# Estruturas de Dados

## Variações de Listas

**Aula 03**

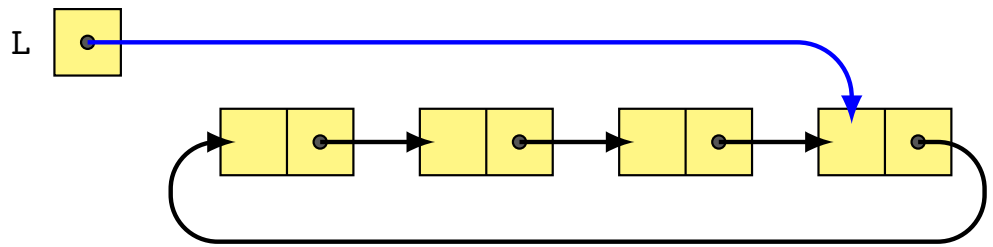
Prof. Felipe A. Louza

# Roteiro

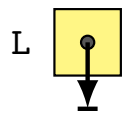
# Roteiro

# Variações - Listas circulares

Lista circular:




Lista circular vazia:




Diferenças:

- O último elemento aponta para o primeiro.
- Podemos varrer a lista a partir de qualquer ponto e voltar ao início.
- A lista sempre aponta para o último elemento, com isso, para acessar o primeio: `L->prox`

# TAD - Interface


## lista_circular.h

```c
#ifndef LISTA_CIRCULAR_H
#define LISTA_CIRCULAR_H

//Dados
typedef struct no {
  int dado;
  struct no *prox;
} No;

//Funções
No* criar_lista();
void destruir_lista(No **L);

//Imprimir
void imprimir_lista(No *L);

//Adicionar
void adicionar_inicio(No **L, int x);
void adicionar_final(No **L, int x);

//Remover
void remover_inicio(No **L);
void remover_final(No **L);
void remover_valor(No **L, int x);

//Buscar
int buscar_valor(No *L, int x);

//Extra
int tamanho_lista(No *L);

#endif
```

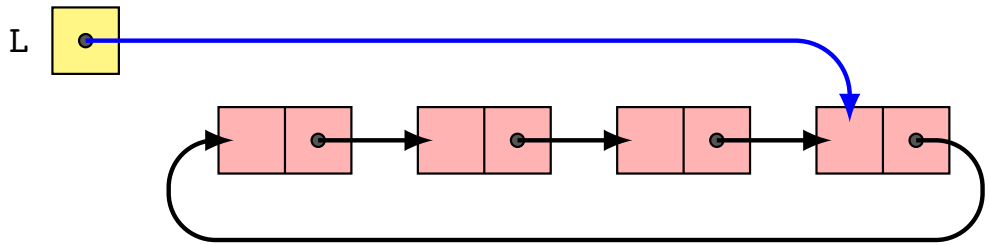
# Lista circular – Criar lista

lista_circular.c

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

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

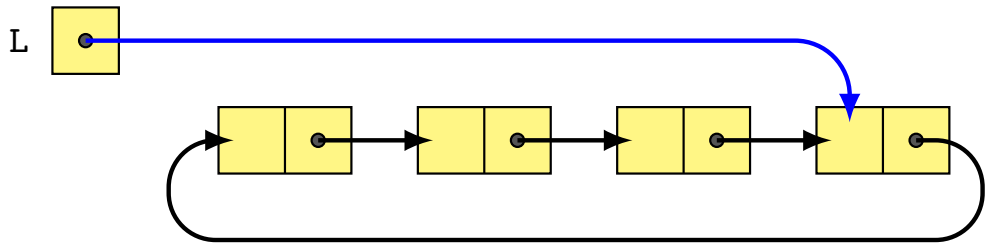
# Lista circular – Destruir lista




lista_circular.c

```
9  void destruir_lista(No **p) {//versão iterativa
10   if(*p == NULL) return;// lista vazia
11   No *q = (*p)->prox, *aux;
12   while(q != *p){
13     aux = q;
14     q = q->prox;
15     free(aux);
16   }
17   free(*p);
18   *p = NULL;
19 }
```

# Lista circular – Imprimir

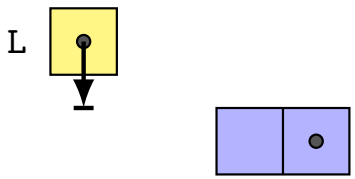



lista_circular.c

```
52 void imprimir_lista(No *p) { // p recebe L
53   No* q, *aux = p;
54   if(p!=NULL){
55     for (q = p->prox; q != aux; q = q->prox) printf("%d -> ", q->dado);
56     printf("%d -> ", q->dado);
57   }
58   printf("NULL\n");
59 }
```

- Precisamos verificar se a lista não é vazia

# Lista circular – Adicionar no início

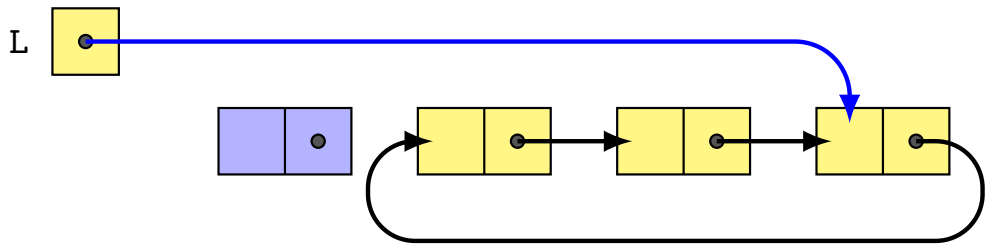



lista_circular.c

```
23 void adicionar_inicio(No** p, int x){//p recebe &L
24   No *q = (No*) malloc(sizeof(No));
25   q->dado = x;
26   if(*p == NULL){ //lista vazia
27     *p = q;
28     q->prox = q;
29   }
30   else{
31     q->prox = (*p)->prox;
32     (*p)->prox = q;
33   }
34 }
```

- Custo computacional: $O(1)$

# Lista circular – Adicionar no início

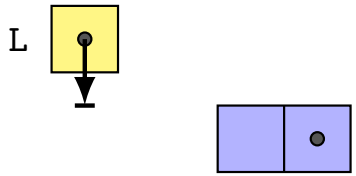



lista_circular.c

```
23 void adicionar_inicio(No** p, int x){//p recebe &L
24   No *q = (No*) malloc(sizeof(No));
25   q->dado = x;
26   if(*p == NULL){ //lista vazia
27     *p = q;
28     q->prox = q;
29   }
30   else{
31     q->prox = (*p)->prox;
32     (*p)->prox = q;
33   }
34 }
```

- Custo computacional: $O(1)$

# Lista circular – Adicionar no final

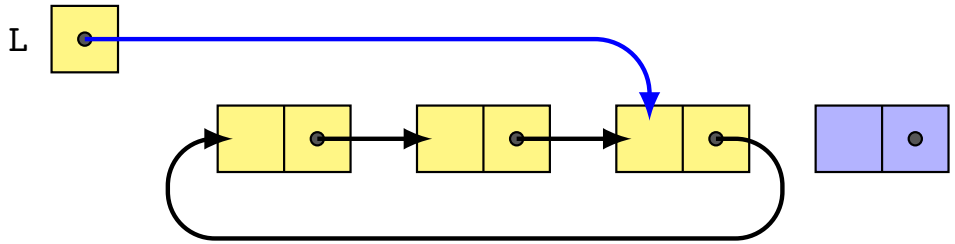



lista_circular.c

```
36 void adicionar_final(No **p, int x) {// p recebe L
37   No *q = (No*) malloc(sizeof(No));
38   q->dado = x;
39   if(*p == NULL){ //lista vazia
40     *p = q;
41     q->prox = q;
42   }
43   else{
44     q->prox = (*p)->prox;
45     (*p)->prox = q;
46     *p = q;
47   }
48 }
```

