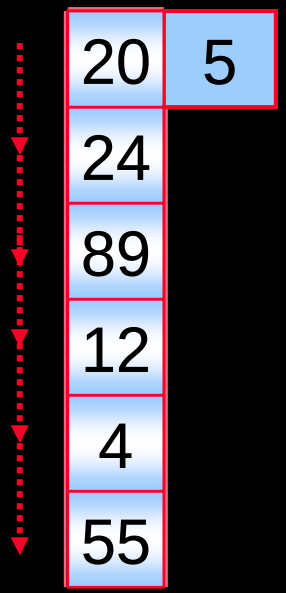
INE5408
Estruturas de Dados

Listas Encadeadas
Simples

Listas com Vetores: Desvantagens

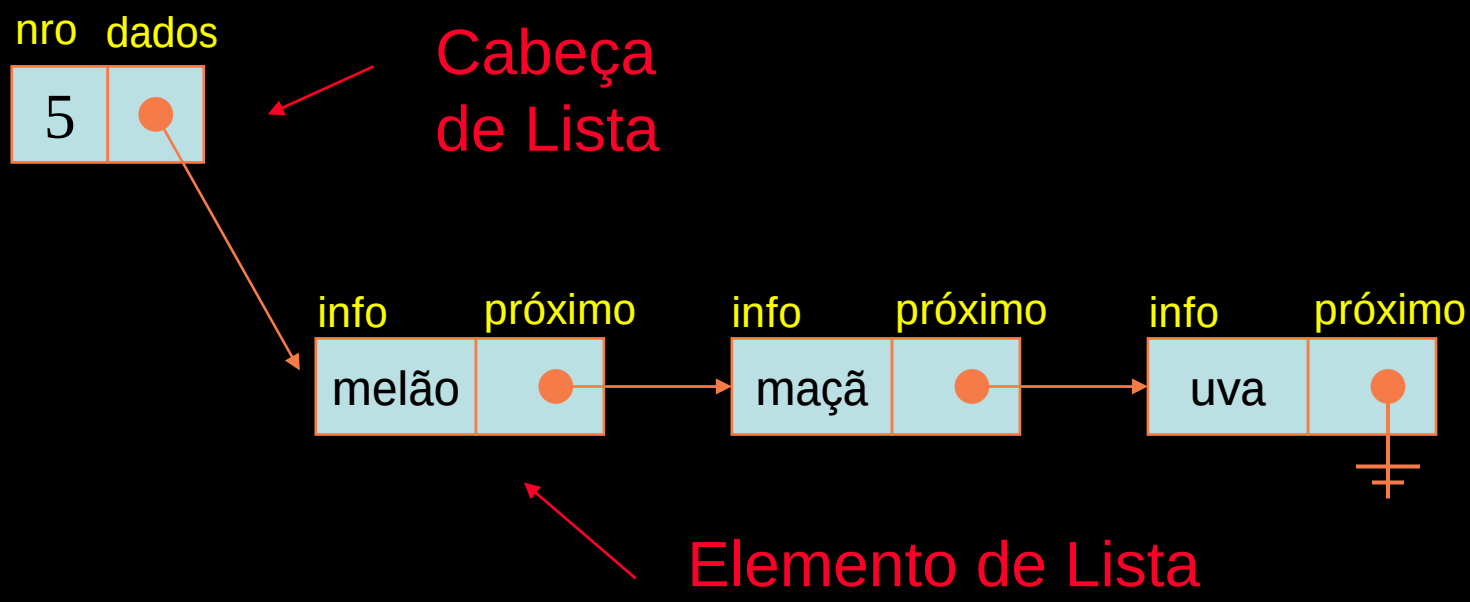


- Tamanho máximo fixo;
- mesmo vazias ocupam um grande espaço de memória:
  - mesmo que utilizemos um vetor de ponteiros, se quisermos prever uma lista de 10.000 elementos, teremos 40.000 bytes desperdiçados;
- operações podem envolver muitos deslocamentos de dados:
  - inclusão em uma posição ou no início;
  - exclusão em uma posição ou no início.

Listas Encadeadas

- São listas onde cada elemento está armazenado em um TAD chamado elemento de lista;
- cada elemento de lista referencia o próximo e só é alocado dinamicamente quando necessário;
- para referenciar o primeiro elemento utilizamos um TAD cabeça de lista.

Listas Encadeadas



Modelagem: Cabeça de Lista

- Necessitamos:
  - um ponteiro para o primeiro elemento da lista;
  - um inteiro para indicar quantos elementos a lista possui.

