Estruturas de Dados Elementares: LISTAS

Profª. Rose Yuri Shimizu

Roteiro

1. ESTRUTURAS DE DADOS ELEMENTARES
2. LISTAS ESTÁTICAS
3. LISTAS SIMPLESMENTE ENCADEADAS
4. LISTA DUPLAMENTE ENCADEADAS

Estrutura de Dados Elementares

- Estrutura de dados
  - Organizam uma coleção de dados
  - Possuem um conjunto de operações
- Elementar
  - Utilizados por outras estruturas
- Estrutura elementar: lista

Roteiro

1. ESTRUTURAS DE DADOS ELEMENTARES
2. LISTAS ESTÁTICAS
3. LISTAS SIMPLESMENTE ENCADEADAS
4. LISTA DUPLAMENTE ENCADEADAS

LISTA ESTÁTICA

- Conjunto do **mesmo tipo** de dado
- Espaço **consecutivo** na memória RAM
- **Acesso aleatório**: qualquer posição pode ser acessada facilmente através de um **index**
- Nome $\rightarrow$ corresponde ao **endereço de memória**
- Tamanho **fixo** (stack) ou alocado **dinamicamente** (heap)
- Arrays:
  - https://fga.rysh.com.br/eda1/aulas/1-estrutura_de_dados.pdf
  - https://fga.rysh.com.br/eda1/aulas/2-alocacao_memoria.pdf
  - https://www.ime.usp.br/~pf/algoritmos/aulas/array.html

LISTA ESTÁTICA

- VANTAGEM: fácil acesso
- DESVANTAGEM: difícil manipulação
- Alternativa: LISTAS ENCADEADAS
- `https://www.ime.usp.br/~pf/algoritmos/aulas/lista.html`
- `https://www.ime.usp.br/~pf/mac0122-2002/aulas/llists.html`

Roteiro

1 ESTRUTURAS DE DADOS ELEMENTARES

2 LISTAS ESTÁTICAS

3 LISTAS SIMPLESMENTE ENCADEADAS

4 LISTA DUPLAMENTE ENCADEADAS

LISTA SIMPLESMENTE ENCADEADAS

- Conjunto de nós ou células
- Cada nó é tipo um contêiner que armazena **item + link (para outro nó)**

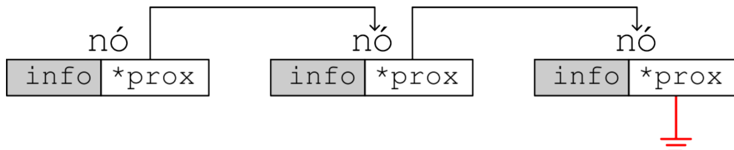
- Alocação conforme necessidade
- Mais adequado para **manipulações** do que acessos:
  - Maior eficiência para **rearranjar os itens** (reapontamentos)
  - **Não tem acesso direto** aos itens pela sua posição
- Operações: buscar, inserir, remover

LISTA SIMPLESMENTE ENCADEADAS

Nós da lista

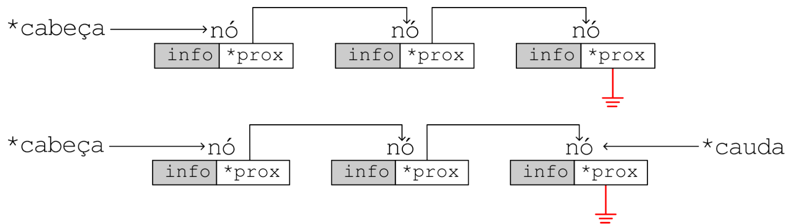
```c
//typedef struct item Item;
typedef int Item;
```

```c
typedef struct node no;
struct node {
    Item info;
    no *prox;
};
```



LISTA SIMPLESMENTE ENCADEADAS - Tipos

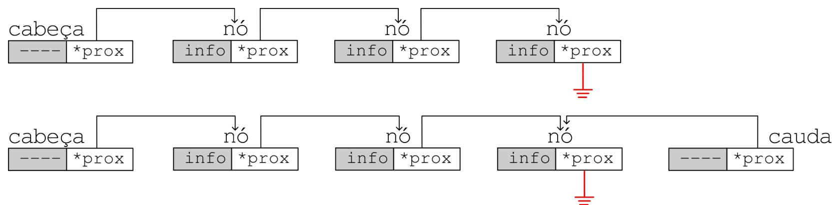
- Fim da lista: último nó aponta para NULL
- Início sem cabeça
  - Primeiro nó é o primeiro item da lista
  - Ponteiro (auxiliar) pode armazenar o endereço do primeiro nó
  - Com ou sem cauda




```c
no *lista = NULL;

no *novo = malloc(sizeof(no));
novo->prox = NULL;
novo->info = 2;

lista = novo;
```

LISTA SIMPLESMENTE ENCADEADAS - Tipos

- Fim da lista: último nó aponta para NULL
- Início sem cabeça
- Início com cabeça do tipo **nó**
  - Conteúdo é ignorado
  - Elementos da lista: a partir do segundo nó
  - Com ou sem cauda

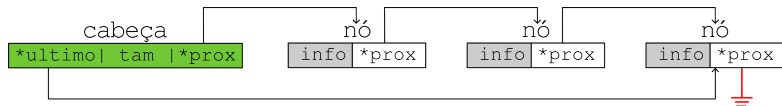



```c
no *lista = malloc(sizeof(no));

no *novo = malloc(sizeof(no));
novo->prox = NULL;
novo->info = 2;

lista->prox = novo;
```

Início com cabeça do tipo cabeça (específico)

LISTA SIMPLESMENTE ENCADEADAS - Tipos

- Fim da lista: último nó aponta para NULL
- Início sem cabeça
- Início com cabeça do tipo **nó**
- Início com cabeça do tipo **cabeca** (específico)
  - Aproveita para guardar metadados
  - Tamanho da lista e fim da lista(cauda), por exemplo
  - Elementos da lista: a partir do segundo nó




```c
typedef struct {
    int tam;
    no *prox;
    no *ultimo;
}cabeca;
```

LISTA SIMPLESMENTE ENCADEADAS - Operações básicas

```
typedef struct node no;
typedef int Item;
struct node{
    Item info;
    no *prox;
};

typedef struct head cabeca;
struct head{
    int num_itens;
    no *prox;
    no *ultimo;
};


//PROTÓTIPO DAS OPERAÇÕES BÁSICAS
cabeca *criar_lista();
no *criar_no(Item);
int vazia(cabeca *);
int tamanho(cabeca*);

```

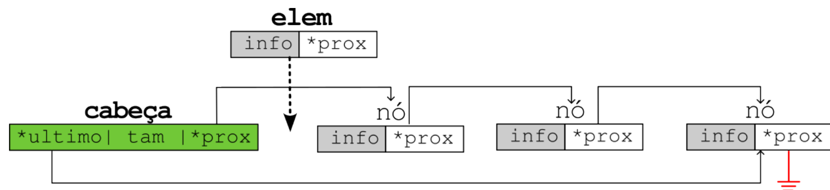
LISTA SIMPLESMENTE ENCADEADAS - Operações básicas

```c
22
23 void inserir_depois(cabeca *, no *, no *);
24 void inserir_inicio(cabeca *, no *);
25 void inserir_fim(cabeca *, no *);
26
27 no *remover_inicio(cabeca *);
28 no *remover_fim(cabeca *);
29 no *remover_no(cabeca *, no *);
30
31 no *buscar(cabeca *, Item);
```

```c
cabeca *criar_lista() {
    cabeca *le = malloc(sizeof(cabeca));
    le->num_itens = 0;
    le->prox = NULL;
    le->ultimo = NULL;

    return le;
}

no *criar_no(Item x) {
    no *novo = malloc(sizeof(no));
    novo->prox = NULL;
    novo->info = x;
    return novo;
}

int vazia(cabeca *lista) {
    return (lista->prox==NULL);
}

no *buscar(cabeca *lista, Item x) {
    no *a = NULL;
    for(a=lista->prox; a && a->info!=x; a=a->prox);
    return a;
}
```

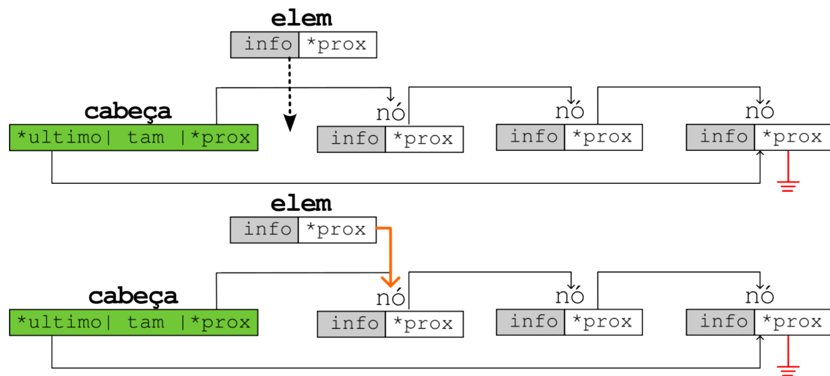
```
void inserir_inicio(cabeca *lista, no *elem)
```

Sendo: `elem!=NULL`



```
void inserir_inicio(cabeca *lista, no *elem)
```

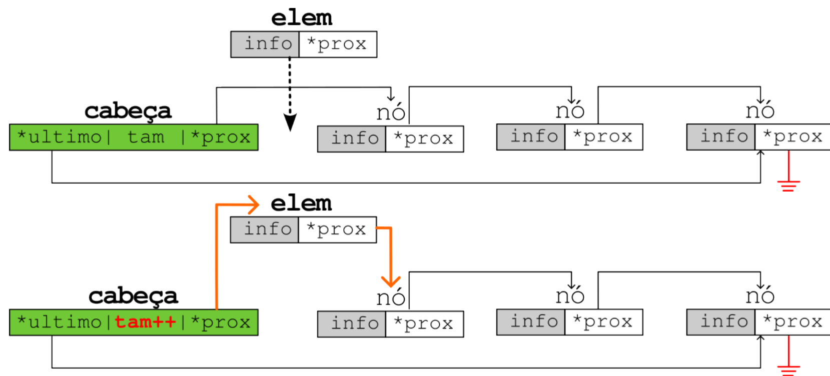
Sendo: `elem!=NULL`




```
elem->prox = lista->prox
```

```
void inserir_inicio(cabeca *lista, no *elem)
```

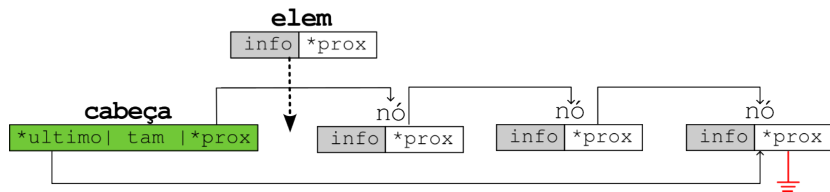
Sendo: elem!=NULL

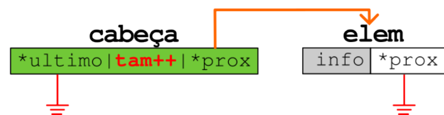



```
lista->prox = elem
```

```
void inserir_inicio(cabeca *lista, no *elem)
```