- Custo computacional: $O(1)$

# Lista circular – Adicionar no final

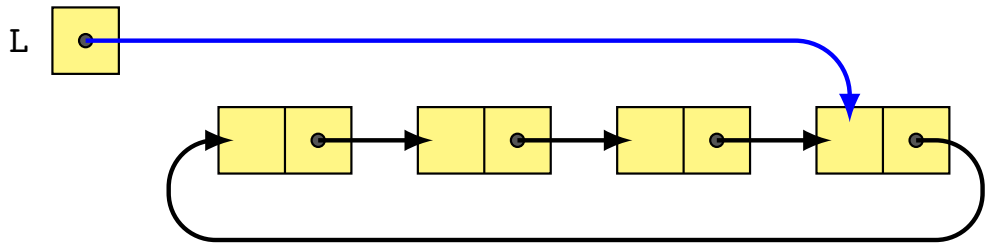



lista_circular.c

```
36 void adicionar_final(No **p, int x) {// p recebe L
37   No *q = (No*) malloc(sizeof(No));
38   q->dado = x;
39   if(*p == NULL){ //lista vazia
40     *p = q;
41     q->prox = q;
42   }
43   else{
44     q->prox = (*p)->prox;
45     (*p)->prox = q;
46     *p = q;
47   }
48 }
```

- Custo computacional: $O(1)$

# Lista circular – Buscar na lista

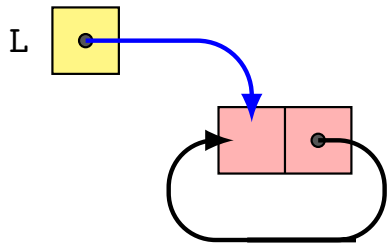



lista_circular.c

```
62 int buscar_valor(No *p, int x) {// q recebe L
63   if(p==NULL) return -1; //lista vazia
64   No *q = p;
65   p = p->prox;
66   while(p != q){
67     if(p->dado == x) return 1; //true!
68       p = p->prox;
69   }
70   if(p->dado == x) return 1; //true!
71   return 0;//false == não encontrou
72 }
```

- Custo computacional: $O(n)$

# Lista circular – Remover do início

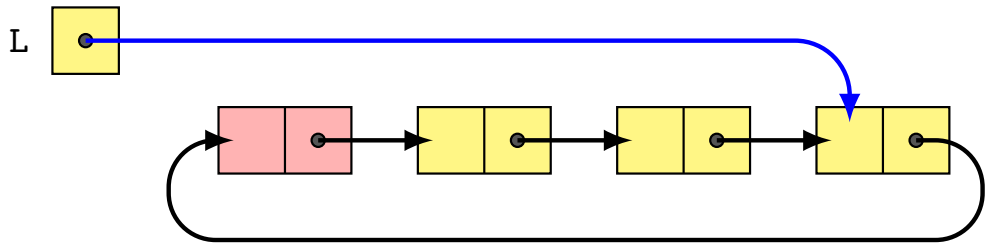



lista_circular.c

```
76 void remover_inicio(No **p) {// p recebe L
77   if(*p==NULL) return; //lista vazia
78   No* q = *p;
79   if(q->prox==q){//1 único nó
80     free(q); *p = NULL;
81     return;
82   }
83   q = q->prox;
84   (*p)->prox = q->prox;
85   free(q);
86 }
```

- Custo computacional: $O(1)$

# Lista circular – Remover do início

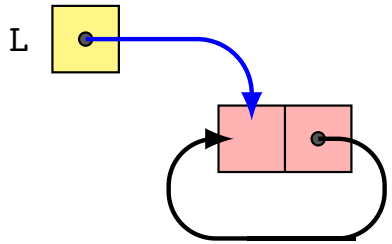



lista_circular.c

```
76 void remover_inicio(No **p) {// p recebe L
77   if(*p==NULL) return; //lista vazia
78   No* q = *p;
79   if(q->prox==q){//1 único nó
80     free(q); *p = NULL;
81     return;
82   }
83   q = q->prox;
84   (*p)->prox = q->prox;
85   free(q);
86 }
```

- Custo computacional: $O(1)$

# Lista circular – Remover do final

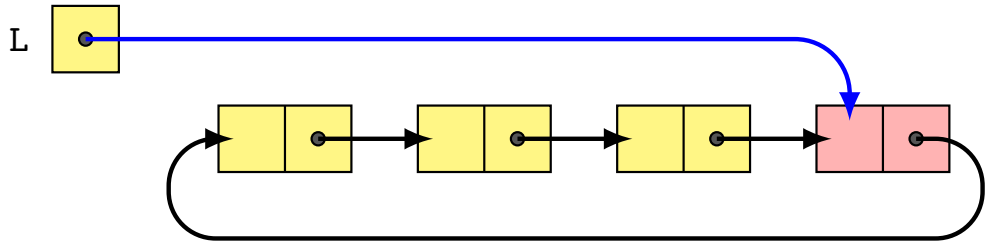



lista_circular.c

```
88  void remover_final(No **p) {// p recebe L
89    if(*p==NULL) return; //lista vazia
90    No* q = *p;
91    if(q->prox==q){//1 único nó
92      free(q); *p = NULL;
93      return;
94    }
95    No *aux = q;
96    while(q->prox != aux) q = q->prox;
97    q->prox = aux->prox;
98    *p = q;
99    free(aux);
100 }
```

- Custo computacional: $O(n)$

# Lista circular – Remover do final




lista_circular.c

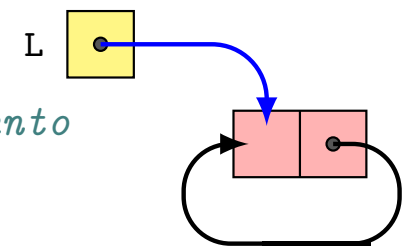
```
88  void remover_final(No **p) {// p recebe L
89    if(*p==NULL) return; //lista vazia
90    No* q = *p;
91    if(q->prox==q){//1 único nó
92      free(q); *p = NULL;
93      return;
94    }
95    No *aux = q;
96    while(q->prox != aux) q = q->prox;
97    q->prox = aux->prox;
98    *p = q;
99    free(aux);
100 }
```

- Custo computacional: $O(n)$

# Lista circular – Remover valor

lista_circular.c

```
102 void remover_valor(No **p, int x) {// p recebe &L
103   if(*p==NULL) return; //lista vazia
104   No *q = *p;
105   if(q->prox == q && q->dado == x){//1 único nó
106     free(q); *p = NULL;
107     return;
108   }
109   while(q->prox != *p && (q->prox)->dado != x) q = q->prox;
110   if((q->prox)->dado == x){//encontrou
111     No *aux = q->prox;
112     q->prox = aux->prox;
113     if(*p == aux)//último elemento
114       *p = q;
115     free(aux);
116   }
117 }
118
```