- Pseudo-código:

```
tipo tLista {
   tElemento *dados;
   inteiro tamanho;
 };
```

Modelagem: Elemento de Lista

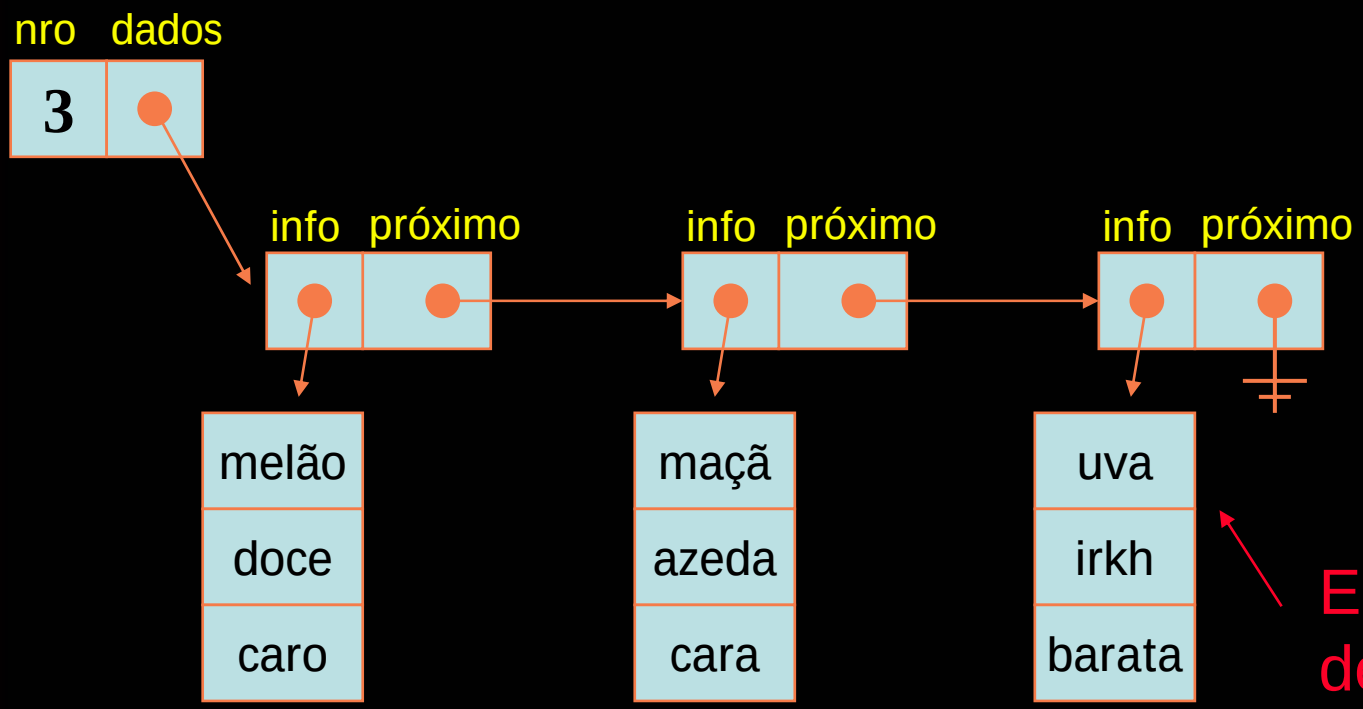
- Necessitamos:
  - um ponteiro para o próximo elemento da lista;
  - um campo do tipo da informação que vamos armazenar.

- Pseudo-código:

```
tipo tElemento {
   tElemento *próximo;
   tipo-que-eu-vou-usar-nesta-aplicação info;
 };
```

Listas Encadeadas: Modelagem

- **Para tornar todos os algoritmos da lista mais genéricos, fazemos o campo info ser um ponteiro para um elemento de informação.**



Modelagem: Elemento de Lista II

- Pseudo-código II:

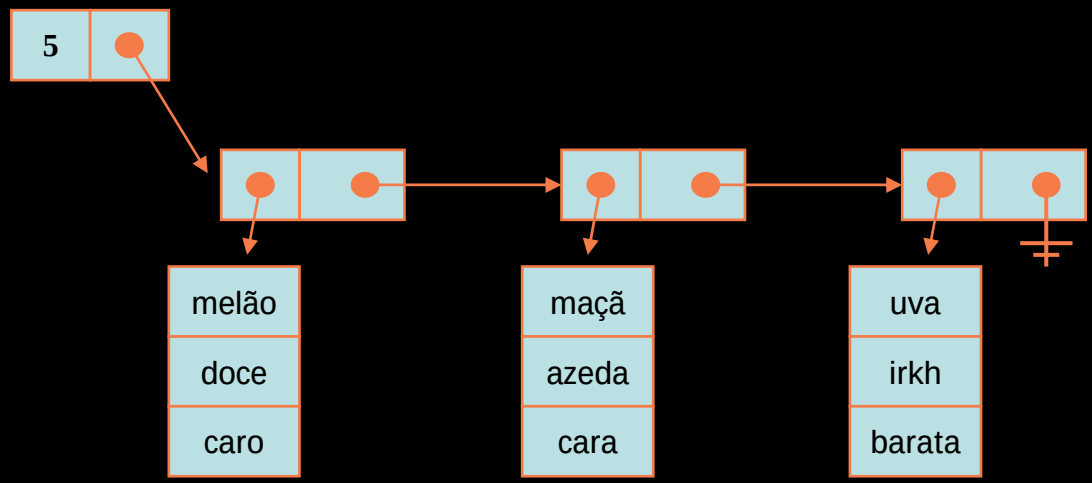
```
tipo tElemento {
  tElemento *próximo;
  TipoInfo *info;
};

tipo TipoInfo {
  tipo-do-campo1 campo1;
  tipo-do-campo2 campo2;
  …
  tipo-do-campoN campoN;
}
```

Modelagem: Elemento de Lista II

- Razões para a modelagem do TipoInfo:
  - vamos na maioria dos algoritmos trabalhar com algum elemento de infomação;
  - se este elemento é somente um ponteiro para um TipoInfo, não importando o que este seja, teremos algoritmos totalmente genéricos:
    - posso usar o mesmo código de lista para muitas aplicações diferentes simplesmente recompilando.
- Desvantagens:
  - o algoritmo de destruição da lista torna-se mais complexo.

Modelagem

- Aspecto funcional:
  - colocar e retirar dados da lista;
  - testar se a lista está vazia e outros testes;
  - inicializá-la e garantir a ordem dos elementos.



Modelagem da Lista

- Operações - colocar e retirar dados da lista:
  - Adiciona(lista, dado)
  - AdicionaNoInício(lista, dado)
  - AdicionaNaPosição(lista, dado, posição)
  - AdicionaEmOrdem(lista, dado)

  - Retira(lista)
  - RetiraDoInício(lista)
  - RetiraDaPosição(lista, posição)
  - RetiraEspecífico(lista, dado)

Modelagem da Lista

- Operações - testar a lista e outros testes:
  - ListaVazia(lista)
  - Posição(lista, dado)
  - Contém(lista, dado)
- Operações - inicializar ou limpar:
  - CriaLista()
  - DestróiLista(lista)

Algoritmo CriaLista

```
Lista* FUNÇÃO criaLista()
  //Retorna ponteiro para uma nova cabeça de lista ou NULO.
  variáveis
      Lista *aLista;
  início
      aLista <- aloque(Lista);
      SE (aLista ~= NULO) ENTÃO
            //Só posso inicializar se consegui alocar.
            aLista->tamanho <- 0;
            aLista->dados <- NULO;
      FIM SE
      RETORNE(aLista);
  fim;
```

Algoritmo CriaLista

```
Lista* FUNÇÃO criaLista()
   //Retorna ponteiro para uma nova cabeça de lista ou NULO.
   variáveis
       Lista *aLista;
   início
       aLista <- aloque(Lista);
       SE (aLista ~= NULO) ENTÃO
              //Só posso inicializar se consegui alocar.
              aLista->tamanho <- 0;
              aLista->dados <- NULO;
       FIM SE
       RETORNE(aLista);
   fim;
```

Algoritmo CriaLista

```
Lista* FUNÇÃO criaLista()
   //Retorna ponteiro para uma nova cabeça de lista ou NULO.
   variáveis
       Lista *aLista;
   início
       aLista <- aloque(Lista);
       SE (aLista ~= NULO) ENTÃO
              //Só posso inicializar se consegui alocar.
              aLista->tamanho <- 0;
              aLista->dados <- NULO;
       FIM SE
       RETORNE(aLista);
   fim;
```

