void destruir_recursivo(No *p) { //versão recursiva
19     if (p != NULL) {
20         destruir_recursivo(p->prox);
21         free(p);
22     }
23 }
```



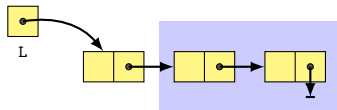
```
25 void destruir_lista(No **p){
26     destruir_iterativo(p);
27     //destruir_recursivo(*p); *p = NULL;
28 }
```

Lista ligada - Imprimir

lista_ligada.c

```
59 void imprimir_lista(No *p) { //versão iterativa  
60     No* q;  
61     for (q = p; q != NULL; q = q->prox) printf("%d -> ", q->dado);  
62     printf("NULL\n");  
63 }
```

```
65 void imprimir_recursivo(No *p) {  
66     if (p != NULL){  
67         printf("%d -> ", p->dado);  
68         imprimir_recursivo(p->prox);  
69     }  
70     else printf("NULL\n");  
71 }
```

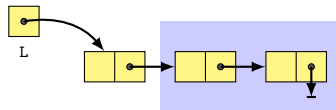


Lista ligada - Imprimir

lista_ligada.c

```
59 void imprimir_lista(No *p) { //versão iterativa
60     No* q;
61     for (q = p; q != NULL; q = q->prox) printf("%d -> ", q->dado);
62     printf("NULL\n");
63 }
```

```
65 void imprimir_recursivo(No *p) {
66     if (p != NULL){
67         printf("%d -> ", p->dado);
68         imprimir_recursivo(p->prox);
69     }
70     else printf("NULL\n");
71 }
```



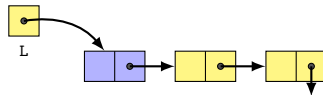
Algoritmos recursivos são, em geral, **mais elegantes** e simples

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

Lista ligada - Adicionar no início

lista_ligada.c

```
32 void adicionar_inicio(No **p, int x) {// p recebe &L
33     No *q;
34     q = (No*) malloc(sizeof(No));
35     if(q==NULL){
36         perror("malloc");
37         exit(EXIT_FAILURE);
38     }
39     q->dado = x;
40     q->prox = *p;
41     *p = q;
42 }
```

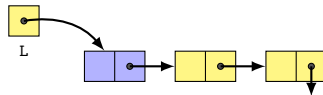


- A inserção no início é feita em $O(1)$

Lista ligada - Adicionar no início

lista_ligada.c

```
32 void adicionar_inicio(No **p, int x) {// p recebe &L
33     No *q;
34     q = (No*) malloc(sizeof(No));
35     if(q==NULL){
36         perror("malloc");
37         exit(EXIT_FAILURE);
38     }
39     q->dado = x;
40     q->prox = *p;
41     *p = q;
42 }
```

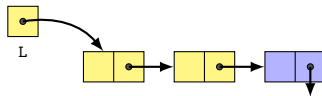


- A inserção no início é feita em $O(1)$
- Devemos sempre verificar se `malloc` não devolve `NULL`
 - Poderia ter acabado a memória
 - Será omitido, mas precisa ser tratado na prática

Lista ligada - Adicionar no final

lista_ligada.c

```
44 void adicionar_final(No **p, int x) {// p recebe &L
45     No *aux, *q;
46     q = (No*) malloc(sizeof(No));
47     q->dado = x;
48     q->prox = NULL;
49     if (*p == NULL) *p = q;
50     else {
51         aux = *p;
52         while(aux->prox != NULL) aux = aux->prox;
53         aux->prox = q;
54     }
55 }
```



- A inserção no final é feita em $O(n)$
 - Uma alternativa é armazenar um ponteiro para o **último da lista** (próxima aula)



exemplo1.c

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

Como compilar?

Teremos três arquivos diferentes:

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

Vamos compilar por partes:

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

Makefile

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

```
1 CFLAGS= -Wall -Werror
2
3 all: exemplo1
4
5 exemplo1: exemplo1.c lista_ligada.o
6         gcc $^ -o $@
7
8 #regra genérica
9 %.o: %.c %.h
10        gcc $(CFLAGS) -c $<
```

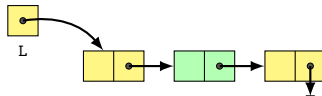
Algumas novidades:

- `$^` representa todas as dependências da regra, e `$@` o alvo
- Regras genéricas:
 - Para cada `*.o` criamos uma regra com dependências `*.c` e `*.h`
 - `$<` representa a primeira dependência

Lista ligada - Buscar na lista

lista_ligada.c