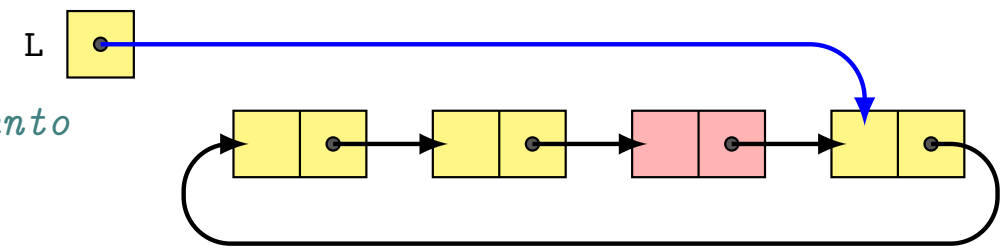



- Custo computacional: $O(n)$

# Lista circular – Remover valor

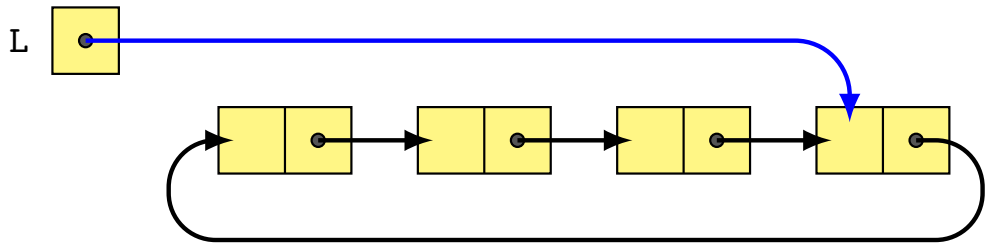
lista_circular.c

```
102 void remover_valor(No **p, int x) {// p recebe &L
103   if(*p==NULL) return; //lista vazia
104   No *q = *p;
105   if(q->prox == q && q->dado == x){//1 único nó
106     free(q); *p = NULL;
107     return;
108   }
109   while(q->prox != *p && (q->prox)->dado != x) q = q->prox;
110   if((q->prox)->dado == x){//encontrou
111     No *aux = q->prox;
112     q->prox = aux->prox;
113     if(*p == aux)//último elemento
114       *p = q;
115     free(aux);
116   }
117 }
118
```




- Custo computacional: $O(n)$

# Lista circular – Tamanho da lista




lista_circular.c

```
119 int tamanho_lista(No *p){
120   if(p==NULL) return 0; //lista vazia
121   No *q = p;
122   int tam=1;
123   for(p = p->prox; p != q; p = p->prox) tam++;
124   return tam;
125 }
```

- Precisamos percorrer a lista para saber o tamanho: $O(n)$

# Cliente


exemplo1.c

```c
#include <stdio.h>
#include "lista_circular.h"

int main(){
  int i, num;
  No *L = criar_lista();
  scanf("%d", &num);
  for(i=1; i<=num; i++) adicionar_final(&L, i);
  imprimir_lista(L);
  while(L!=NULL){
    scanf("%d", &i); remover_valor(&L, i);
    imprimir_lista(L);
    printf("[%d]\n", tamanho_lista(L));
  }
  destruir_lista(&L); //testar valgrind!!
return 0;
}
```

# Como compilar?

Teremos três arquivos diferentes:

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

Vamos compilar por partes:

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

# Makefile

Vamos usar o Makefile para compilar:

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

Relembrando:

- $^ 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

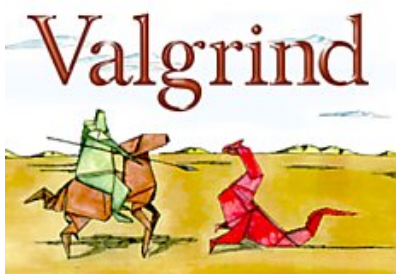
www.gnu.org/software/make/manual/html_node/Pattern-Rules.html

# Valgrind

Podemos verificar se toda memória foi desalocada com o `Valgrind`:

```
1 $ valgrind --leak-check=yes ./exemplo1
```

- Podemos encontrar vazamento de memória e acesso a posições inválidas:
  - Valgrind avisa que houveram bytes perdidos
  - Indica qual o `malloc` responsável pelo vazamento




valgrind.org/docs/manual/quick-start.html

# Listas circular vs. Listas ligadas (simples)

| | Listas circular | Listas Ligadas |
|---|---|---|
| Inserção no início | $O(1)$ | $O(1)$ |
| Inserção no final | $O(1)$ | $O(n)$ |
| Remoção no início | $O(1)$ | $O(1)$ |
| Remoção no final | $O(n)$ | $O(n)$ |
| Busca | $O(n)$ | $O(n)$ |

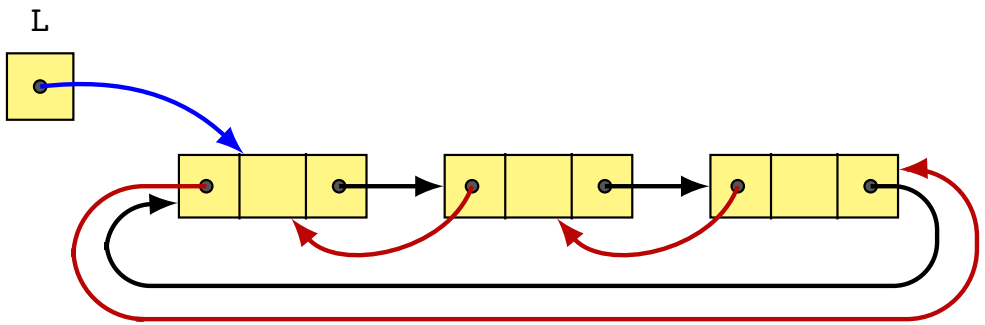
Comparação:

- Custo da inserção no final: $O(1)$
- Vamos ver mais uma variação de listas ligadas

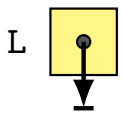
# Roteiro

# Variações - Listas duplamente ligadas

Lista (circular) duplamente ligada:



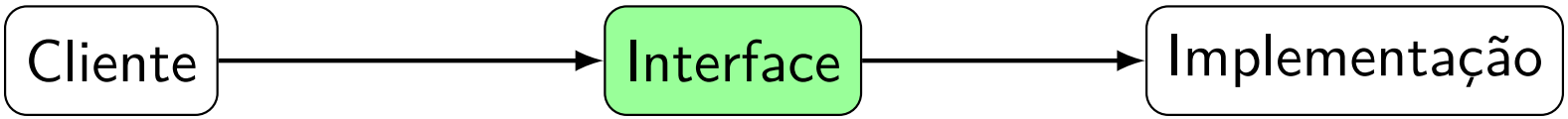

Lista vazia:




Diferenças:

- Cada nó aponta para o próximo e para o anterior.
- Podemos navegar para qualquer direção da lista.