Algoritmo ListaVazia

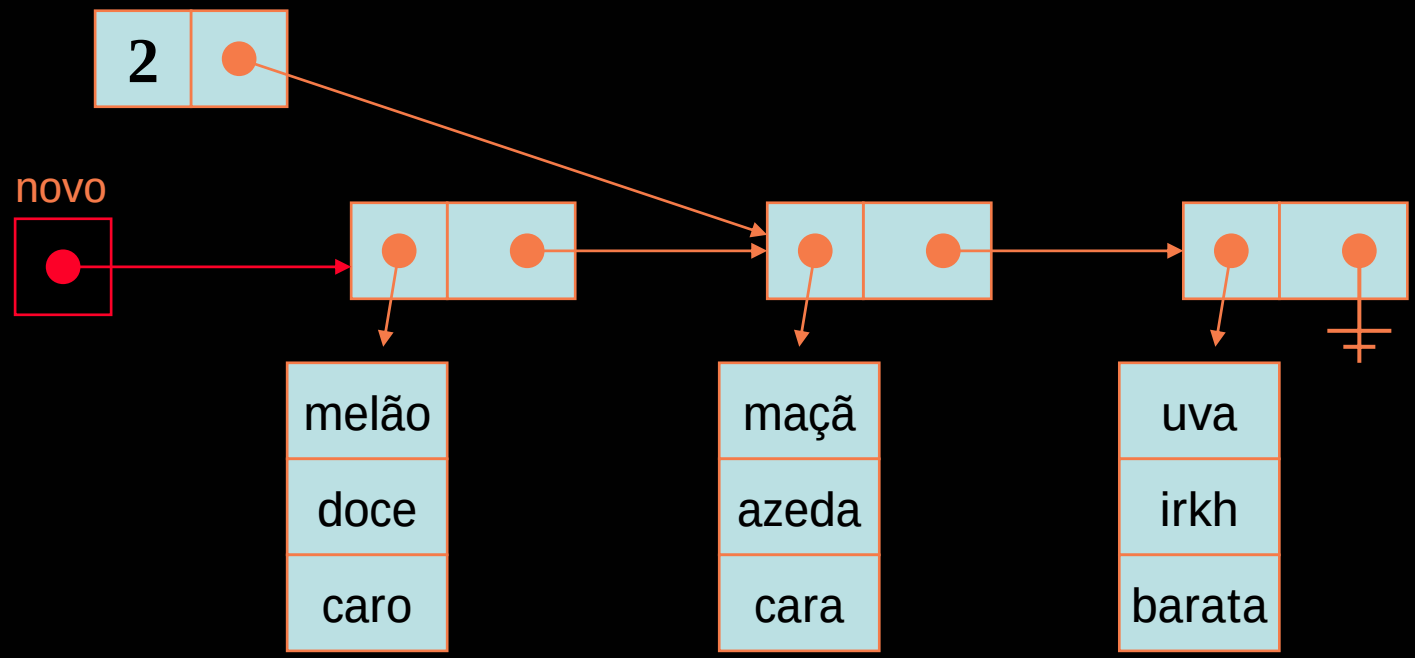
```
Booleano FUNÇÃO listaVazia(Lista *aLista)
  início
     SE (aLista->tamanho = 0) ENTÃO
          RETORNE(Verdadeiro)
     SENÃO
          RETORNE(Falso);
  fim;
```

- Um algoritmo ListaCheia não existe aqui;
- verificar se houve espaço na memória para um novo elemento será responsabilidade de cada operação de adição.