Sendo: `elem!=NULL`

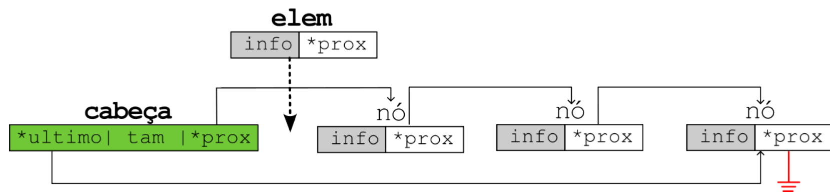


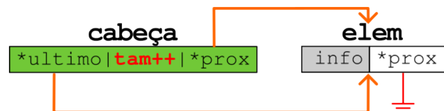



```
if(elem->prox == NULL) ???
```

```
void inserir_inicio(cabeca *lista, no *elem)
```

Sendo: elem!=NULL



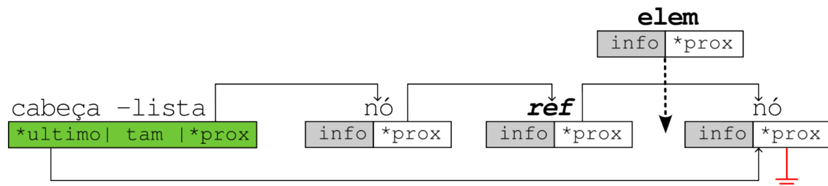



```
if(elem->prox == NULL) lista->ultimo = elem
```

```c
void inserir_inicio(cabeca *lista, no *elem)
{
    elem->prox = lista->prox;
    lista->prox = elem;

    //metadados
    lista->num_itens++;
    if(elem->prox == NULL) lista->ultimo = elem;
}
```

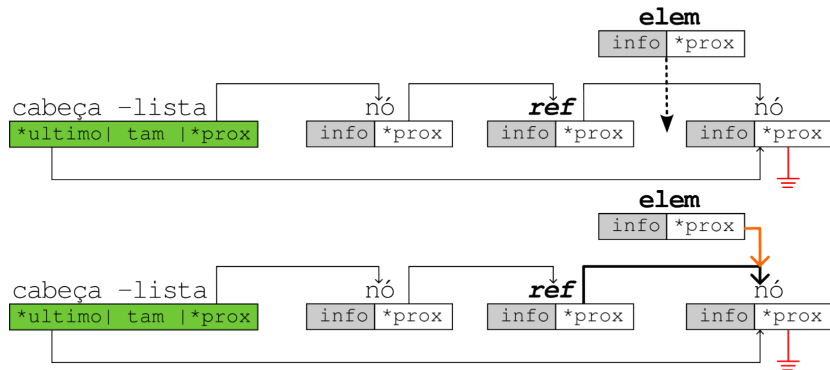
```
void inserir_depois(cabeca *lista, no *ref, no *elem)
```

Sendo: `elem!=NULL`



```
void inserir_depois(cabeca *lista, no *ref, no *elem)
```

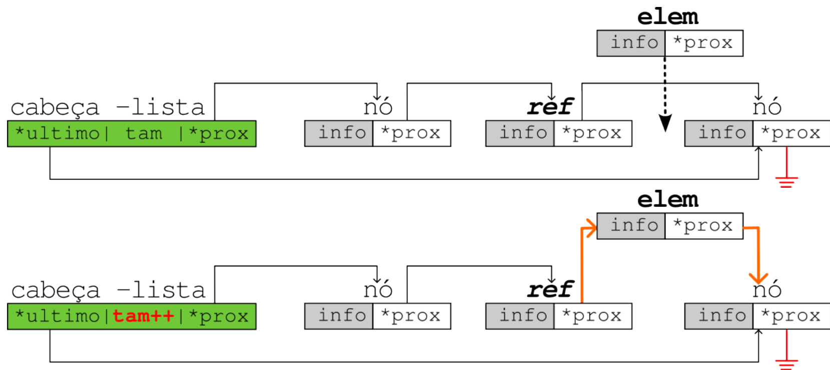
Sendo: `elem!=NULL`




```
elem->prox = ref->prox
```

`void inserir_depois(cabeca *lista, no *ref, no *elem)`

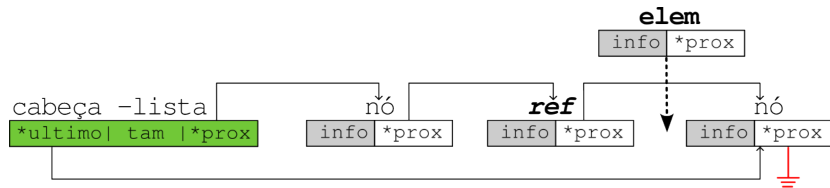
Sendo: `elem!=NULL`

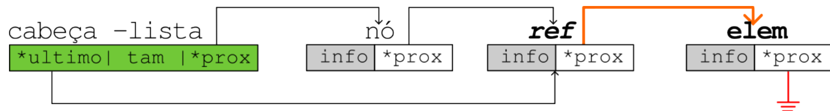



```
ref->prox = elem
```

```
void inserir_depois(cabeca *lista, no *ref, no *elem)
```

Sendo: `elem!=NULL`

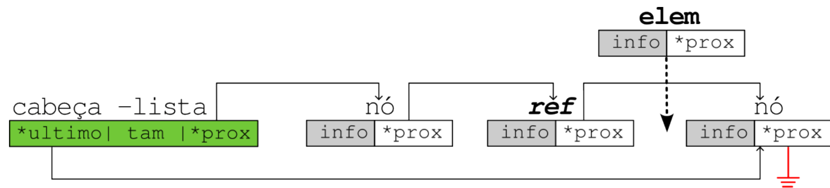


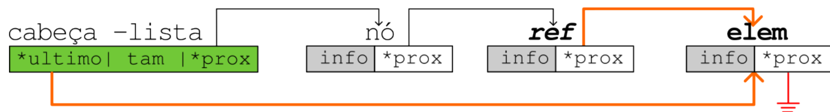



```
if(elem->prox == NULL) ???
```

```
void inserir_depois(cabeca *lista, no *ref, no *elem)
```

Sendo: `elem!=NULL`

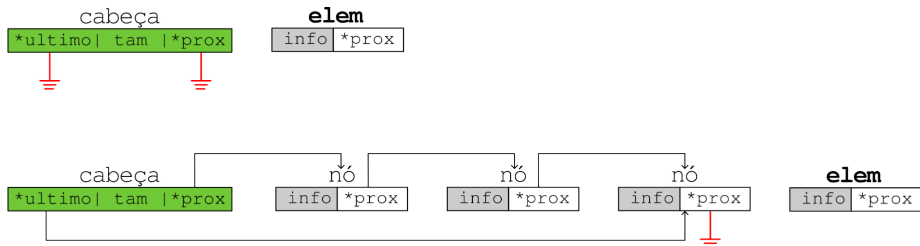





```
if(elem->prox == NULL) lista->ultimo = elem
```

```c
void inserir_depois(cabeca *lista, no *ref, no *elem)
{
    elem->prox = ref->prox;
    ref->prox = elem;

    //metadados
    lista->num_itens++;
    if(elem->prox == NULL) lista->ultimo = elem;
}
```

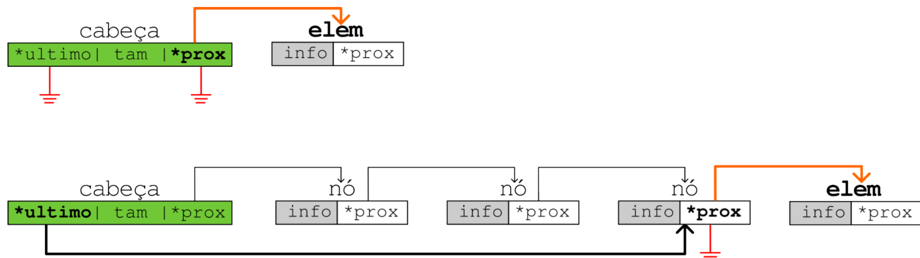
```
void inserir_fim(cabeca *lista, no *elem)
```
Sendo: `elem!=NULL`




```
if(vazia(lista)) ??? e if(!vazia(lista)) ???
```

```
void inserir_fim(cabeca *lista, no *elem)
```

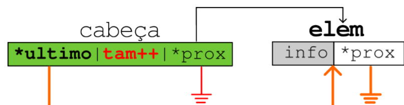
Sendo: elem!=NULL

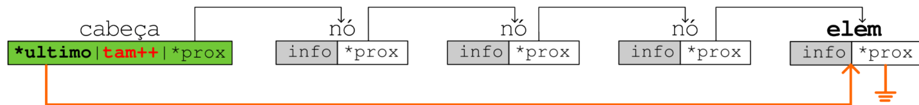



```
if(vazia(lista)) lista->prox = elem
 else lista->ultimo->prox = elem
```

`void inserir_fim(cabeca *lista, no *elem)`

Sendo: `elem!=NULL`

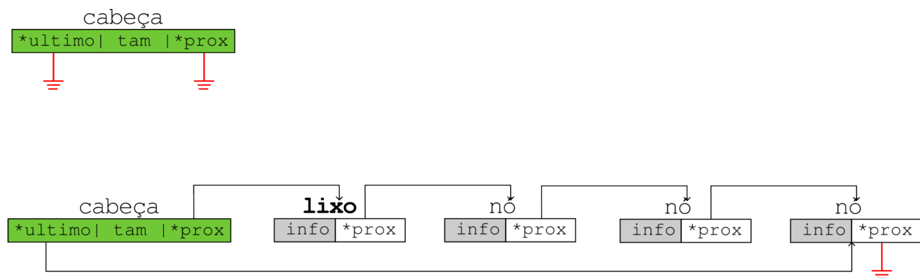




`lista->ultimo = elem e elem->prox = NULL`

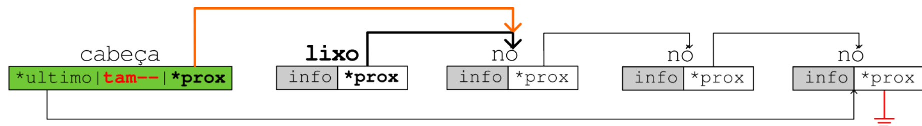
```c
void inserir_fim(cabeca *lista, no *ref, no *elem)
{
    if(vazia(lista)) lista->prox = elem;
    else lista->ultimo->prox = elem;

    lista->ultimo = elem;
    elem->prox = NULL;

    lista->num_itens++;

    //OU
    /*if(vazia(lista))
        return inserir_inicio(lista, elem);

    inserir_depois(lista, lista->ultimo, elem);*/;

}
```

```
void remover_inicio(cabeca *lista)
```




```
if(vazia(lista)) return; e if(!vazia(lista))
```

```
void remover_inicio(cabeca *lista)
```



lixo = lista->prox e lista->prox = lixo->prox

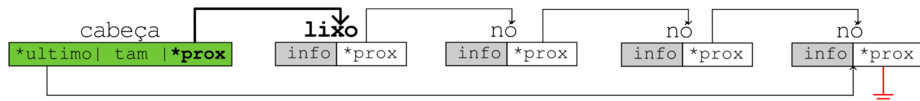
```
void remover_inicio(cabeca *lista)
```

```
if(vazia(lista)) lista->ultimo = NULL
```

```c
void remover_inicio(cabeca *lista)
{
    if(vazia(lista)) return;

    no *lixo = lista->prox;
    lista->prox = lixo->prox;

    //metadados
    lista->num_itens--;
    if(vazia(lista)) lista->ultimo = NULL;

    //free(lixo)?

}
```

```
void remover_no(cabeca *lista, no *lixo)
```

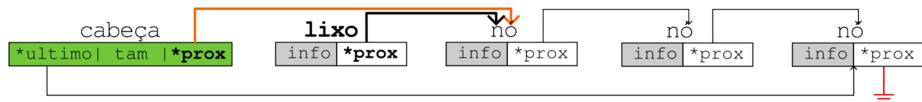
Sendo: `lixo!=NULL`




```
if(lista->prox == lixo) ???
```

```
void remover_no(cabeca *lista, no *lixo)
```