# TAD - Interface




## lista_dupla.h

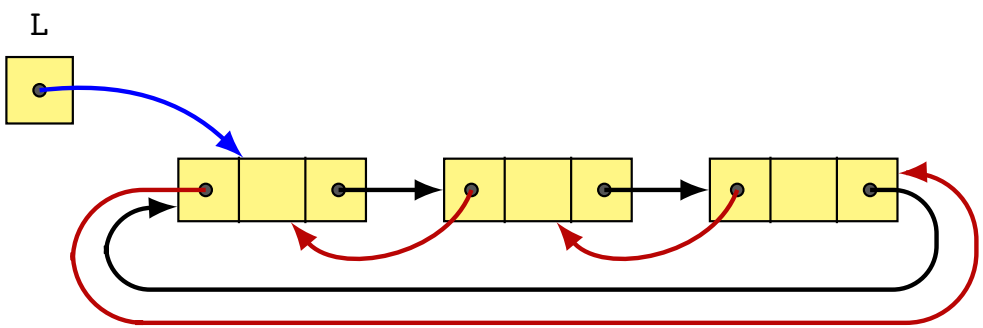
```c
#ifndef LISTA_DUPLA_H
#define LISTA_DUPLA_H

//Dados
typedef struct no {
  int dado;
  struct no *prox, *ant;
} No;

//Funções
No* criar_lista();
void destruir_lista(No **L);

//Imprimir
void imprimir_lista(No *L);

//Adicionar
void adicionar_inicio(No **L, int x);
void adicionar_final(No **L, int x);

//Remover
void remover_inicio(No **L);
void remover_final(No **L);
void remover_valor(No **L, int x);

//Buscar
int buscar_valor(No *L, int x);

//Extra
int tamanho_lista(No *L);

#endif
```

# Lista duplamente ligada – Criar e destruir

lista_dupla.c

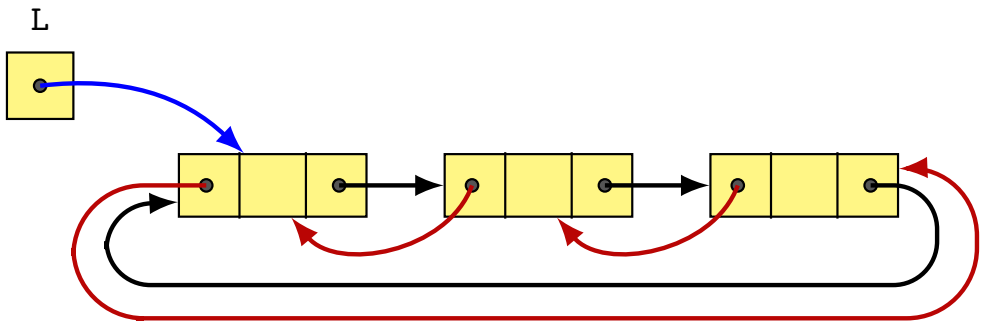
```
5  No* criar_lista() {
6    return NULL;
7  }
```

```
9  void destruir_lista(No **p) {//versão iterativa
10   if(*p == NULL) return;// lista vazia
11   No *q = (*p)->prox, *aux;
12   while(q != *p){
13     aux = q;
14     q = q->prox;
15     free(aux);
16   }
17   free(*p);
18   *p = NULL;
19 }
```




- Mesmas funções!

# Lista duplamente ligada – Imprimir



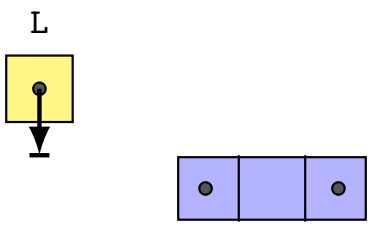

lista_dupla.c

```
58 void imprimir_lista(No *p) { // p recebe L
59   No* q;
60   if(p!=NULL){
61     No *aux = p->ant;
62     for (q = p; q != aux; q = q->prox) printf("%d -> ", q->dado);
63     printf("%d -> ", q->dado);
64   }
65   printf("NULL\n");
66 }
```

# Lista duplamente ligada – Adicionar no início

lista_dupla.c

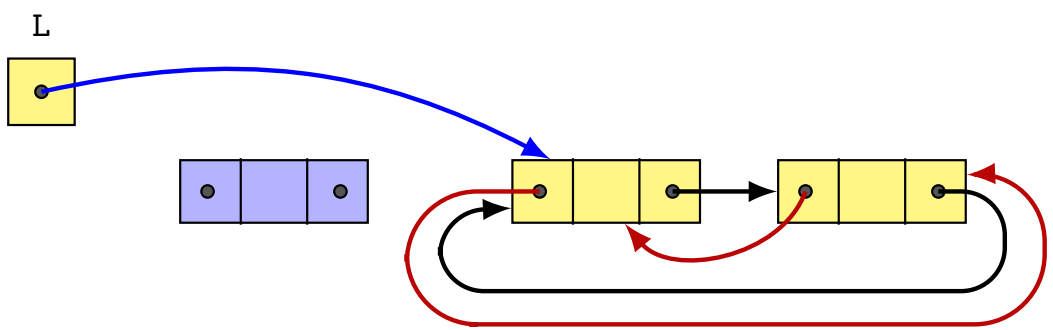
```
23 void adicionar_inicio(No** p, int x){//p recebe &L
24   No *q = (No*) malloc(sizeof(No));
25   q->dado = x;
26   if(*p == NULL){ //lista vazia
27     *p = q;
28     q->prox = q->ant = q;
29   }
30   else{
31     No *aux = *p;
32     q->ant  = aux->ant;
33     q->prox = aux;
34     (aux->ant)->prox = q;
35     aux->ant = q;
36     *p = q;
37   }
38 }
```



# Lista duplamente ligada – Adicionar no início

lista_dupla.c

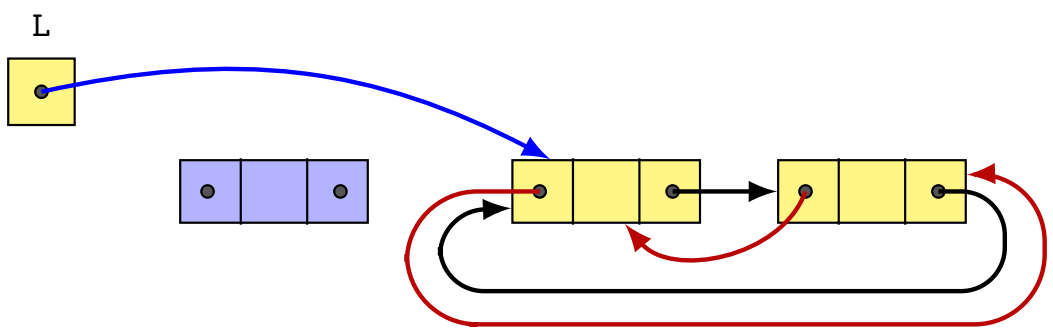
```
23 void adicionar_inicio(No** p, int x){//p recebe &L
24   No *q = (No*) malloc(sizeof(No));
25   q->dado = x;
26   if(*p == NULL){ //lista vazia
27     *p = q;
28     q->prox = q->ant = q;
29   }
30   else{
31     No *aux = *p;
32     q->ant  = aux->ant;
33     q->prox = aux;
34     (aux->ant)->prox = q;
35     aux->ant = q;
36     *p = q;
37   }
38 }
```



# Lista duplamente ligada – Adicionar no início

lista_dupla.c

```
23 void adicionar_inicio(No** p, int x){//p recebe &L
24   No *q = (No*) malloc(sizeof(No));
25   q->dado = x;
26   if(*p == NULL){ //lista vazia
27     *p = q;
28     q->prox = q->ant = q;
29   }
30   else{
31     No *aux = *p;
32     q->ant  = aux->ant;
33     q->prox = aux;
34     (aux->ant)->prox = q;
35     aux->ant = q;
36     *p = q;
37   }
38 }
```