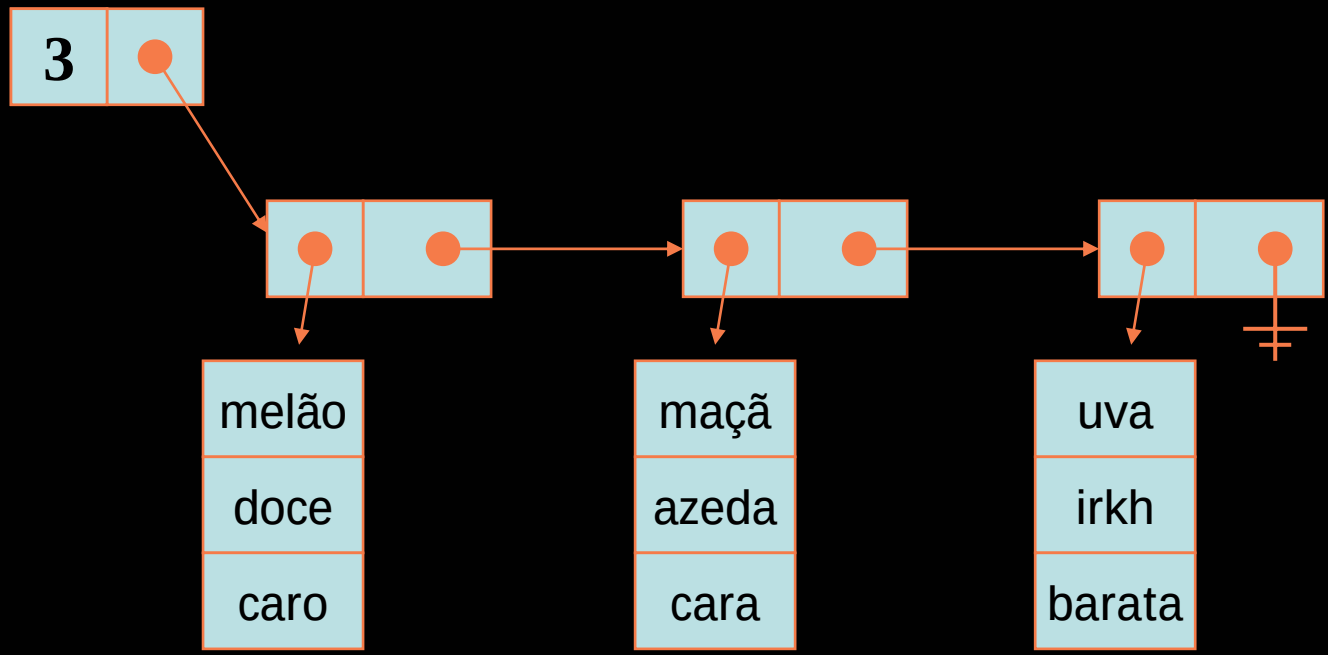
Algoritmo AdicionaNoInício

- Procedimento:
  - testamos se é possível alocar um elemento;
  - fazemos o próximo deste novo elemento ser o primeiro da lista;
  - fazemos a cabeça de lista apontar para o novo elemento.
- Parâmetros:
  - O tipo info (dado) a ser inserido;
  - a Lista.

Algoritmo AdicionaNoInício



Algoritmo AdicionaNoInício



Algoritmo AdicionaNoInício

```
Inteiro FUNÇÃO adicionaNoInício(Lista *aLista,
                                TipoInfo *dado)
   variáveis
       tElemento *novo; //Variável auxiliar.
   início
       novo <- aloque(tElemento);
       SE (novo = NULO) ENTÃO
         RETORNE(ErroListaCheia);
       SENÃO
         novo->próximo <- aLista->dados;
         novo->info <- dado;
         aLista->dados <- novo;
         aLista->tamanho <- aLista->tamanho + 1;
         RETORNE(1);
       FIM SE
   fim;
```

Algoritmo AdicionaNoInício

```
Inteiro FUNÇÃO adicionaNoInício(Lista *aLista,
                                TipoInfo *dado)
   variáveis
       tElemento *novo; //Variável auxiliar.
   início
       novo <- aloque(tElemento);
       SE (novo = NULO) ENTÃO
         RETORNE(ErroListaCheia);
       SENÃO
         novo->próximo <- aLista->dados;
         novo->info <- dado;
         aLista->dados <- novo;
         aLista->tamanho <- aLista->tamanho + 1;
         RETORNE(1);
       FIM SE
   fim;
```