```
74 int buscar_valor(No *p, int x) {// q recebe L
75     while(p != NULL){
76         if(p->dado == x) return 1; //true!
77         p = p->prox;
78     }
79     return 0;//false == não encontrou
80 }
```



- A busca realiza no pior caso $O(n)$ comparações

Lista ligada - Remover no início

lista_ligada.c

```
93 void remover_inicio(No **p) {// p recebe &L  
94     No* q = *p;  
95     if(q==NULL) return; //lista vazia  
96     *p = q->prox;  
97     free(q);  
98 }
```

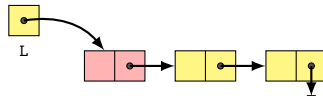


- A remoção no início é feita em $O(1)$

Lista ligada - Remover no início

lista_ligada.c

```
93 void remover_inicio(No **p) {// p recebe &L  
94     No* q = *p;  
95     if(q==NULL) return; //lista vazia  
96     *p = q->prox;  
97     free(q);  
98 }
```



- A remoção no início é feita em $O(1)$

Lista ligada - Remover no final

lista_ligada.c

```
100 void remover_final(No **p) {// p recebe &L
101     No* q = *p;
102     if(q==NULL) return; //lista vazia
103     if(q->prox==NULL){ //apenas 1 elemento
104         *p = NULL;
105         free(q);
106         return;
107     }
108     while (q->prox->prox != NULL) q = q->prox;
109     free(q->prox);
110     q->prox = NULL;
111 }
```

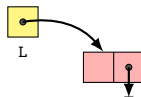


- A remoção no final é feita em $O(n)$

Lista ligada - Remover no final

lista_ligada.c

```
100 void remover_final(No **p) {// p recebe &L
101     No* q = *p;
102     if(q==NULL) return; //lista vazia
103     if(q->prox==NULL){ //apenas 1 elemento
104         *p = NULL;
105         free(q);
106         return;
107     }
108     while (q->prox->prox != NULL) q = q->prox;
109     free(q->prox);
110     q->prox = NULL;
111 }
```

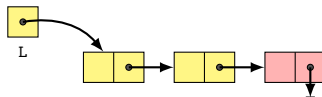


- A remoção no final é feita em $O(n)$

Lista ligada - Remover no final

lista_ligada.c

```
100 void remover_final(No **p) {// p recebe &L
101     No* q = *p;
102     if(q==NULL) return; //lista vazia
103     if(q->prox==NULL){ //apenas 1 elemento
104         *p = NULL;
105         free(q);
106         return;
107     }
108     while (q->prox->prox != NULL) q = q->prox;
109     free(q->prox);
110     q->prox = NULL;
111 }
```

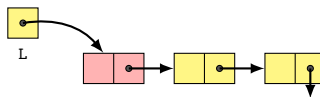


- A remoção no final é feita em $O(n)$

Lista ligada - Remover valor

lista_ligada.c

```
113 void remover_valor(No **p, int v) {// p recebe &L
114     No* q = *p;
115     if(q==NULL) return; //lista vazia
116     if(q->dado==v){ //encontrou no 1o elemento
117         *p = q->prox;
118         free(q);
119         return;
120     }
121     while (q->prox != NULL){
122         if(q->prox->dado==v) break;
123         q = q->prox;
124     }
125     if(q->prox==NULL) return; //não encontrou o dado
126     No* tmp = q->prox;
127     q->prox = tmp->prox;
128     free(tmp);
129 }
```

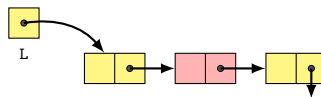


- A remoção no final é feita em $O(n)$

Lista ligada - Remover valor

lista_ligada.c