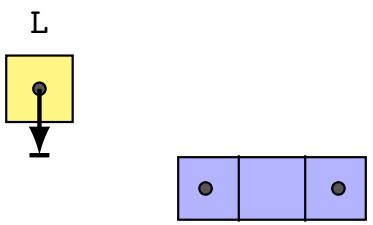



- Custo computacional: $O(1)$

# Lista duplamente ligada – Adicionar no final

lista_dupla.c

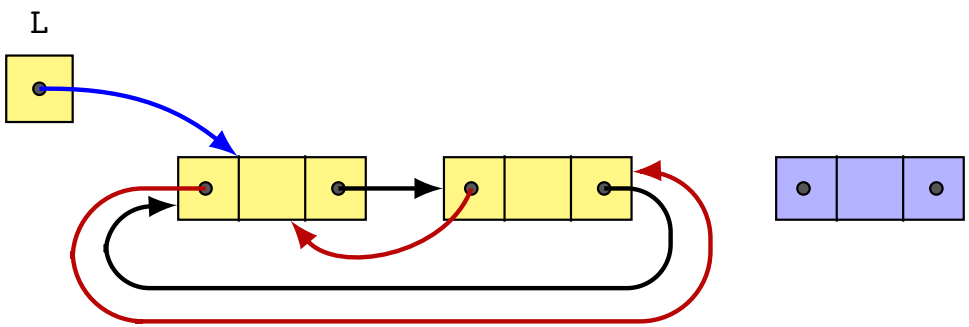
```
40 void adicionar_final(No **p, int x) {// p recebe L
41   No *q = (No*) malloc(sizeof(No));
42   q->dado = x;
43   if(*p == NULL){ //lista vazia
44     *p = q;
45     q->prox = q->ant = q;
46   }
47   else{
48     No *aux = (*p)->ant;
49     (*p)->ant = q;
50     q->prox = *p;
51     aux->prox =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



# Lista duplamente ligada – Adicionar no final

lista_dupla.c

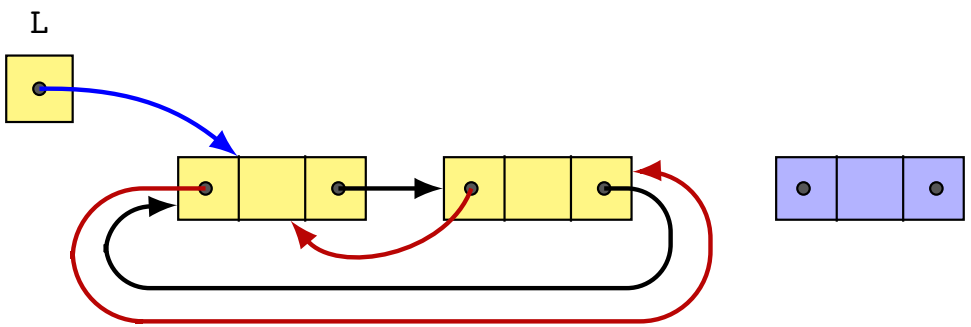
```
40 void adicionar_final(No **p, int x) {// p recebe L
41   No *q = (No*) malloc(sizeof(No));
42   q->dado = x;
43   if(*p == NULL){ //lista vazia
44     *p = q;
45     q->prox = q->ant = q;
46   }
47   else{
48     No *aux = (*p)->ant;
49     (*p)->ant = q;
50     q->prox = *p;
51     aux->prox =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



# Lista duplamente ligada – Adicionar no final

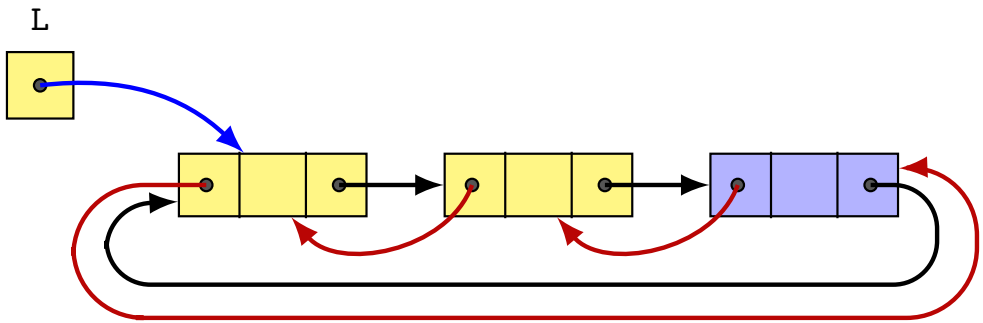
lista_dupla.c

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




- Custo computacional: $O(1)$

# Lista duplamente ligada – Buscar no lista

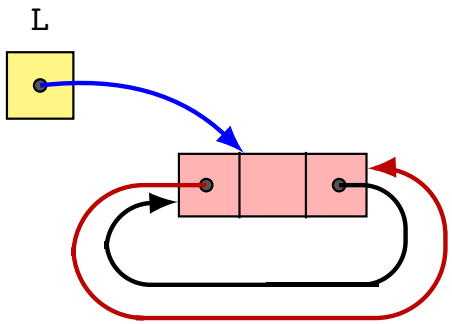



lista_dupla.c

```
69 int buscar_valor(No *p, int x) {// q recebe L
70   if(p==NULL) return -1; //lista vazia
71   No *q = p->ant;
72   while(p != q){
73     if(p->dado == x) return 1; //true!
74       p = p->prox;
75   }
76   if(p->dado == x) return 1; //true!
77   return 0;//false == não encontrou
78 }
```

- Custo computacional: $O(n)$

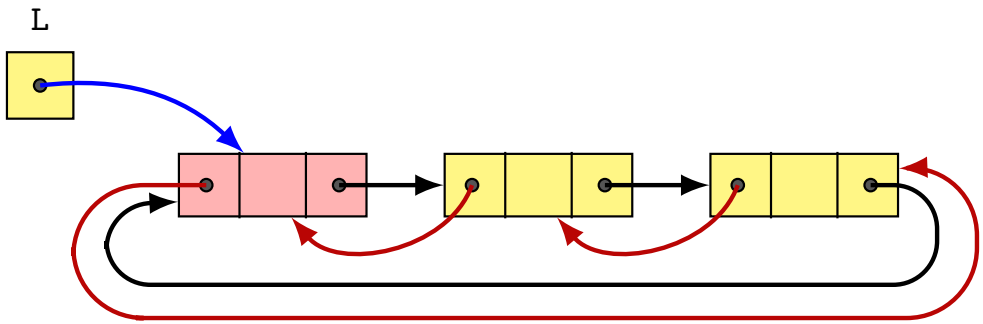
# Lista duplamente ligada – Remover do início




lista_dupla.c

```
82 void remover_inicio(No **p) {// p recebe L
83   if(*p==NULL) return; //lista vazia
84   No* q = *p;
85   if(q->prox==q){//1 único nó
86     free(q); *p = NULL;
87     return;
88   }
89   *p = q->prox;
90   (*p)->ant = q->ant;
91   (q->ant)->prox = *p;
92   free(q);
93 }
```