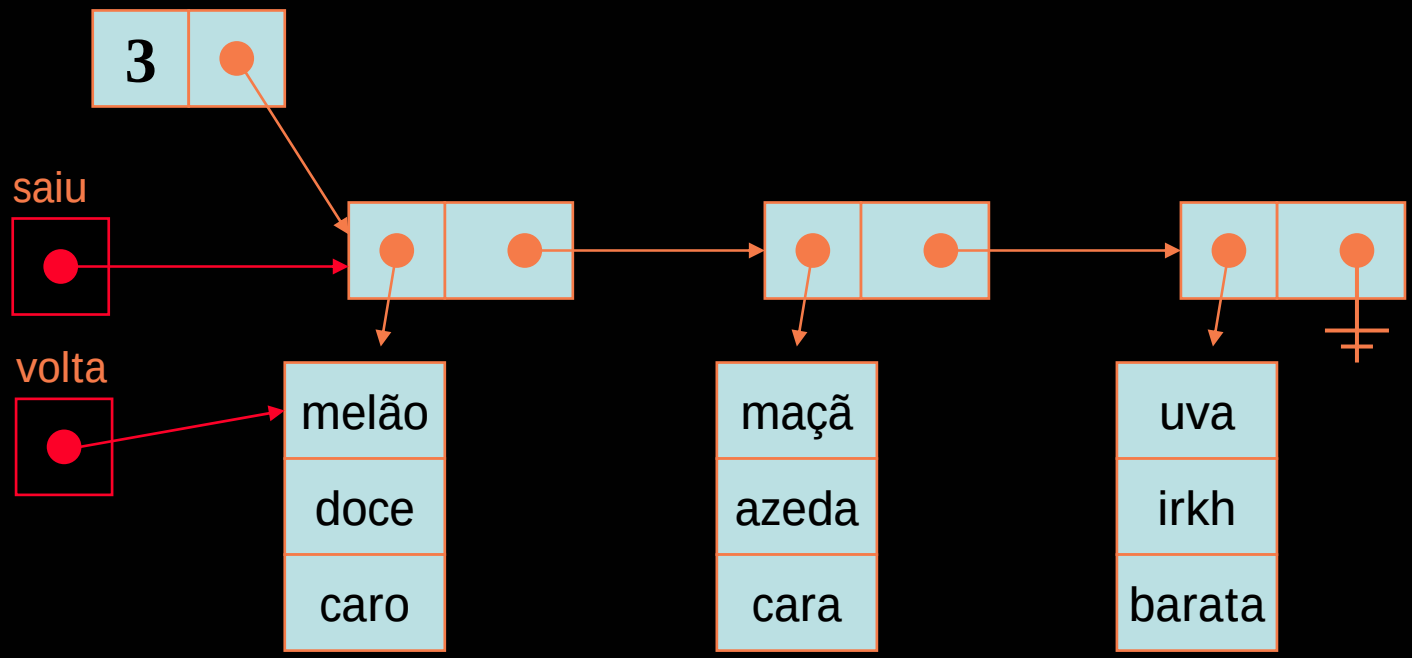
Algoritmo AdicionaNoInício

```
Inteiro FUNÇÃO adicionaNoInício(Lista *aLista,
                                TipoInfo *dado)
   variáveis
       tElemento *novo; //Variável auxiliar.
   início
       novo <- aloque(tElemento);
       SE (novo = NULO) ENTÃO
         RETORNE(ErroListaCheia);
       SENÃO
         novo->próximo <- aLista->dados;
         novo->info <- dado;
         aLista->dados <- novo;
         aLista->tamanho <- aLista->tamanho + 1;
         RETORNE(1);
       FIM SE
   fim;
```