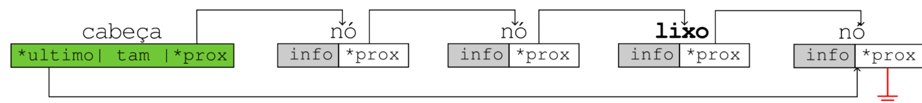
Sendo: `lixo!=NULL`




```
if(lista->prox == lixo) lista->prox = lixo->prox;
```

```c
void remover_no(cabeca *lista, no *lixo)
```

Sendo: `lixo!=NULL`




```c
if(lista->prox != lixo) ???
```

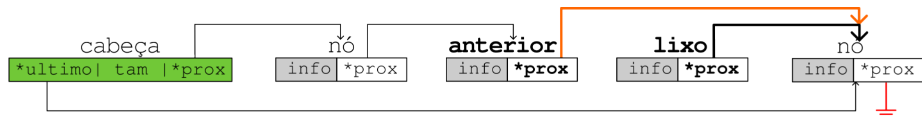
```c
void remover_no(cabeca *lista, no *lixo)
```

Sendo: `lixo!=NULL`

`procurar o anterior e anterior->prox ???`

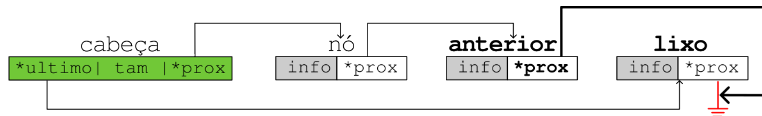
```
void remover_no(cabeca *lista, no *lixo)
```

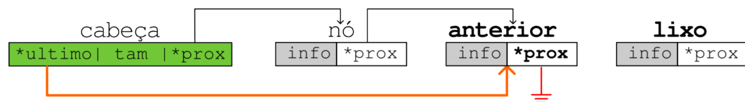
Sendo: `lixo!=NULL`




```
anterior->prox = lixo->prox
```

```
void remover_no(cabeca *lista, no *lixo)
```
Sendo: `lixo!=NULL`






```
if(anterior->prox == NULL) lista->ultimo = anterior
```

```c
void remover_no(cabeca *lista, no *lixo)
{
    if(lista->prox == lixo)
        lista->prox = lixo->prox; //return remover_inicio(lista
    );

    no *ant;
    for(ant = lista->prox;
        ant != NULL && ant->prox!=lixo;
        ant = ant->prox);

    if(ant) {
        ant->prox = lixo->prox;
        lista->num_itens--;
        if(ant->prox==NULL) lista->ultimo = ant;
    }

    //free(lixo)?
}
```

LISTA SIMPLESMENTE ENCADEADAS

- Implementado na STL (Standard Template Library) do C++
  - https://en.cppreference.com/w/
- Implementado na <sys/queue.h> da libc
  - man queue
  - man slist
  - man list
  - etc.

```c
#include <sys/queue.h>

SLIST_ENTRY(TYPE);

SLIST_HEAD(HEADNAME, TYPE);
SLIST_HEAD SLIST_HEAD_INITIALIZER(SLIST_HEAD head);
void SLIST_INIT(SLIST_HEAD *head);

int SLIST_EMPTY(SLIST_HEAD *head);

void SLIST_INSERT_HEAD(SLIST_HEAD *head,
                    struct TYPE *elm, SLIST_ENTRY NAME);
void SLIST_INSERT_AFTER(struct TYPE *listelm,
                        struct TYPE *elm, SLIST_ENTRY NAME);

struct TYPE *SLIST_FIRST(SLIST_HEAD *head);
struct TYPE *SLIST_NEXT(struct TYPE *elm, SLIST_ENTRY NAME);

SLIST_FOREACH(struct TYPE *var, SLIST_HEAD *head, SLIST_ENTRY NAME);

void SLIST_REMOVE(SLIST_HEAD *head, struct TYPE *elm,
                        SLIST_ENTRY NAME);
void SLIST_REMOVE_HEAD(SLIST_HEAD *head,
                        SLIST_ENTRY NAME);
```

```c
//man slist
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/queue.h>

struct entry {
    int data;
    /* Singly linked list - prox */
    SLIST_ENTRY(entry) entries;
};

SLIST_HEAD(slisthead, entry);

int main(void) {
    struct entry *n1, *n2, *n3, *np;
    /* Singly linked list - head */
    struct slisthead head;

    /* Initialize the queue */
    SLIST_INIT(&head);

    /* Insert at the head */
    n1 = malloc(sizeof(struct entry));
    SLIST_INSERT_HEAD(&head, n1, entries);

    /* Insert after */
    n2 = malloc(sizeof(struct entry));
    SLIST_INSERT_AFTER(n1, n2, entries);

    /* Deletion */
    SLIST_REMOVE(&head, n2, entry, entries);
    free(n2);

    n3 = SLIST_FIRST(&head);
    /* Deletion from the head */
    SLIST_REMOVE_HEAD(&head, entries);
    free(n3);

    for (unsigned int i = 0; i < 5; i++) {
        n1 = malloc(sizeof(struct entry));
        SLIST_INSERT_HEAD(&head, n1, entries);
        n1->data = i;
    }

    /* Forward traversal */
    SLIST_FOREACH(np, &head, entries)
        printf("%i\n", np->data);

    /* List deletion */
    while (!SLIST_EMPTY(&head)) {
        n1 = SLIST_FIRST(&head);
        SLIST_REMOVE_HEAD(&head, entries);
        free(n1);
    }

    SLIST_INIT(&head);
    exit(EXIT_SUCCESS);
}
```

```c
struct celula {
  int l, c;
  SLIST_ENTRY(celula) prox;
};

SLIST_HEAD(head, celula);
int abrir_mapa(int m, int n, struct area campo[m][n], int l, int c) {
  campo[l][c].visivel = 1;
  if(campo[l][c].item!=0) return campo[l][c].item;

  struct celula *no = malloc(sizeof no);
  struct head fila;

  no->l = l;
  no->c = c;

  SLIST_INIT(&fila);
  SLIST_INSERT_HEAD(&fila, no, prox);

  while (!SLIST_EMPTY(&fila)) {
    struct celula *no2 = SLIST_FIRST(&fila);
    struct celula *ultimo = no2;
    for(int i=no2->l-1; i<=no2->l+1; i++){
        for(int j=no2->c-1; j<=no2->c+1; j++){
            if(i>=0 && i<m && j>=0 && j<n && campo[i][j].visivel==0) {
                campo[i][j].visivel = 1;
                if(campo[i][j].item==0){
                    struct celula *no3 = malloc(sizeof no3);
                    no3->l = i; no3->c = j;
                    SLIST_INSERT_AFTER(ultimo, no3, prox);
                    ultimo = no3;
                }
            }
        }
    }
    SLIST_REMOVE_HEAD(&fila, prox);
    free(no2);
  }
  return 0;
}
```

LISTA SIMPLESMENTE ENCADEADAS - Exercícios

1. Escreva uma função que conte o número de células de uma lista encadeada.
2. Escreva uma função que concatene duas listas encadeadas.
3. Escreva uma função que insira uma nova célula com conteúdo x imediatamente depois da k-ésima célula de uma lista encadeada.
4. Escreva uma função que troque de posição duas células de uma mesma lista encadeada.

LISTA SIMPLESMENTE ENCADEADAS - Exercícios

1. **Altura**. A altura de uma célula c em uma lista encadeada é a **distância entre c e o fim da lista**. Escreva uma função que calcule a altura de uma dada célula.

2. **Profundidade**. A profundidade de uma célula c em uma lista encadeada é **distância entre o início da lista e c**. Escreva uma função que calcule a profundidade de uma dada célula.

3. Escreva uma função que inverta a ordem das células de uma lista encadeada (a primeira passa a ser a última, a segunda passa a ser a penúltima etc.). Faça isso sem usar espaço auxiliar, apenas alterando ponteiros. Dê duas soluções: uma iterativa e uma recursiva.

LISTA SIMPLESMENTE ENCADEADAS - Exercícios

1. Escreva uma função que encontre uma célula com **conteúdo mínimo**. Faça duas versões: uma iterativa e uma recursiva.

2. Escreva uma função para remover de uma lista encadeada todas as células que contêm y.

3. Escreva uma função que verifique se **duas listas encadeadas são iguais**, ou melhor, se têm o mesmo conteúdo. Faça duas versões: uma iterativa e uma recursiva.

4. Listas de strings. Este exercício trata de listas encadeadas que contêm strings ASCII (cada célula contém uma string). Escreva uma função que verifique se uma lista desse tipo está em ordem lexicográfica. As células são do seguinte tipo:

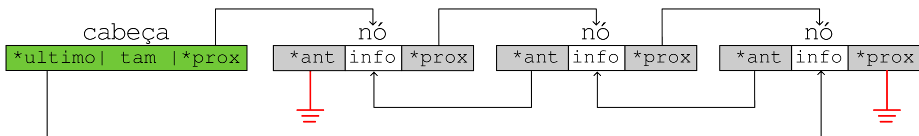
```
1 typedef struct reg {
2    char *str; struct reg *prox;
3 } celula;
```

Roteiro

1. ESTRUTURAS DE DADOS ELEMENTARES
2. LISTAS ESTÁTICAS
3. LISTAS SIMPLESMENTE ENCADEADAS
4. LISTA DUPLAMENTE ENCADEADAS

LISTA DUPLAMENTE ENCADEADAS

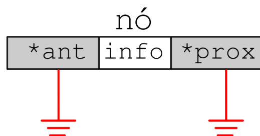
- Armazena a informação do nó **anterior** e **posterior**
- Útil quando ocorrem muitas inserções e remoções, principalmente de elementos intermediários
- Anterior do primeiro e posterior do último: NULL



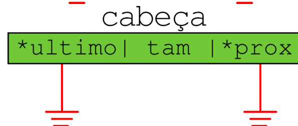
LISTA DUPLAMENTE ENCADEADAS

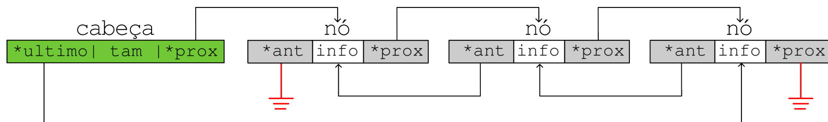
Lista com cabeça

```c
typedef struct node no;
struct node {
    Item info;
    no *ant; //<<<<
    no *prox;
};
```




```c
typedef struct head cabeca;
struct head {
    int tam;
    no *prox;
    no *ultimo;
};
```





LISTA DUPLAMENTE ENCADEADAS

Operações básicas

```
typedef int Item;

typedef struct registro no;
struct registro {
    Item info;
    no *ant; //<<<<<<<<<<<<<<<<<<<<<<<<<<<
    no *prox;
    no *node;
};

//tipo cabeca
typedef struct {
    int tam;
    no *prox;
    no *ultimo;
} cabeca;

//PROTÓTIPO DAS OPERAÇÕES BÁSICAS
cabeca *criar();
no *criar_no(Item, no *);