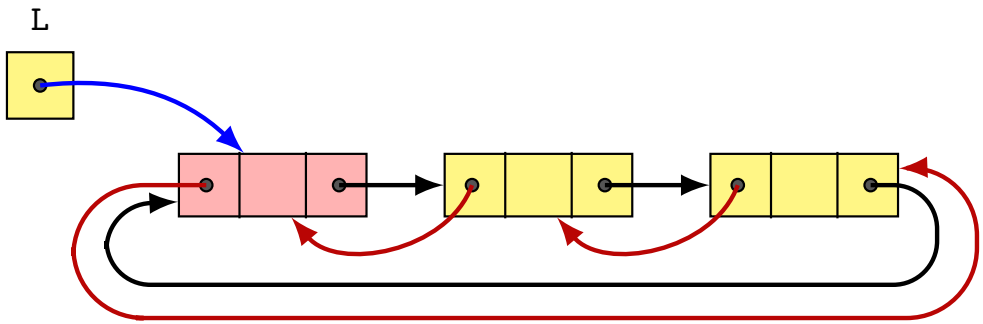
# Lista duplamente ligada – Remover do início




lista_dupla.c

```
82 void remover_inicio(No **p) {// p recebe L
83   if(*p==NULL) return; //lista vazia
84   No* q = *p;
85   if(q->prox==q){//1 único nó
86     free(q); *p = NULL;
87     return;
88   }
89   *p = q->prox;
90   (*p)->ant = q->ant;
91   (q->ant)->prox = *p;
92   free(q);
93 }
```

# Lista duplamente ligada – Remover do início

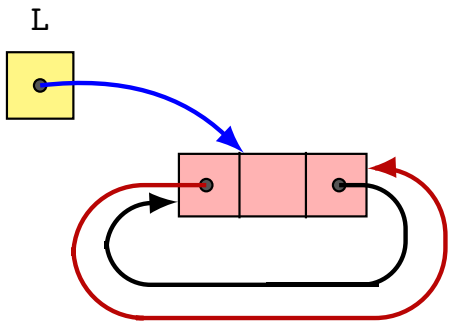



lista_dupla.c

```
82 void remover_inicio(No **p) {// p recebe L
83   if(*p==NULL) return; //lista vazia
84   No* q = *p;
85   if(q->prox==q){//1 único nó
86     free(q); *p = NULL;
87     return;
88   }
89   *p = q->prox;
90   (*p)->ant = q->ant;
91   (q->ant)->prox = *p;
92   free(q);
93 }
```

- Custo computacional: $O(1)$

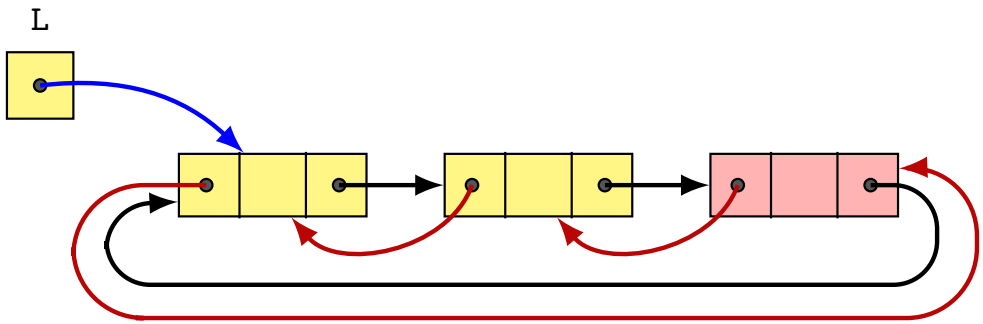
# Lista duplamente ligada – Remover do final




lista_dupla.c

```
95  void remover_final(No **p) {// p recebe L
96    if(*p==NULL) return; //lista vazia
97    No* q = (*p)->ant;
98    if(q==*p){//1 único nó
99      free(q); *p = NULL;
100     return;
101   }
102   No *aux = q->ant;
103   aux->prox = *p;
104   (*p)->ant = aux;
105   free(q);
106 }
```

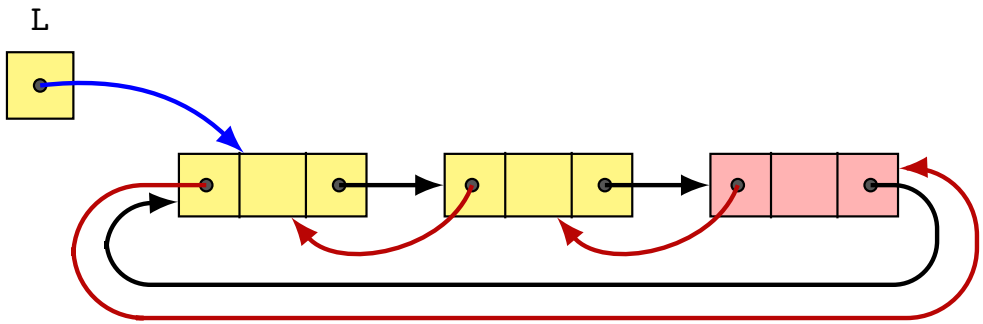
# Lista duplamente ligada – Remover do final




lista_dupla.c

```
95  void remover_final(No **p) {// p recebe L
96    if(*p==NULL) return; //lista vazia
97    No* q = (*p)->ant;
98    if(q==*p){//1 único nó
99      free(q); *p = NULL;
100     return;
101   }
102   No *aux = q->ant;
103   aux->prox = *p;
104   (*p)->ant = aux;
105   free(q);
106 }
```

# Lista duplamente ligada – Remover do final




lista_dupla.c

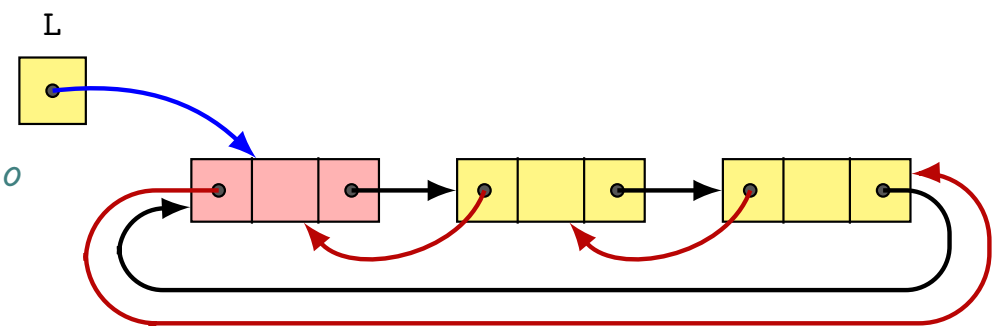
```
95  void remover_final(No **p) {// p recebe L
96    if(*p==NULL) return; //lista vazia
97    No* q = (*p)->ant;
98    if(q==*p){//1 único nó
99      free(q); *p = NULL;
100     return;
101   }
102   No *aux = q->ant;
103   aux->prox = *p;
104   (*p)->ant = aux;
105   free(q);
106 }
```

- Custo computacional: $O(1)$

# Lista duplamente ligada – Remover valor

lista_dupla.c

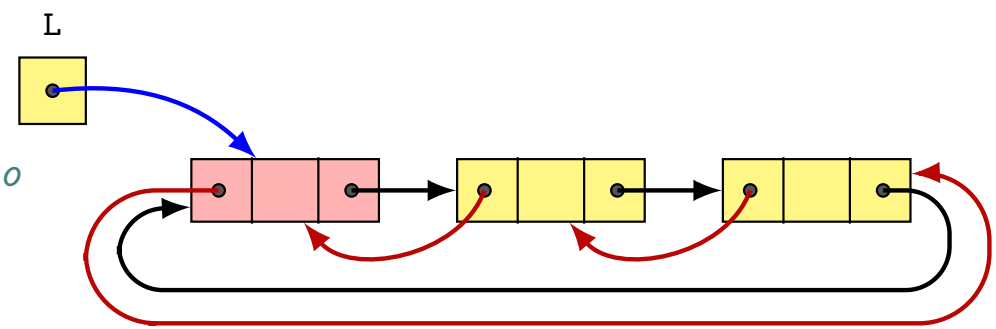
```
108 void remover_valor(No **p, int x) {// p recebe &L
109   if(*p==NULL) return; //lista vazia
110   No *q = *p;
111   if(q->prox == q && q->dado == x){//1 único nó
112     free(q); *p = NULL;
113     return;
114   }
115   while(q != (*p)->ant && q->dado != x) q = q->prox;
116   if(q->dado == x){//encontrou
117     No *aux1 = q->ant;
118     No *aux2 = q->prox;
119     aux1->prox = aux2;
120     aux2->ant  = aux1;
121     if(*p == q)//primeiro elemento
122       *p = aux2;
123     free(q);
124   }
125 }
```