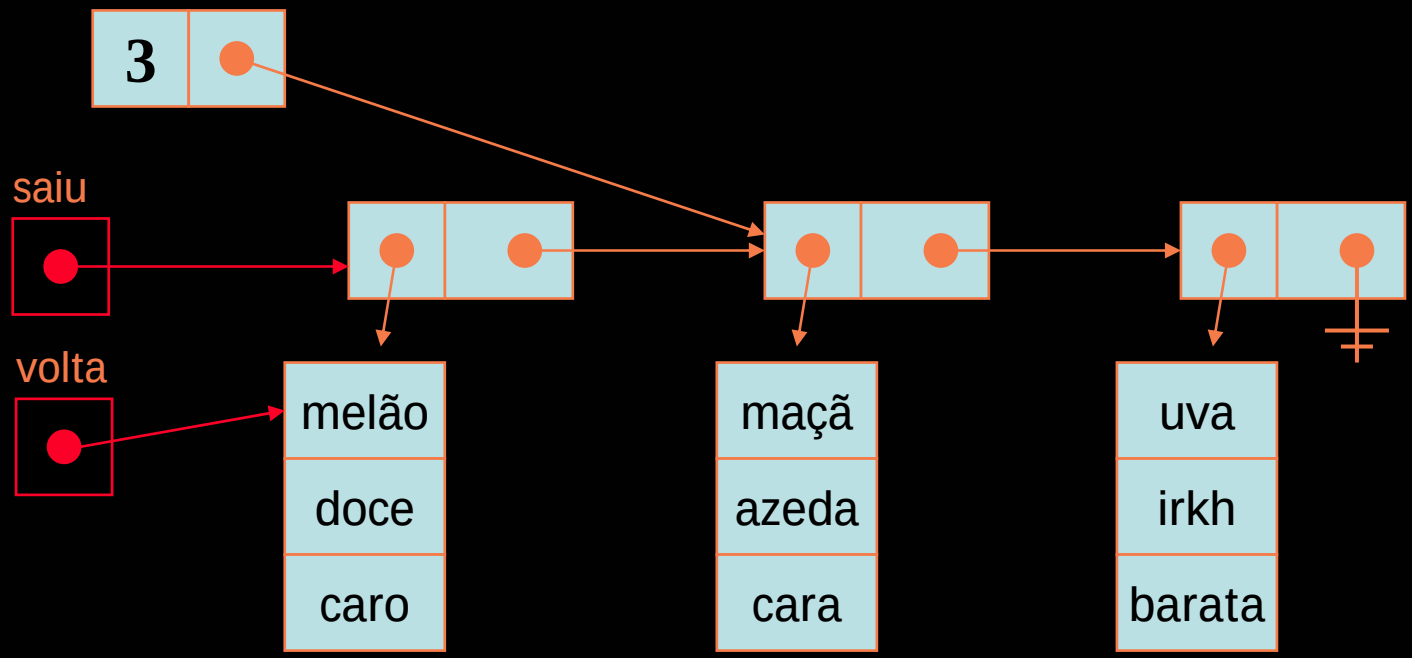
Algoritmo RetiraDoInício

- Procedimento:
  - testamos se há elementos;
  - decrementamos o tamanho;
  - liberamos a memória do elemento;
  - devolvemos a informação.
- Parâmetros:
  - a Lista.

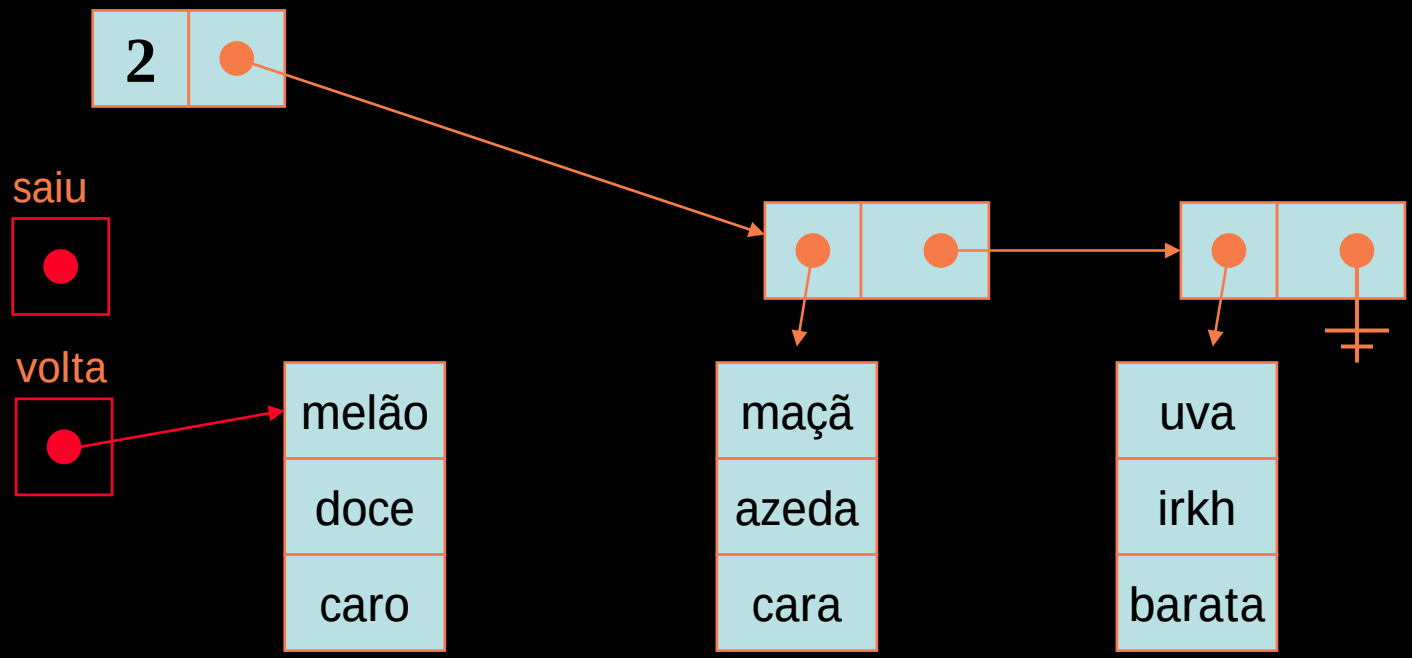
Algoritmo RetiraDoInício



Algoritmo RetiraDoInício



Algoritmo RetiraDoInício



Algoritmo RetiraDoInício

```
TipoInfo* FUNÇÃO retiraDoInício(Lista *aLista)
   //Elimina o primeiro elemento de uma lista.
   //Retorna a informação do elemento eliminado ou NULO.
   variáveis
        tElemento *saiu; //Variável auxiliar para o primeiro elemento.
        TipoInfo *volta; //Variável auxiliar para o dado retornado.
   início
        SE (listaVazia(aLista)) ENTÃO
          RETORNE(NULO);
        SENÃO
          saiu <- aLista->dados;
          volta <- saiu->info;
          aLista->dados <- saiu->próximo;
          aLista->tamanho <- aLista->tamanho - 1;
          LIBERE(saiu);
          RETORNE(volta);
        FIM SE
   fim;
```