```

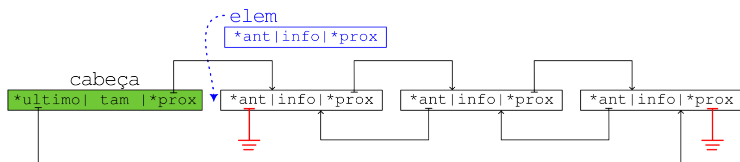
LISTA DUPLAMENTE ENCADEADAS

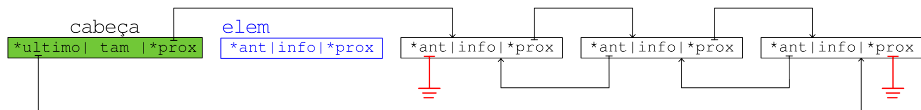
Operações básicas

```
22 int vazia(cabeca*);
23 int tamanho(cabeca*);
24
25 no *primeiro(cabeca *);
26 no *ultimo(cabeca *);
27 no *proximo(no *);
28 no *anterior(no *);
29
30 void inserir_inicio(cabeca *, no *);
31 void inserir_antes(cabeca *, no *, no *);
32 void inserir_depois(cabeca *, no *, no *);
33 void inserir_fim(cabeca *, no *);
34
35 no* remover_no(cabeca *, no *);
36 void imprime(cabeca *lista);
```

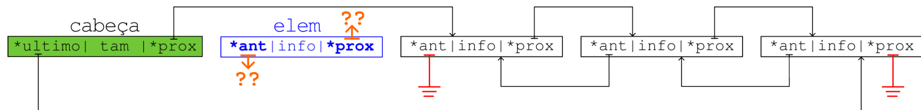
```c
cabeca *criar() {
    cabeca *l = malloc(sizeof(cabeca));
    l->tam = 0;
    l->prox = NULL;
    l->ultimo = NULL;

    return l;
}

no *criar_no(Item x) {
    no *novo = malloc(sizeof(no));
    novo->ant = NULL;  //<<<<<<<<
    novo->prox = NULL;
    novo->info = x;

    return novo;
}

int vazia(cabeca *lista) {
    return (lista->prox==NULL);
}

no *buscar(cabeca *lista, Item x) {
    no *a;
    for(a=lista->prox; a && a->info!=x; a=a->prox);
    return a;
}
```

```
void inserir_inicio(cabeca *lista, no *elem)
```



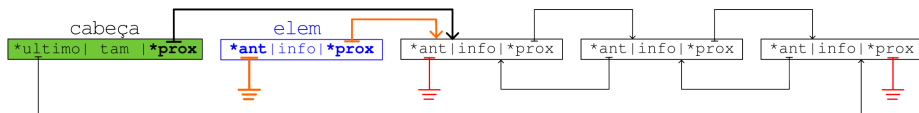


void inserir_inicio(cabeca *lista, no *elem)



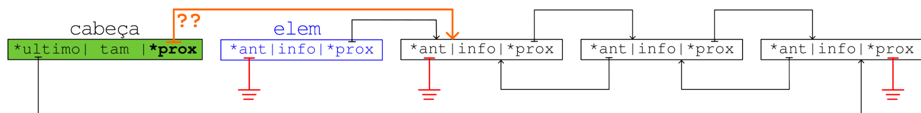

```
1  void inserir_inicio(cabeca *lista, no *elem)
2  {
3      elem->ant = ???
4      elem->prox = ???
5
6
7
8
9
10
11 }
```

void inserir_inicio(cabeca *lista, no *elem)



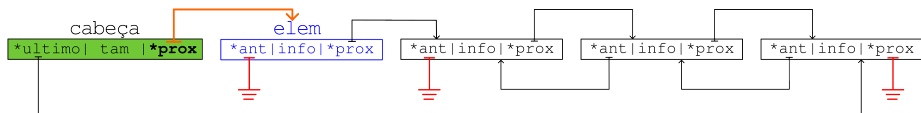

```
1  void inserir_inicio(cabeca *lista, no *elem)
2  {
3      elem->ant = NULL;
4      elem->prox = lista->prox;
5
6
7
8
9
10
11 }
```

void inserir_inicio(cabeca *lista, no *elem)



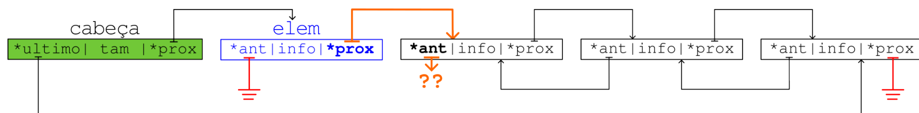

```c
void inserir_inicio(cabeca *lista, no *elem)
{
    elem->ant = NULL;
    elem->prox = lista->prox;
    lista->prox = ???





}
```

void inserir_inicio(cabeca *lista, no *elem)



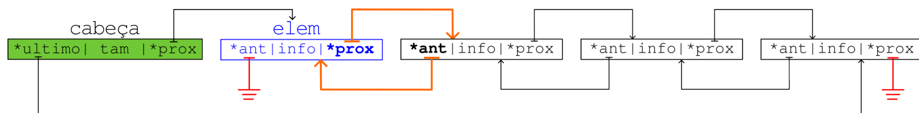

```c
void inserir_inicio(cabeca *lista, no *elem)
{
    elem->ant = NULL;
    elem->prox = lista->prox;
    lista->prox = elem;





}
```

void inserir_inicio(cabeca *lista, no *elem)



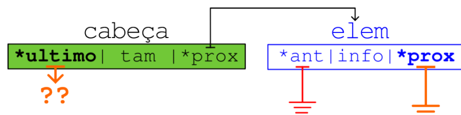

```
1  void inserir_inicio(cabeca *lista, no *elem)
2  {
3      elem->ant = NULL;
4      elem->prox = lista->prox;
5      lista->prox = elem;
6
7      if(elem->prox != NULL) ???
8
9
10
11 }
```

void inserir_inicio(cabeca *lista, no *elem)



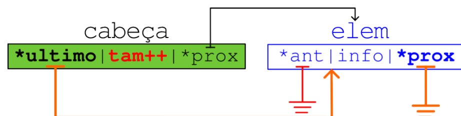

```
1  void inserir_inicio(cabeca *lista, no *elem)
2  {
3      elem->ant = NULL;
4      elem->prox = lista->prox;
5      lista->prox = elem;
6
7      if(elem->prox != NULL) elem->prox->ant = elem;
8
9
10
11 }
```

void inserir_inicio(cabeca *lista, no *elem)



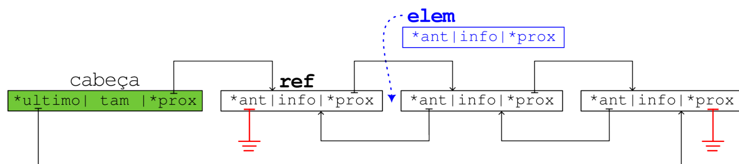

```c
void inserir_inicio(cabeca *lista, no *elem)
{
    elem->ant = NULL;
    elem->prox = lista->prox;
    lista->prox = elem;

    if(elem->prox != NULL) elem->prox->ant = elem;
    if(elem->prox == NULL) ???


}
```

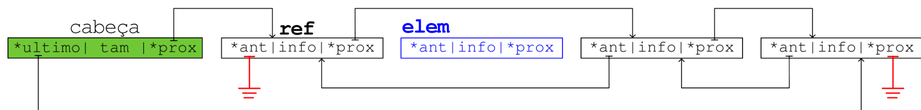
void inserir_inicio(cabeca *lista, no *elem)



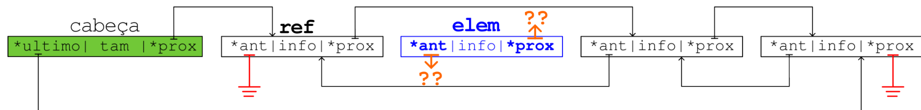

```
1 void inserir_inicio(cabeca *lista, no *elem)
2 {
3     elem->ant = NULL;
4     elem->prox = lista->prox;
5     lista->prox = elem;
6 
7     if(elem->prox) elem->prox->ant = elem;
8     else lista->ultimo = elem;
9 
10    lista->tam++;
11 }
```

```
void inserir_depois(cabeca *lista, no *ref, no *elem)
```



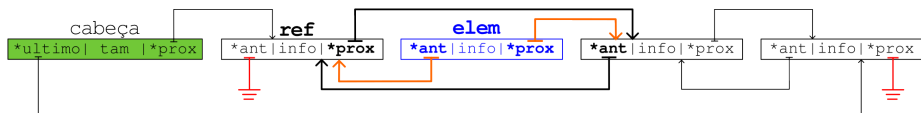


void inserir_depois(cabeca *lista, no *ref, no *elem)



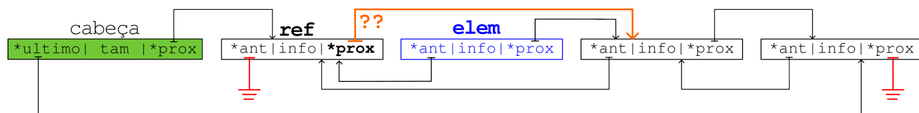

```
1 void inserir_depois(cabeca *lista, no *ref, no *elem)
2 {
3     elem->ant = ???
4     elem->prox = ???
5
6
7
8
9
10
11 }
```

void inserir_depois(cabeca *lista, no *ref, no *elem)



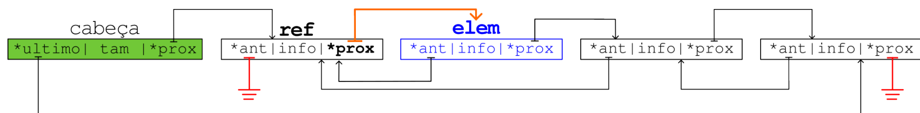

```
1  void inserir_depois(cabeca *lista, no *ref, no *elem)
2  {
3      elem->ant = ref;
4      elem->prox = ref->prox;
5
6
7
8
9
10
11 }
```

void inserir_depois(cabeca *lista, no *ref, no *elem)



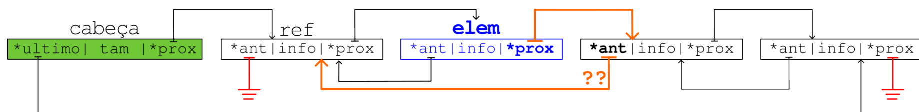

```
1 void inserir_depois(cabeca *lista, no *ref, no *elem)
2 {
3     elem->ant = ref;
4     elem->prox = ref->prox;
5     ref->prox = ???
6
7
8
9
10
11 }
```

void inserir_depois(cabeca *lista, no *ref, no *elem)



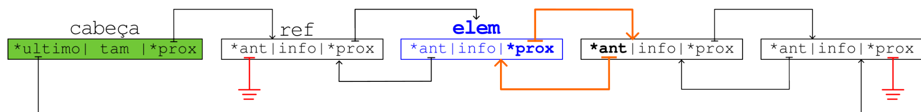

```
1  void inserir_depois(cabeca *lista, no *ref, no *elem)
2  {
3      elem->ant = ref;
4      elem->prox = ref->prox;
5      ref->prox = elem;
6
7
8
9
10
11 }
```

void inserir_depois(cabeca *lista, no *ref, no *elem)



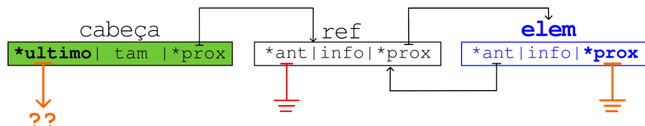

```
1  void inserir_depois(cabeca *lista, no *ref, no *elem)
2  {
3      elem->ant = ref;
4      elem->prox = ref->prox;
5      ref->prox = elem;
6
7      if(elem->prox != NULL) ???
8
9
10
11 }
```

void inserir_depois(cabeca *lista, no *ref, no *elem)