```
113 void remover_valor(No **p, int v) {// p recebe &L
114     No* q = *p;
115     if(q==NULL) return; //lista vazia
116     if(q->dado==v){ //encontrou no 1o elemento
117         *p = q->prox;
118         free(q);
119         return;
120     }
121     while (q->prox != NULL){
122         if(q->prox->dado==v) break;
123         q = q->prox;
124     }
125     if(q->prox==NULL) return; //não encontrou o dado
126     No* tmp = q->prox;
127     q->prox = tmp->prox;
128     free(tmp);
129 }
```

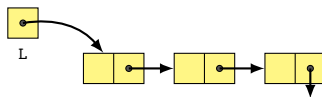


- A remoção no final é feita em $O(n)$

Lista ligada - Remover valor

lista_ligada.c

```
113 void remover_valor(No **p, int v) {// p recebe &L
114     No* q = *p;
115     if(q==NULL) return; //lista vazia
116     if(q->dado==v){ //encontrou no 1o elemento
117         *p = q->prox;
118         free(q);
119         return;
120     }
121     while (q->prox != NULL){
122         if(q->prox->dado==v) break;
123         q = q->prox;
124     }
125     if(q->prox==NULL) return; //não encontrou o dado
126     No* tmp = q->prox;
127     q->prox = tmp->prox;
128     free(tmp);
129 }
```



- A remoção no final é feita em $O(n)$



exemplo2.c

```
1  #include <stdio.h>
2  #include "lista_ligada.h"
3
4  int main() {
5      int i, num;
6      No *L = criar_lista();
7      scanf("%d", &num);
8      for(i=1; i<=num; i++) adicionar_final(&L, i);
9      while(L!=NULL){
10         imprimir_lista(L);
11         scanf("%d", &num);
12         remover_valor(&L, num);
13     }
14     destruir_lista(&L);
15     return 0;
16 }
```


- 1 Listas ligadas
- 2 Vetores vs. Listas ligadas
- 3 Lista com nó cabeça (nó dummy)
- 4 Referências

Vetores vs. Listas ligadas

	Vetores	Listas Ligadas
Inserção no início	$O(n)$	$O(1)$
Inserção no final	$O(1)$	$O(n)$
Remoção no início	$O(n)$	$O(1)$
Remoção no final	$O(1)$	$O(n)$
Busca	$O(n)$	$O(n)$

Uso de espaço:

- Vetor: provavelmente desperdiçará memória
- Lista: não desperdiça memória, mas cada elemento consome mais memória por causa do ponteiro

Vetores vs. Listas ligadas

	Vetores	Listas Ligadas
Inserção no início	$O(n)$	$O(1)$
Inserção no final	$O(1)$	$O(n)$
Remoção no início	$O(n)$	$O(1)$
Remoção no final	$O(1)$	$O(n)$
Busca	$O(n)$	$O(n)$

Uso de espaço:

- Vetor: provavelmente **desperdiçará memória**
- Lista: não desperdiça memória, mas cada elemento **consome mais memória** por causa do ponteiro

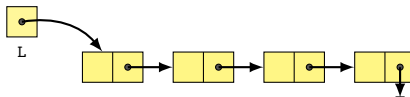
Qual é melhor?

- **depende do problema**, do algoritmo e da implementação

Vetores vs. Listas ligadas

Quando utilizamos **listas ligadas**?

- Em geral, quando **não sabemos antecipadamente** que tamanho a coleção pode alcançar.
- Permitem **inserções/remoções** de nós em qualquer posição.
- Não permitem **acesso direto** a um nó.



Comparando vetores e listas ligadas

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

Busca binária?

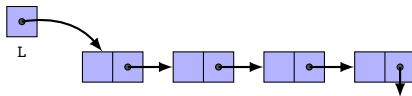
- Com **vetores ordenados** podemos fazer busca binária em $O(\lg n)$

Comparando vetores e listas ligadas

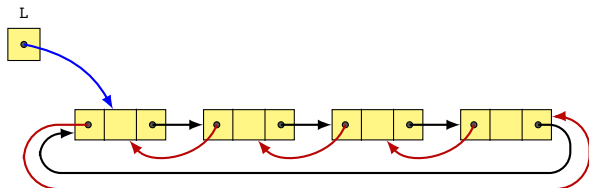
0	1	2	3	4	5	6	7	8	9
9	3	7	6	2	5	?	?	?	?

Busca binária?

- Com **vetores ordenados** podemos fazer busca binária em $O(\lg n)$
- Com **listas ordenadas** a busca ainda leva tempo $O(n)$



Comparando vetores e listas ligadas



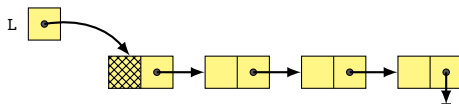
Podemos melhorar a **inserção/remoção** em **listas ligadas** para $O(1)$

- Variações de listas (próxima aula)
- Em geral, utilizamos mais memória para isso

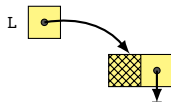
- 1 Listas ligadas
- 2 Vetores vs. Listas ligadas
- 3 Lista com nó cabeça (nó dummy)**
- 4 Referências

Variações - Listas com nó cabeça

Lista com cabeça:



Lista com cabeça **vazia**:

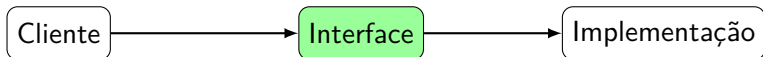


Vantagem:

- O código da inserção e remoção são **mais simples**

Desvantagem:

- Pequeno **gasto de memória** e precisamos **ignorar** o nó dummy (podemos guardar o **tamanho da lista** nesse nó).



lista_com_cabeca.h

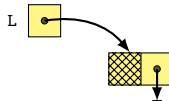
```
1  #ifndef LISTA_HEAD_H
2  #define LISTA_HEAD_H
3
4  //Dados
5  typedef struct no {
6      int dado;
7      struct no *prox;
8  } No;
9
10 //Funções
11 No* criar_lista();
12 void destruir_lista(No **L);
13
14 //Imprimir
15 void imprimir_lista(No *L);
16
```