# Lista duplamente ligada – Remover valor

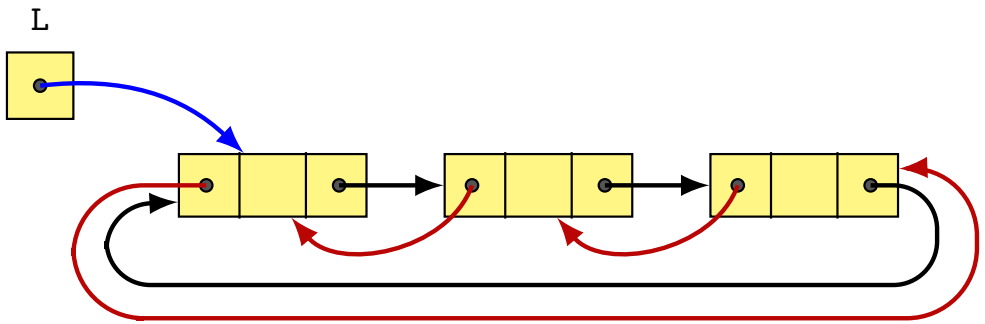
lista_dupla.c

```
108 void remover_valor(No **p, int x) {// p recebe &L
109   if(*p==NULL) return; //lista vazia
110   No *q = *p;
111   if(q->prox == q && q->dado == x){//1 único nó
112     free(q); *p = NULL;
113     return;
114   }
115   while(q != (*p)->ant && q->dado != x) q = q->prox;
116   if(q->dado == x){//encontrou
117     No *aux1 = q->ant;
118     No *aux2 = q->prox;
119     aux1->prox = aux2;
120     aux2->ant  = aux1;
121     if(*p == q)//primeiro elemento
122       *p = aux2;
123     free(q);
124   }
125 }
```




- Custo computacional: $O(1)$

# Lista duplamente ligada – Tamanho da lista




lista_dupla.c

```
127 int tamanho_lista(No *p){
128   if(p==NULL) return 0; //lista vazia
129   No *q;
130   int tam=1;
131   for(q = p; q != p->ant; q = q->prox) tam++;
132   return tam;
133 }
```

- Precisamos percorrer a lista para saber o tamanho: $O(n)$

# Cliente


exemplo2.c

```c
#include <stdio.h>
#include "lista_dupla.h"

int main(){
  int i, num;
  No *L = criar_lista();
  scanf("%d", &num);
  for(i=1; i<=num; i++) adicionar_final(&L, i);
  imprimir_lista(L);
  while(L!=NULL){
    scanf("%d", &i); remover_valor(&L, i);
    imprimir_lista(L);
    printf("[%d]\n", tamanho_lista(L));
  }
  destruir_lista(&L); //testar valgrind!!
return 0;
}
```

# Makefile e Valgrind

Vamos usar o Makefile para compilar:

```
1 exemplo2: exemplo2.c lista_dupla.o
2   gcc $^ -o $@
```

Vamos verificar a memória com o Valgrind:

```
1 $ valgrind --leak-check=yes ./exemplo2
```

# Listas duplamente ligadas vs. Listas ligadas (simples)

| | Listas duplamente ligadas | Listas Ligadas |
|---|---|---|
| Inserção no início | $O(1)$ | $O(1)$ |
| Inserção no final | $O(1)$ | $O(n)$ |
| Remoção no início | $O(1)$ | $O(1)$ |
| Remoção no final | $O(1)$ | $O(n)$ |
| Busca | $O(n)$ | $O(n)$ |

Uso do espaço:

- Listas duplamente ligadas tem um overhead de 8 bytes por nó

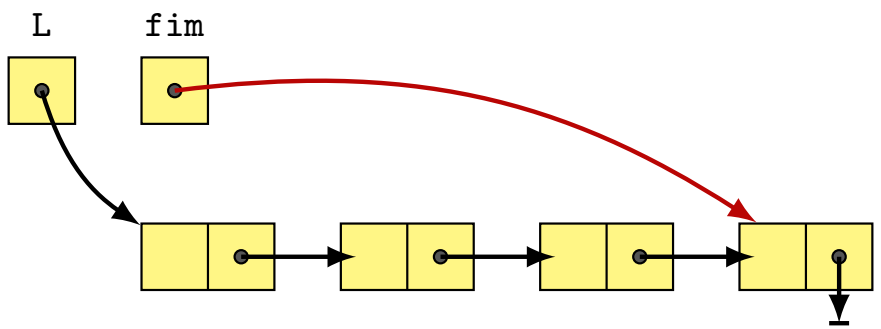
Qual é melhor?

- depende do problema, do algoritmo e da implementação

# Roteiro

# Algumas alternativas

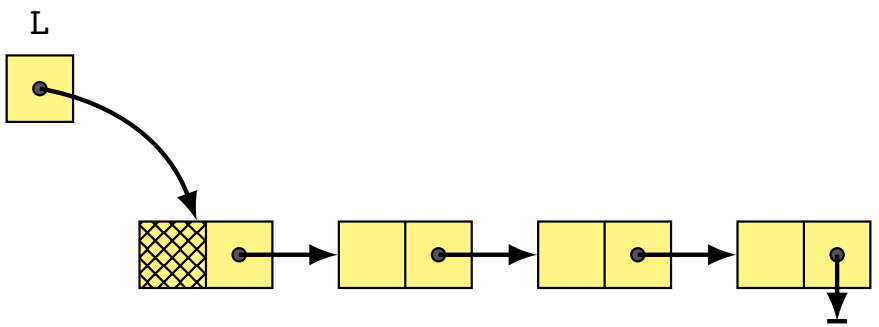
Apontador para o fim da lista:




Inserção no final da lista em $\mathrm{O}(1)$.

# Variações - Nó cabeça

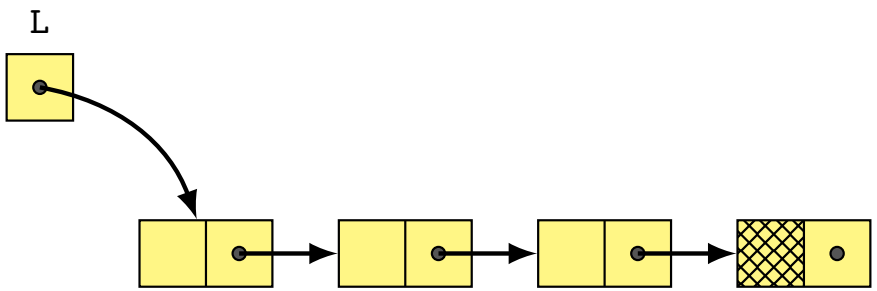
Lista com nó cabeça:




Simplifica as implementações.