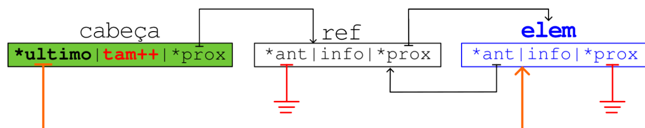

```
1  void inserir_depois(cabeca *lista, no *ref, no *elem)
2  {
3      elem->ant = ref;
4      elem->prox = ref->prox;
5      ref->prox = elem;
6
7      if(elem->prox != NULL) elem->prox->ant = elem;
8
9
10
11 }
```

void inserir_depois(cabeca *lista, no *ref, no *elem)



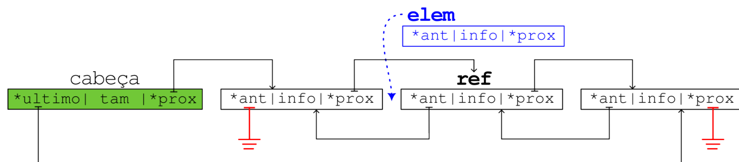

```c
void inserir_depois(cabeca *lista, no *ref, no *elem)
{
    elem->ant = ref;
    elem->prox = ref->prox;
    ref->prox = elem;

    if(elem->prox != NULL) elem->prox->ant = elem;
    if(elem->prox == NULL) ???


}
```

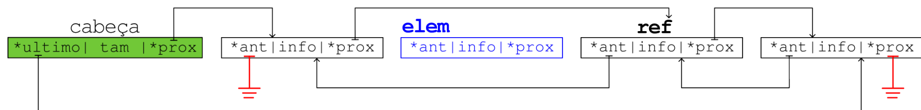
void inserir_depois(cabeca *lista, no *ref, no *elem)



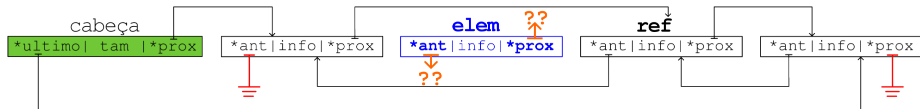

```c
void inserir_depois(cabeca *lista, no *ref, no *elem)
{
    elem->ant = ref;
    elem->prox = ref->prox;
    ref->prox = elem;

    if(elem->prox != NULL) elem->prox->ant = elem;
    else lista->ultimo = elem;

    lista->tam++;
}
```

```
void inserir_antes(cabeca *lista, no *ref, no *elem)
```



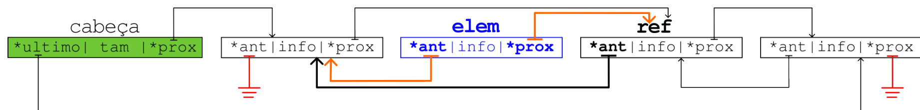


void inserir_antes(cabeca *lista, no *ref, no *elem)



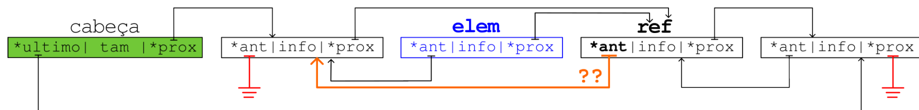

```
1  void inserir_antes(cabeca *lista, no *ref, no *elem)
2  {
3      elem->ant = ???
4      elem->prox = ???
5
6
7
8
9
10
11 }
```

void inserir_antes(cabeca *lista, no *ref, no *elem)



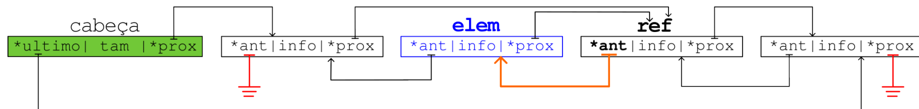

```
1  void inserir_antes(cabeca *lista, no *ref, no *elem)
2  {
3      elem->ant = ref->ant;
4      elem->prox = ref;
5
6
7
8
9
10
11 }
```

void inserir_antes(cabeca *lista, no *ref, no *elem)



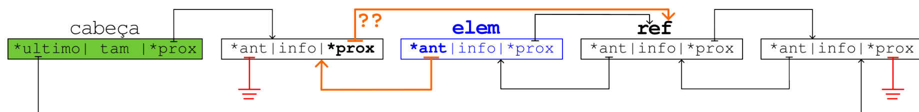

```
1  void inserir_antes(cabeca *lista, no *ref, no *elem)
2  {
3      elem->ant = ref->ant;
4      elem->prox = ref;
5      ref->ant = ???
6
7
8
9
10
11 }
```

void inserir_antes(cabeca *lista, no *ref, no *elem)



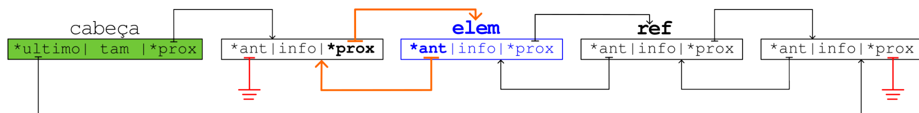

```
1  void inserir_antes(cabeca *lista, no *ref, no *elem)
2  {
3      elem->ant = ref->ant;
4      elem->prox = ref;
5      ref->ant = elem;
6
7
8
9
10
11 }
```

void inserir_antes(cabeca *lista, no *ref, no *elem)



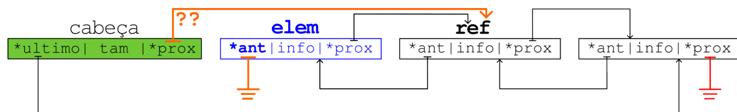

```
1 void inserir_antes(cabeca *lista, no *ref, no *elem)
2 {
3     elem->ant = ref->ant;
4     elem->prox = ref;
5     ref->ant = elem;
6
7     if(elem->ant != NULL) ???
8
9
10
11 }
```

void inserir_antes(cabeca *lista, no *ref, no *elem)



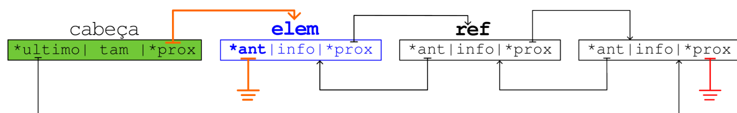

```
1 void inserir_antes(cabeca *lista, no *ref, no *elem)
2 {
3     elem->ant = ref->ant;
4     elem->prox = ref;
5     ref->ant = elem;
6 
7     if(elem->ant != NULL) elem->ant->prox = elem;
8 
9 
10
11 }
```

void inserir_antes(cabeca *lista, no *ref, no *elem)



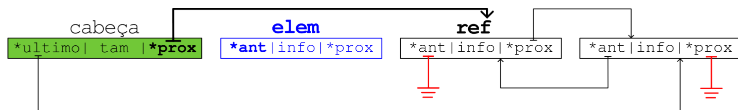

```
1  void inserir_antes(cabeca *lista, no *ref, no *elem)
2  {
3      elem->ant = ref->ant;
4      elem->prox = ref;
5      ref->ant = elem;
6  
7      if(elem->ant != NULL) elem->ant->prox = elem;
8      if(elem->ant == NULL) ???
9  
10 
11 }
```

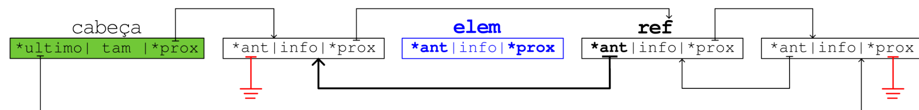
void inserir_antes(cabeca *lista, no *ref, no *elem)




```
1  void inserir_antes(cabeca *lista, no *ref, no *elem)
2  {
3      elem->ant = ref->ant;
4      elem->prox = ref;
5      ref->ant = elem;
6
7      if(elem->ant) elem->ant->prox = elem;
8      else lista->prox = elem;
9
10     lista->tam++;
11 }
```

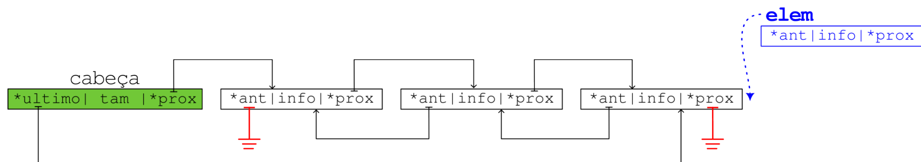
void inserir_antes(cabeca *lista, no *ref, no *elem)

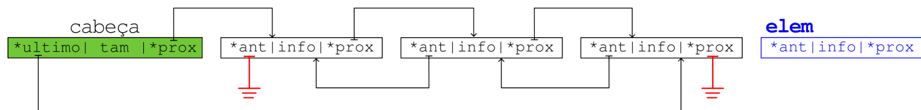




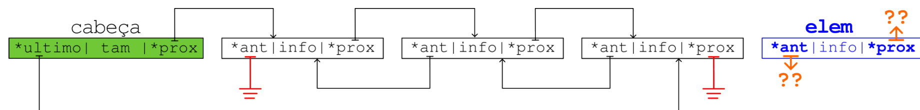

```
1 void inserir_antes(cabeca *lista, no *ref, no *elem)
2 {
3     if(lista->prox == ref)
4         return insere_inicio(lista, elem);
5
6     insere_depois(lista, ref->ant, elem);
7 }
```

void insere_fim(cabeca *lista, no *elem)



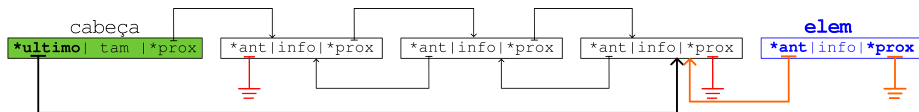


void insere_fim(cabeca *lista, no *elem)



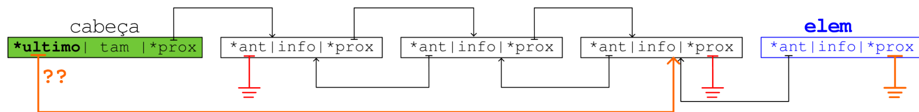

```
1  void inserir_fim(cabeca *lista, no *elem)
2  {
3      elem->ant = ???
4      elem->prox = ???
5
6
7
8
9
10
11
12 }
```

void insere_fim(cabeca *lista, no *elem)



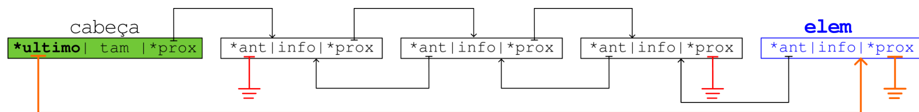

```
1  void inserir_fim(cabeca *lista, no *elem)
2  {
3      elem->ant = lista->ultimo;
4      elem->prox = NULL;
5
6
7
8
9
10
11
12 }
```

void insere_fim(cabeca *lista, no *elem)



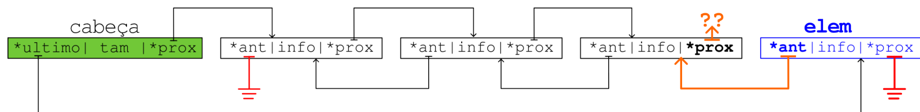

```
1  void inserir_fim(cabeca *lista, no *elem)
2  {
3      elem->ant = lista->ultimo;
4      elem->prox = NULL;
5
6      lista->ultimo = ???
7
8
9
10
11
12 }
```

void insere_fim(cabeca *lista, no *elem)