```
17 //Adicionar
18 void adicionar_inicio(No *L, int x);
19 void adicionar_final(No *L, int x);
20
21 //Remover
22 void remover_inicio(No *L);
23 void remover_final(No *L);
24 void remover_valor(No *L, int x);
25
26 //Buscar
27 int buscar_valor(No *L, int x);
28
29 //Extra
30 int tamanho_lista(No *L);
31
32 #endif
```

Lista com cabeça – Criar lista

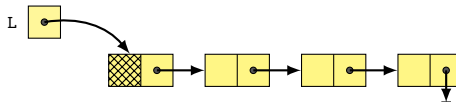
lista_com_cabeca.c

```
5 No* criar_lista() {  
6     No *q = (No*) malloc(sizeof(No));  
7     q->dado = 0;  
8     q->prox = NULL;  
9     return q;  
10 }
```



- Vamos apenas retornar o endereço do nó dummy
- Código do cliente: `No *L = criar_lista();`

Lista com cabeça – Imprimir

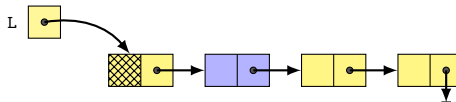


`lista_com_cabeca.c`

```
56 void imprimir_lista(No *p) { //versão iterativa
57     No* q;
58     for (q = p->prox; q != NULL; q = q->prox) printf("%d -> ", q->dado);
59     printf("NULL\n");
60 }
```

- Precisamos apenas ignorar o primeiro nó

Lista com cabeça – Adicionar no início

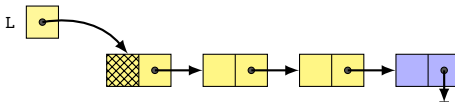


lista_com_cabeca.c

```
35 void adicionar_inicio(No *p, int x) {// p recebe L
36     No* q = (No*) malloc(sizeof(No));
37     q->dado = x;
38     q->prox = p->prox;
39     p->prox = q;
40     p->dado++;
41 }
```

- A inserção no início é feita em $O(1)$
- Ignoramos o primeiro nó
- Incrementamos o contador no nó dummy

Lista com cabeça – Adicionar no final

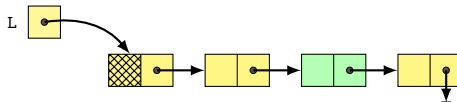


lista_com_cabeca.c