# Variações - Nó sentinela

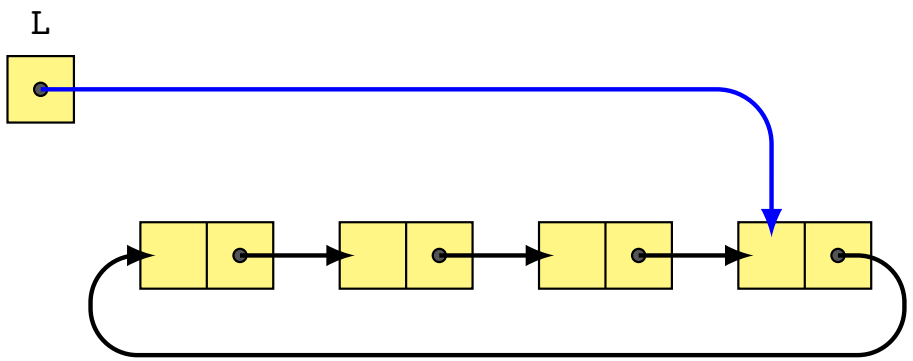
Nó sentinela para marcar o fim da lista:




Simplifica as implementações.

# Variações - Lista circular

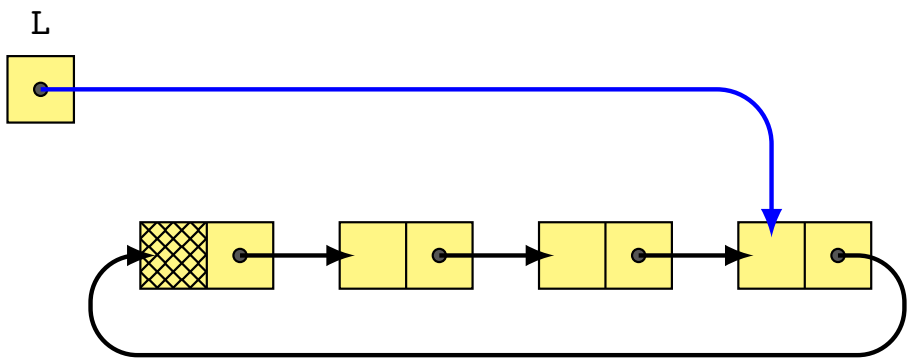
Lista circular:




Facilita a navegação.

# Variações - Lista circular com nó cabeça

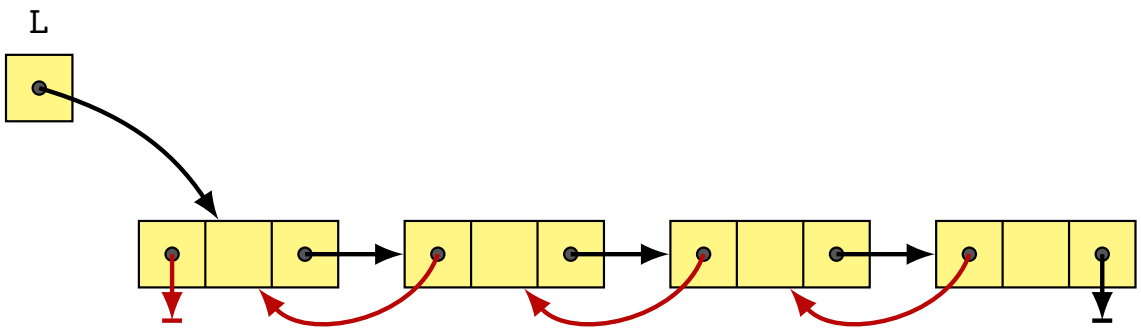
Lista circular com nó cabeça:




Simplifica as implementações.

# Variações - Lista duplamente ligada

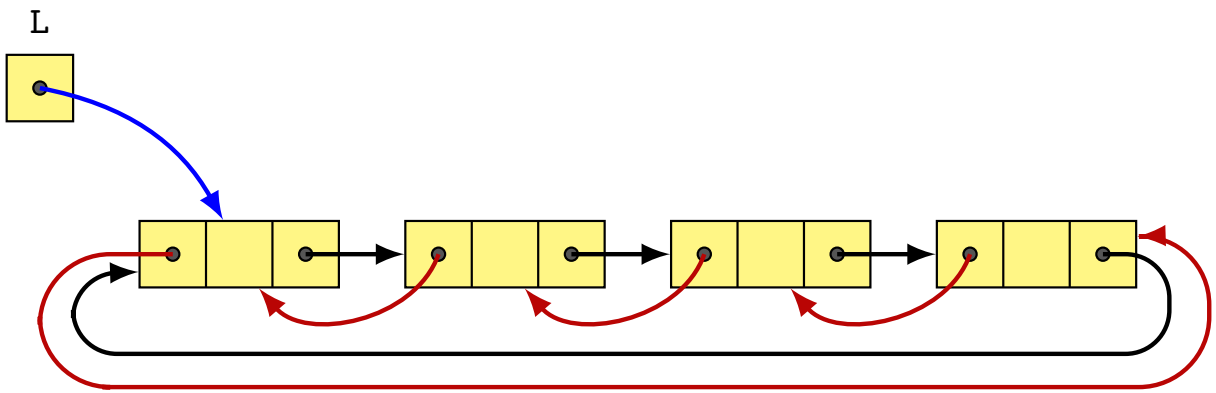
Lista duplamente ligada:




Facilita a navegação. Custo extra de memória.

# Variações - Lista duplamente ligada

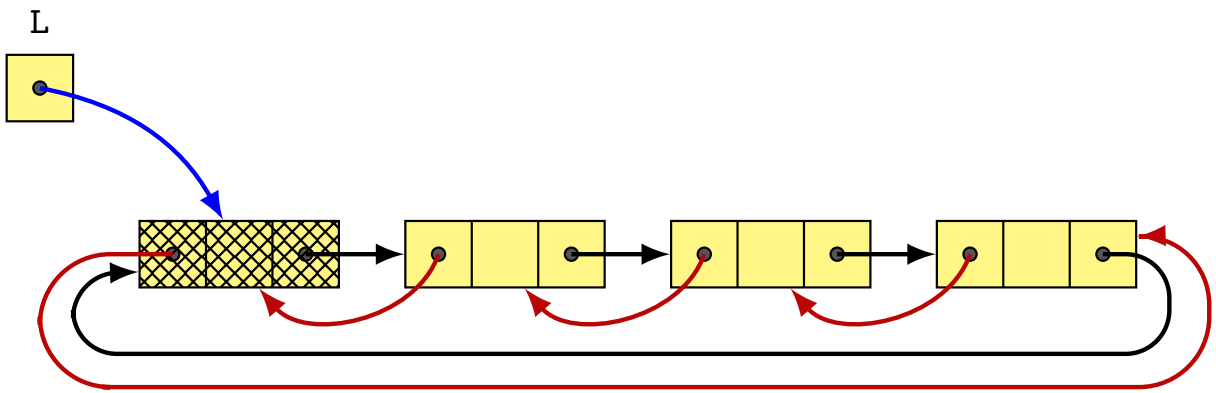
Lista (circular) duplamente ligada:




Facilita a navegação. Custo extra de memória.

# Variações - Lista duplamente ligada

Lista (circular) duplamente ligada com nó cabeça:




Facilita a navegação. Custo extra de memória.

# Fim

Dúvidas?

# Roteiro

# 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.