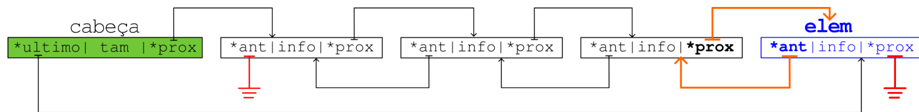

```
1  void inserir_fim(cabeca *lista, no *elem)
2  {
3      elem->ant = lista->ultimo;
4      elem->prox = NULL;
5
6      lista->ultimo = elem;
7
8
9
10
11
12 }
```

void insere_fim(cabeca *lista, no *elem)



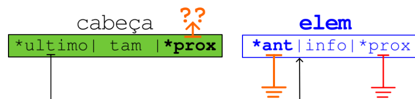

```c
void inserir_fim(cabeca *lista, no *elem)
{
    elem->ant = lista->ultimo;
    elem->prox = NULL;

    lista->ultimo = elem;

    if(elem->ant != NULL) ???



}
```

void insere_fim(cabeca *lista, no *elem)



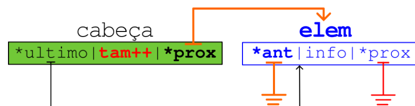

```
1  void inserir_fim(cabeca *lista, no *elem)
2  {
3      elem->ant = lista->ultimo;
4      elem->prox = NULL;
5
6      lista->ultimo = elem;
7
8      if(elem->ant != NULL) elem->ant->prox = elem;
9
10
11
12 }
```

void insere_fim(cabeca *lista, no *elem)



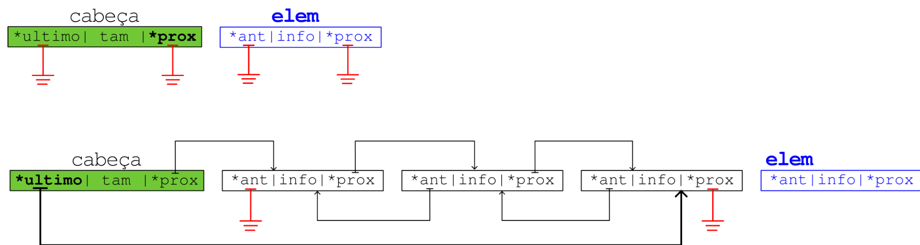

```
1  void inserir_fim(cabeca *lista, no *elem)
2  {
3      elem->ant = lista->ultimo;
4      elem->prox = NULL;
5
6      lista->ultimo = elem;
7
8      if(elem->ant != NULL) elem->ant->prox = elem;
9      if(elem->ant == NULL) ???
10
11
12 }
```

void insere_fim(cabeca *lista, no *elem)



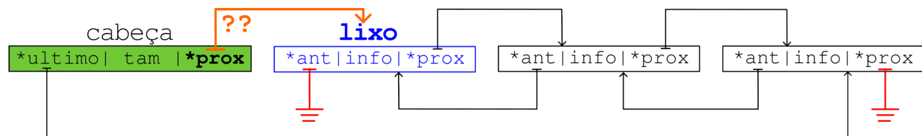

```c
void inserir_fim(cabeca *lista, no *elem)
{
    elem->ant = lista->ultimo;
    elem->prox = NULL;

    lista->ultimo = elem;

    if(elem->ant) elem->ant->prox = elem;
    else lista->prox = elem;

    lista->tam++;
}
```

void insere_fim(cabeca *lista, no *elem)



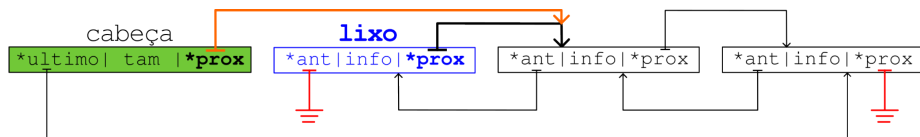

```c
void inserir_fim(cabeca *lista, no *elem)
{
    if(vazia(lista))
        return insere_inicio(lista, elem);
    insere_depois(lista, lista->ultimo, elem);
}
```

void remover_no(cabeca *lista, no *lixo)



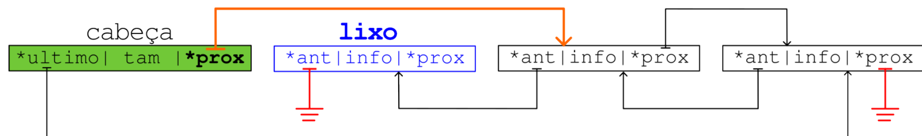

```
1  void remover_no(cabeca *lista, no *lixo)
2  {
3
4      if(lista->prox == lixo) ???
5
6
7
8
9
10
11
12 }
```

void remover_no(cabeca *lista, no *lixo)



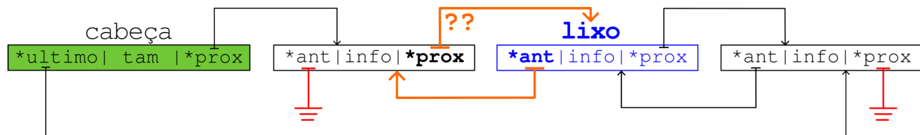

```
1 void remover_no(cabeca *lista, no *lixo)
2 {
3
4     if(lista->prox == lixo) lista->prox = lixo->prox;
5
6
7
8
9
10
11
12 }
```

void remover_no(cabeca *lista, no *lixo)



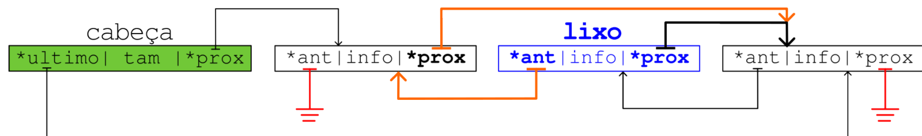

```c
void remover_no(cabeca *lista, no *lixo)
{

    if(lista->prox == lixo) lista->prox = lixo->prox;








}
```

void remover_no(cabeca *lista, no *lixo)



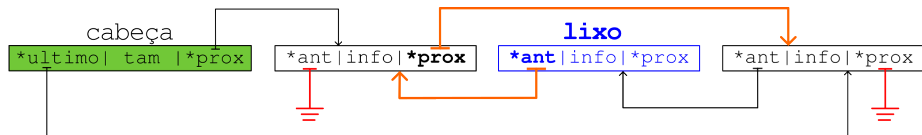

```
1  void remover_no(cabeca *lista, no *lixo)
2  {
3
4      if(lista->prox == lixo) lista->prox = lixo->prox;
5      if(lista->prox != lixo) ???
6
7
8
9
10
11
12 }
```

void remover_no(cabeca *lista, no *lixo)



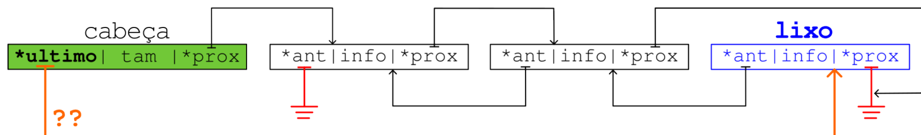

```
1  void remover_no(cabeca *lista, no *lixo)
2  {
3
4      if(lista->prox == lixo) lista->prox = lixo->prox;
5      else lixo->ant->prox = lixo->prox;
6
7
8
9
10
11
12 }
```

void remover_no(cabeca *lista, no *lixo)



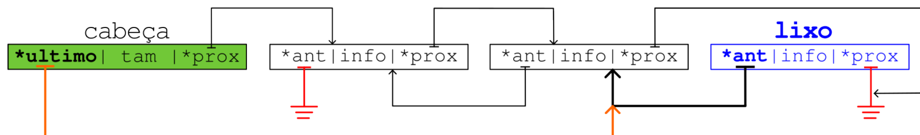

```
1  void remover_no(cabeca *lista, no *lixo)
2  {
3
4      if(lista->prox == lixo) lista->prox = lixo->prox;
5      else lixo->ant->prox = lixo->prox;
6
7
8
9
10
11
12 }
```

void remover_no(cabeca *lista, no *lixo)



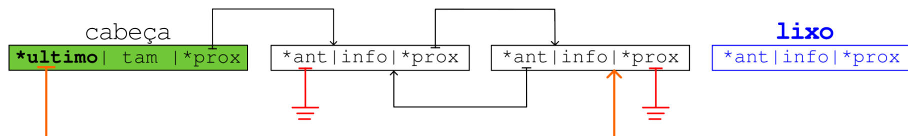

```
1  void remover_no(cabeca *lista, no *lixo)
2  {
3
4      if(lista->prox == lixo) lista->prox = lixo->prox;
5      else lixo->ant->prox = lixo->prox;
6
7      if(lista->ultimo == lixo) ???
8
9
10
11
12 }
```

void remover_no(cabeca *lista, no *lixo)



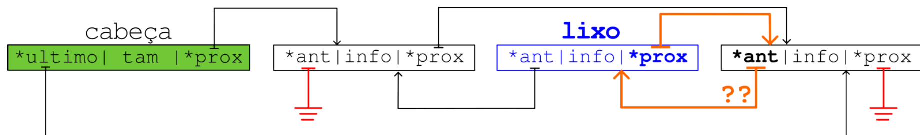

```
1  void remover_no(cabeca *lista, no *lixo)
2  {
3
4      if(lista->prox == lixo) lista->prox = lixo->prox;
5      else lixo->ant->prox = lixo->prox;
6
7      if(lista->ultimo == lixo) lista->ultimo = lixo->ant;
8
9
10
11
12 }
```

void remover_no(cabeca *lista, no *lixo)



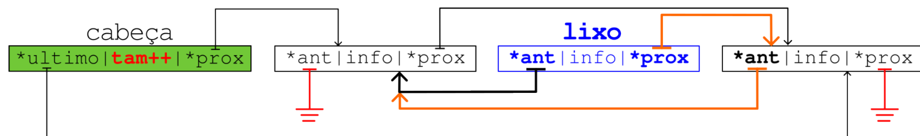

```c
void remover_no(cabeca *lista, no *lixo)
{

    if(lista->prox == lixo) lista->prox = lixo->prox;
    else lixo->ant->prox = lixo->prox;

    if(lista->ultimo == lixo) lista->ultimo = lixo->ant;




}
```

void remover_no(cabeca *lista, no *lixo)



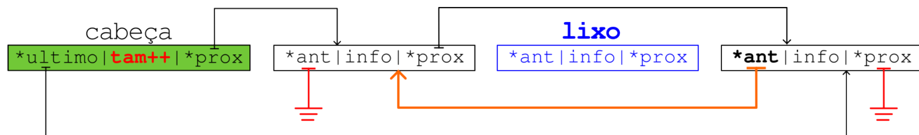

```
1  void remover_no(cabeca *lista, no *lixo)
2  {
3
4      if(lista->prox == lixo) lista->prox = lixo->prox;
5      else lixo->ant->prox = lixo->prox;
6
7      if(lista->ultimo == lixo) lista->ultimo = lixo->ant;
8      if(lista->ultimo != lixo) ???
9
10
11
12 }
```

void remover_no(cabeca *lista, no *lixo)




```c
void remover_no(cabeca *lista, no *lixo)
{

    if(lista->prox == lixo) lista->prox = lixo->prox;
    else lixo->ant->prox = lixo->prox;

    if(lista->ultimo == lixo) lista->ultimo = lixo->ant;
    else lixo->prox->ant = lixo->ant;

    lista->tam--;

}
```

void remover_no(cabeca *lista, no *lixo)




```
1 void remover_no(cabeca *lista, no *lixo)
2 {
3
4     if(lista->prox == lixo) lista->prox = lixo->prox;
5     else lixo->ant->prox = lixo->prox;
6
7     if(lista->ultimo == lixo) lista->ultimo = lixo->ant;
8     else lixo->prox->ant = lixo->ant;
9
10    lista->tam--;
11
12 }
```

```c
//man list
#include <sys/queue.h>

LIST_ENTRY(TYPE);

LIST_HEAD(HEADNAME, TYPE);
LIST_HEAD LIST_HEAD_INITIALIZER(LIST_HEAD head);
void LIST_INIT(LIST_HEAD *head);

int LIST_EMPTY(LIST_HEAD *head);

void LIST_INSERT_HEAD(LIST_HEAD *head,
                        struct TYPE *elm, LIST_ENTRY NAME);
void LIST_INSERT_BEFORE(struct TYPE *listelm,
                        struct TYPE *elm, LIST_ENTRY NAME);
void LIST_INSERT_AFTER(struct TYPE *listelm,
                        struct TYPE *elm, LIST_ENTRY NAME);

struct TYPE *LIST_FIRST(LIST_HEAD *head);
struct TYPE *LIST_NEXT(struct TYPE *elm, LIST_ENTRY NAME);

LIST_FOREACH(struct TYPE *var, LIST_HEAD *head, LIST_ENTRY NAME);

void LIST_REMOVE(struct TYPE *elm, LIST_ENTRY NAME);
```