```
43 void adicionar_final(No *p, int x) {// p recebe L
44     No *q, *aux;
45     q = (No*) malloc(sizeof(No));
46     q->dado = x;
47     q->prox = NULL;
48     aux = p;
49     while (aux->prox != NULL) aux = aux->prox;
50     aux->prox = q;
51     p->dado++;
52 }
```

- Implementação mais simples, evita passagem por referência
- Custo computacional: $O(n)$

Lista com cabeça – Buscar no lista

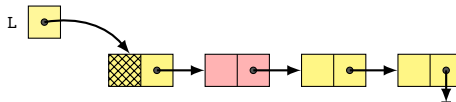


`lista_com_cabeca.c`

```
63 int buscar_valor(No *p, int x) {// q recebe L
64     p = p->prox;
65     while(p != NULL){
66         if(p->dado == x) return 1; //true!
67         p = p->prox;
68     }
69     return 0;//false == não encontrou
70 }
```

- Precisamos apenas ignorar o primeiro nó
- Custo computacional: $O(n)$

Lista com cabeça – Remover do início

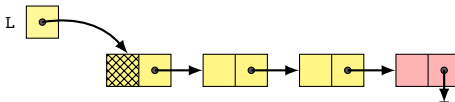


lista_com_cabeca.c

```
74 void remover_inicio(No *p) {// p recebe L
75     No* q = p->prox;
76     if(q==NULL) return; //lista vazia
77     p->prox = q->prox;
78     free(q);
79     p->dado--;
80 }
```

- Decrementados o contador no nó dummy
- Custo computacional: $O(1)$

Lista com cabeça – Remover do final

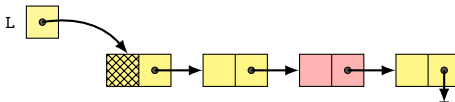


lista_com_cabeca.c

```
82 void remover_final(No *p) {// p recebe L
83     No* q = p;
84     if(q->prox!=NULL){
85         while (q->prox->prox != NULL)
86             q = q->prox;
87         No* aux = q->prox;
88         q->prox = NULL;
89         free(aux);
90     }
91     p->dado--;
92 }
```

- Custo computacional: $O(n)$

Lista com cabeça – Remover valor

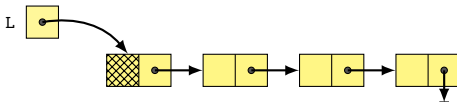


lista_com_cabeca.c

```
94 void remover_valor(No *p, int v) {// p recebe &L
95     No* q = p;
96     while (q->prox != NULL){
97         if(q->prox->dado==v) break;
98         q = q->prox;
99     }
100     if(q->prox==NULL) return; //não encontrou o dado
101     No* tmp = q->prox;
102     q->prox = tmp->prox;
103     free(tmp);
104     p->dado--;
105 }
```

- Custo computacional: $O(n)$

Lista com cabeça – Tamanho da lista



lista_com_cabeca.c

```
107 int tamanho_lista(No *p){  
108     return p->dado;  
109 }
```

- Não precisamos percorrer a lista para saber o tamanho
- Custo computacional: $O(1)$



exemplo3.c

```
1  #include <stdio.h>
2  #include "lista_com_cabeca.h"
3
4  int main(){
5      int i, num;
6      No *L = criar_lista();
7      scanf("%d", &num);
8      for(i=1; i<=num; i++) adicionar_inicio(L, i);
9      imprimir_lista(L);
10     while(tamanho_lista(L)>0){
11         scanf("%d", &i); remover_valor(L, i);
12         imprimir_lista(L);
13     }
14     destruir_lista(&L);
15     return 0;
16 }
```

Listas com nó cabeça vs. Listas ligadas (simples)

	Listas com nó cabeça	Listas Ligadas
Inserção no início	$O(1)$	$O(1)$
Inserção no final	$O(n)$	$O(n)$
Remoção no início	$O(1)$	$O(1)$
Remoção no final	$O(n)$	$O(n)$
Busca	$O(n)$	$O(n)$

Comparação:

- Mesmo custo computacional!
- Implementação **mais simples**, com um **pequeno overhead** do nó dummy

Dúvidas?

- 1 Listas ligadas
- 2 Vetores vs. Listas ligadas
- 3 Lista com nó cabeça (nó dummy)
- 4 Referências

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