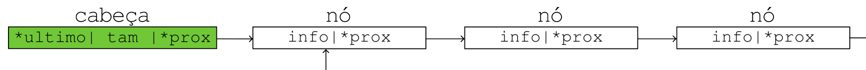
```c
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/queue.h>

struct entry {
    int data;
    LIST_ENTRY(entry) entries; /* List */
};

LIST_HEAD(listhead, entry);

int main(void)
{
    struct entry *n1, *n2, *n3, *np;
    /* List head */
    struct listhead head;
    int i;

    /* Initialize the list*/
    LIST_INIT(&head);

    /* Insert at the head */
    n1 = malloc(sizeof(struct entry));
    LIST_INSERT_HEAD(&head, n1, entries);

    /* Insert after */
    n2 = malloc(sizeof(struct entry));
    LIST_INSERT_AFTER(n1, n2, entries);

    /* Insert before */
    n3 = malloc(sizeof(struct entry));
    LIST_INSERT_BEFORE(n2, n3, entries);

    /* Forward traversal */
    i = 0;
    LIST_FOREACH(np, &head, entries)
        np->data = i++;

    /* Deletion */
    LIST_REMOVE(n2, entries);
    free(n2);

    /* Forward traversal */
    LIST_FOREACH(np, &head, entries)
        printf("%i\n", np->data);

    /* List deletion */
    n1 = LIST_FIRST(&head);
    while (n1 != NULL) {
        n2 = LIST_NEXT(n1, entries);
        free(n1);
        n1 = n2;
    }
    LIST_INIT(&head);
}
```

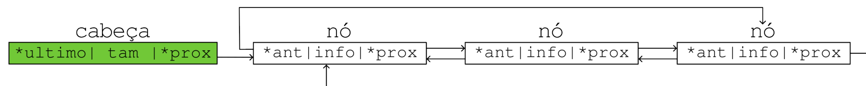
LISTA DUPLAMENTE ENCADEADA - Exercícios

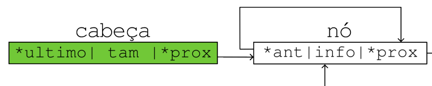
```c
/*  Escreva uma função que imprime o reverso de uma lista*/
void imprime_reverso(cabeca *lista);

/*  Escreva uma função que troque dois nós uma lista através de
     reapontamentos*/
void troca_nos(cabeca *lista, no *no1, no *no2);

/*  Escreva uma função que inverta os nós de uma lista, sem
    utilizar outra lista*/
void inverte_lista(cabeca *lista);

/*  Escreva uma função que concatene duas listas*/
void concatena(cabeca *lista1, cabeca *lista2);
```

Outras listas encadeadas

- Circular:
  - Simplesmente: último aponta para o primeiro
  - Duplamente: primeiro elemento aponta para o último e vice-versa
  - `lista->ultimo` ??
  - Único elemento da lista ??
  - Implementem as operações básicas!!

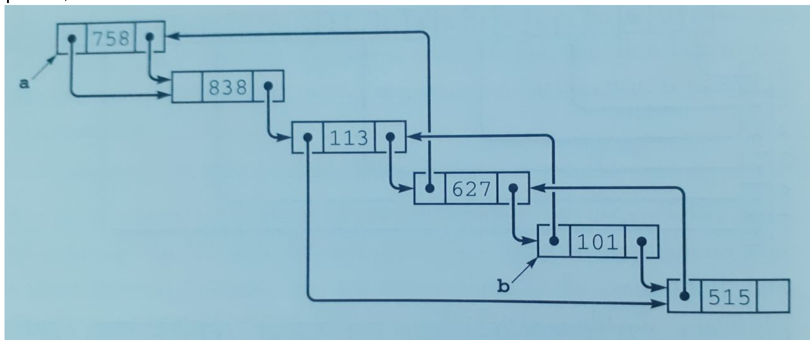






Outras listas encadeadas

- Multilista:
  - Apontamentos para o próximo e anterior são independentes, não necessariamente um nó aponta de volta para o nó que aponta para ele
  - Exemplo: começando por A, temos a ordem na qual os itens foram inseridos; por B, temos a lista ordenada



- Duplamente encadeada:
  - Multilista
  - x->prox->ant = x = x->ant->prox

Listas estáticas (array) - relembrando

- Alocação contígua/sequencial na memória
- Acesso aleatório e direto pelo índice/posição: tempo constante
- Inserção: tempo constante
- Remoções no meio: tempo linear (percorrer a lista)
- Busca sequencial: tempo linear (percorrer a lista)

Listas encadeadas - relembrando

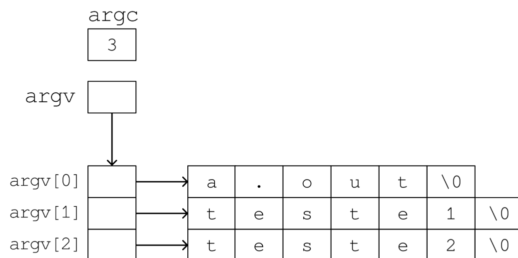
- Sequência de células/nós
- Alocação conforme necessidade
- Remoção de um nó específico:
  - Simples: tempo linear
  - Dupla: tempo constante
- Inserção antes/após um nó específico:
  - Simples: tempo linear(antes), tempo constante(após)
  - Dupla: tempo constante
- Busca sequencial
- Sem acesso direto a uma posição
- Implementações:
  - Sem cabeça, com cabeça, com cauda, com cabeça e cauda
  - Cabeça do mesmo tipo dos elementos do corpo
  - Cabeça com elementos diferentes do corpo
- Tipos: simplesmente, duplamente, circular, multilista

Composição de estruturas

- Vetor de strings

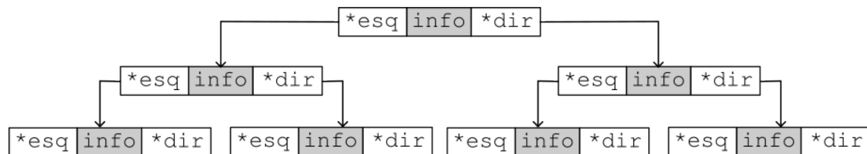
```c
int main(int argc, char *argv[ ]){
    for(int cont=0; cont < argc; cont++)
        printf("%s\n", argv[cont]);

    return 0;
}
```

./a.out teste1 teste2



Composição de estruturas

- Multilistas: representação de árvores



Composição de estruturas

- Vetor de vetores (matriz)
  - Matriz estática: `int mat[10][10];`
  - **Matrizes com listas encadeadas**:
    - Bom **desempenho espacial** em casos de **matrizes esparsas**
    - Matriz esparsa: grande quantidade dos elementos são não-válidos (zeros)
    - Esparsidade/dispersão aparece em:

      - Mineração de dados, análises numéricas, combinatórias,

      - Aplicações científicas e de engenharia: fenômenos eletrostática, eletrodinâmica, eletromagnetismo, dinâmica dos fluidos, difusão do calor, propagação de ondas

      - Tabela hash: estrutura de dados que associa chave de pesquisa a valores de indíces - ótimo desempenho em inserções, remoções e buscas

      - Grafos: par de conjuntos relacionados

Exemplo de aplicação: Grafos

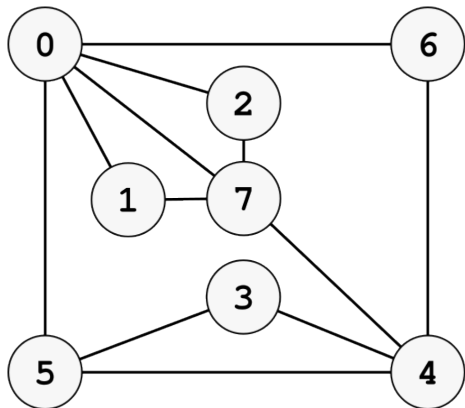
- É um par de conjuntos:
  - Conjunto de entidades conhecidas como vértices + um conjunto de entidades conhecidas como arcos (arestas)
  - Cada aresta é um par ordenado de vértices (liga): primeiro vértice do par é a ponta inicial e o segundo é a ponta final
- A organização em grafos contribui para a representação de problemas com rotas, combinatórias, logística, fluxo, etc.
- Auxiliando em operações para definir o melhor (menor, mais rápido) caminho, geração de preferências, perfis, classificações, hierarquias, combinações, etc.

Exemplo de aplicação: Grafos

- Exemplos:
  - Escalonamento de vôos, transporte de mercadorias, mapas, rede de computadores, links nas páginas web, relacionamento nas redes sociais, banco de dados, máquinas de aprendizagem, mineração de dados, busca na Internet, escalonamento de tempo, engine de jogos (ex.: `https://docs.cocos2d-x.org/cocos2d-x/v4/en/basic_concepts/scene.html`).
- Possível representação por uma matriz de adjacências
  - Dado matriz[i][j] = 1, diz-se que o vértice i está conectado ao vértice j
  - Linhas e colunas representam os vértices
  - Quando há poucas conexões, com muitos dos pares (i,j) iguais a 0, diz-se que é uma matriz esparsa
  - Grafos esparsos, são eficientemente representados com **listas de adjacências (array de listas)**

Exemplo de aplicação: matriz de adjacências

- Dado matriz[i][j] = 1, diz-se que o vértice i está conectado ao vértice j
- Exemplo: hierarquia entre componentes de um jogo, ligações entre pessoas (rede sociais), mapas, etc.

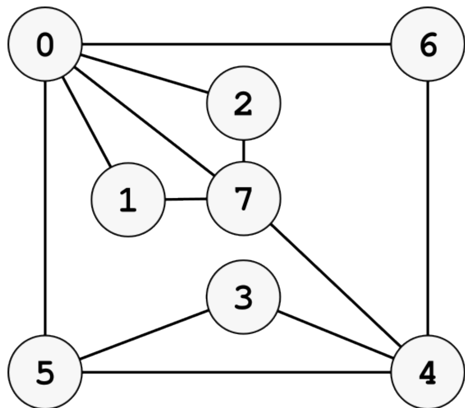


Matriz de adjacências

	0	1	2	3	4	5	6	7
0	1	1	1	0	0	1	1	1
1	1	1	0	0	0	0	0	1
2	1	0	1	0	0	0	0	1
3	0	0	0	1	1	1	0	0
4	0	0	0	1	1	1	1	1
5	1	0	0	1	1	1	0	0
6	1	0	0	0	1	0	1	0
7	1	1	1	0	1	0	0	1

Exemplo de aplicação: matriz de adjacências

- Dado matriz[i][j] = 1, diz-se que o vértice i está conectado ao vértice j
- Exemplo: hierarquia entre componentes de um jogo, ligações entre pessoas (rede sociais), mapas, etc.

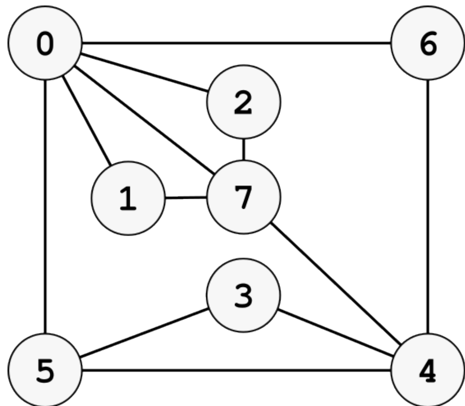


Matriz de adjacências

	0	1	2	3	4	5	6	7
0	1	1	1	0	0	1	1	1
1	1	1	0	0	0	0	0	1
2	1	0	1	0	0	0	0	1
3	0	0	0	1	1	1	0	0
4	0	0	0	1	1	1	1	1
5	1	0	0	1	1	1	0	0
6	1	0	0	0	1	0	1	0
7	1	1	1	0	1	0	0	1

Exemplo de aplicação: matriz de adjacências

- Dado matriz[i][j] = 1, diz-se que o vértice i está conectado ao vértice j
- Exemplo: hierarquia entre componentes de um jogo, ligações entre pessoas (rede sociais), mapas, etc.

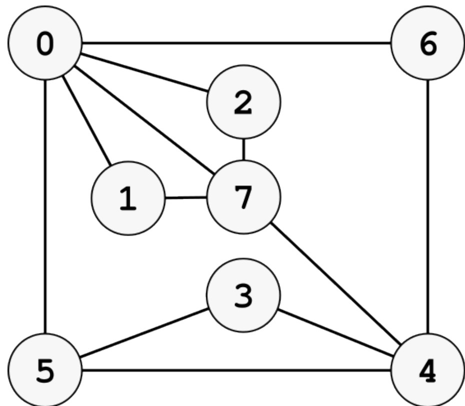


Matriz de adjacências

	0	1	2	3	4	5	6	7
0	1	1	1	0	0	1	1	1
1	1	1	0	0	0	0	0	1
2	1	0	1	0	0	0	0	1
3	0	0	0	1	1	1	0	0
4	0	0	0	1	1	1	1	1
5	1	0	0	1	1	1	0	0
6	1	0	0	0	1	0	1	0
7	1	1	1	0	1	0	0	1

Exemplo de aplicação: matriz de adjacências

- Dado matriz[i][j] = 1, diz-se que o vértice i está conectado ao vértice j
- Exemplo: hierarquia entre componentes de um jogo, ligações entre pessoas (rede sociais), mapas, etc.

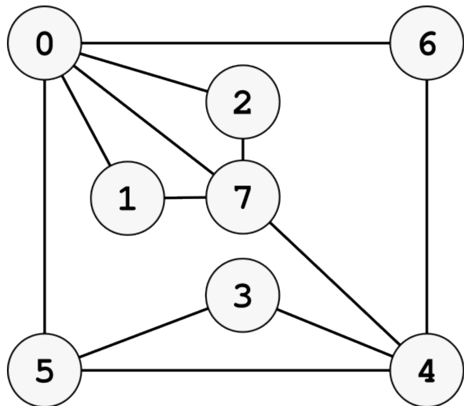


Matriz de adjacências

	0	1	2	3	4	5	6	7
0	1	1	1	0	0	1	1	1
1	1	1	0	0	0	0	0	1
2	1	0	1	0	0	0	0	1
3	0	0	0	1	1	1	0	0
4	0	0	0	1	1	1	1	1
5	1	0	0	1	1	1	0	0
6	1	0	0	0	1	0	1	0
7	1	1	1	0	1	0	0	1

Exemplo de aplicação: matriz de adjacências

- Dado matriz[i][j] = 1, diz-se que o vértice i está conectado ao vértice j
- Exemplo: hierarquia entre componentes de um jogo, ligações entre pessoas (rede sociais), mapas, etc.

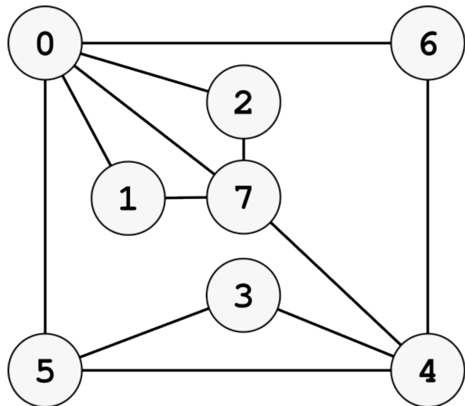


Matriz de adjacências

	0	1	2	3	4	5	6	7
0	1	1	1	0	0	1	1	1
1	1	1	0	0	0	0	0	1
2	1	0	1	0	0	0	0	1
3	0	0	0	1	1	1	0	0
4	0	0	0	1	1	1	1	1
5	1	0	0	1	1	1	0	0
6	1	0	0	0	1	0	1	0
7	1	1	1	0	1	0	0	1

Exemplo de aplicação: matriz de adjacências

- Dado matriz[i][j] = 1, diz-se que o vértice i está conectado ao vértice j
- Exemplo: hierarquia entre componentes de um jogo, ligações entre pessoas (rede sociais), mapas, etc.

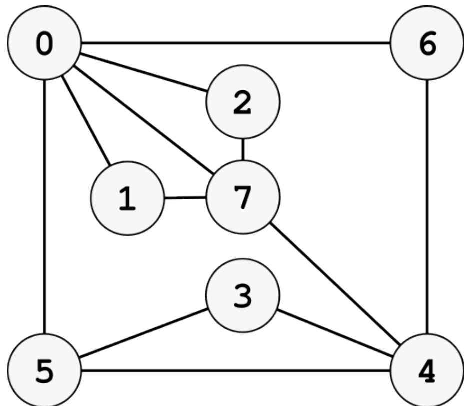


Matriz de adjacências

	0	1	2	3	4	5	6	7
0	1	1	1	0	0	1	1	1
1	1	1	0	0	0	0	0	1
2	1	0	1	0	0	0	0	1
3	0	0	0	1	1	1	0	0
4	0	0	0	1	1	1	1	1
5	1	0	0	1	1	1	0	0
6	1	0	0	0	1	0	1	0
7	1	1	1	0	1	0	0	1

Exemplo de aplicação: matriz de adjacências

- Dado matriz[i][j] = 1, diz-se que o vértice i está conectado ao vértice j
- Exemplo: hierarquia entre componentes de um jogo, ligações entre pessoas (rede sociais), mapas, etc.

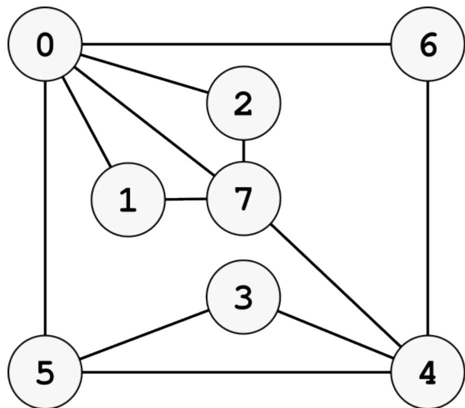


Matriz de adjacências

	0	1	2	3	4	5	6	7
0	1	1	1	0	0	1	1	1
1	1	1	0	0	0	0	0	1
2	1	0	1	0	0	0	0	1
3	0	0	0	1	1	1	0	0
4	0	0	0	1	1	1	1	1
5	1	0	0	1	1	1	0	0
6	1	0	0	0	1	0	1	0
7	1	1	1	0	1	0	0	1

Exemplo de aplicação: matriz de adjacências

- Dado matriz[i][j] = 1, diz-se que o vértice i está conectado ao vértice j
- Exemplo: hierarquia entre componentes de um jogo, ligações entre pessoas (rede sociais), mapas, etc.

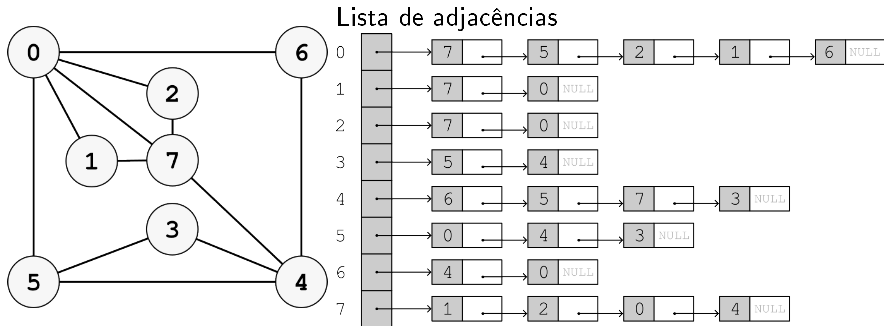


Matriz de adjacências

	0	1	2	3	4	5	6	7
0	1	1	1	0	0	1	1	1
1	1	1	0	0	0	0	0	1
2	1	0	1	0	0	0	0	1
3	0	0	0	1	1	1	0	0
4	0	0	0	1	1	1	1	1
5	1	0	0	1	1	1	0	0
6	1	0	0	0	1	0	1	0
7	1	1	1	0	1	0	0	1

Exemplo: lista de adjacências

- Quando há poucas conexões, com muitos dos pares (i,j) iguais a 0, diz-se que é uma matriz esparsa:
  - Eficientemente representada com **listas de adjacências (array de listas)**.



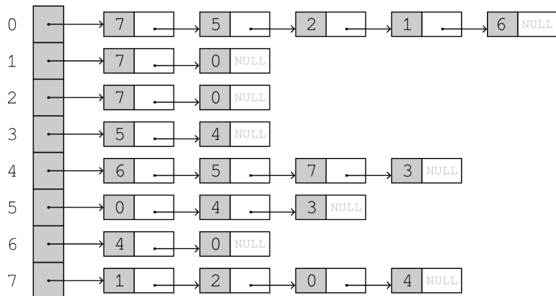
Exemplo: lista de adjacências

```c
//V número de objetos (vértices)
//A número de relacionamentos (arestas)

//Entradas: pares de vértices conectados

no *adj[V];
for(int i=0; i<V; i++){
    adj[i] = criar();
}

while(scanf("%d%d", &i, &j) == 2){
    inserir_inicio(adj[j], criar_no(i));
    inserir_inicio(adj[i], criar_no(j));
}
```

Exemplo: lista de adjacências

```
1 no *adj[V];
2 for(int i=0; i<V; i++){
3     adj[i] = criar();
4 }
5
6 while(scanf("%d%d", &i, &j) == 2){
7     inserir_inicio(adj[j], criar_no(i));
8     inserir_inicio(adj[i], criar_no(j));
9 }
```

(7,1) (2,7) (4,6) (7,0)
(0,5) (5,4) (2,0) (4,7)
(3,5) (1,0) (0,6) (3,4)



Representações das adjacências

- Matriz:
  - Acesso aleatório e direto
  - Em grafos esparsos, desperdício de espaço: V^2;
  - Processar todos os elementos: V^2
- Lista:
  - Em grafos esparsos, custo espacial é menor: $V + A$
  - Processar todos elementos: $V + A$
  - Se $A \geq V$, como acontece em muitas aplicações, o consumo de tempo é proporcional a A (linear)
  - Em grafos densos, com a maioria dos vértices conectados entre si $(V * V)$, a vantagem espacial é menor do que a desvantagem do acesso