Algoritmo EliminaDoInício

```
inteiro FUNÇÃO eliminaDoInício(Lista *aLista)
   //Elimina o primeiro elemento de uma lista e sua respectiva informação.
   //Retorna a posição do elemento eliminado ou erro.
   variáveis
        tElemento *saiu; //Variável auxiliar para o primeiro elemento.
   início
        SE (listaVazia(aLista)) ENTÃO
          RETORNE(ErroListaVazia);
        SENÃO
          saiu <- aLista->dados;
          aLista->dados <- saiu->próximo;
          aLista->tamanho <- aLista->tamanho - 1;
          LIBERE(saiu->info);
          LIBERE(saiu);
          RETORNE(aLista->tamanho + 1);
        FIM SE
   fim;
```

Algoritmo EliminaDoInício

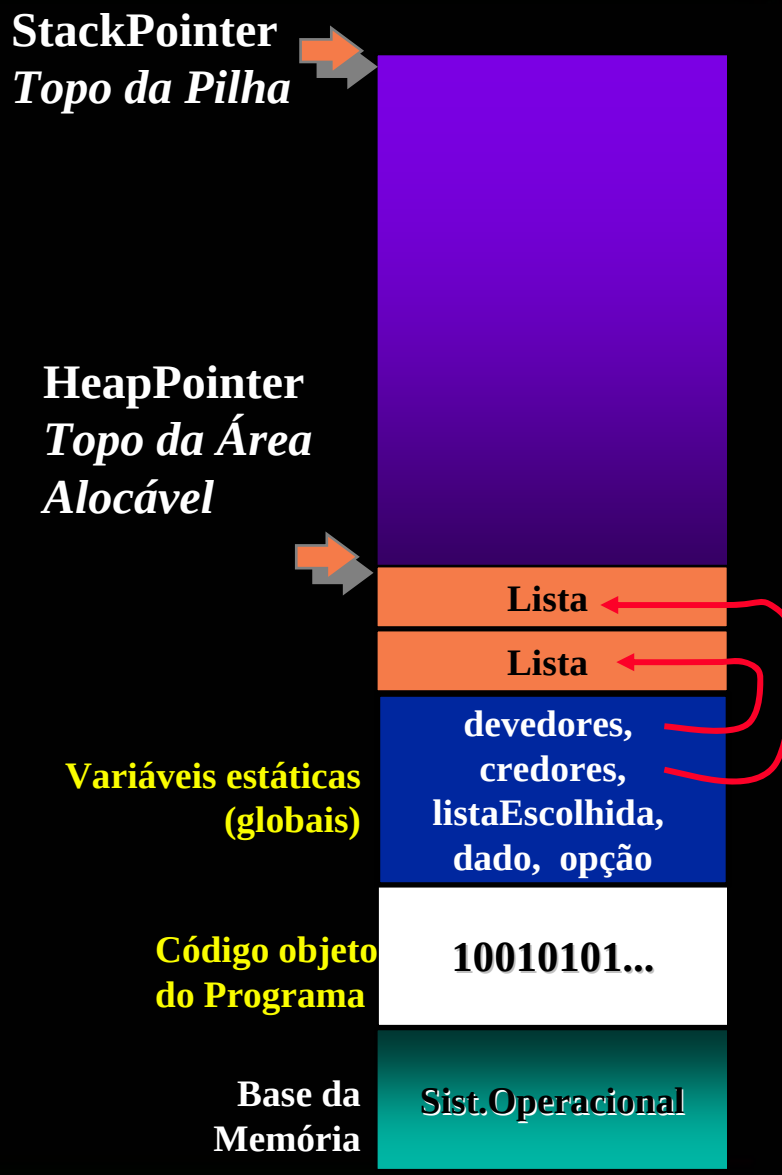
- Observe que a linha **`LIBERE(saiu->info)`** possui um perigo:
  - se o TipoInfo for por sua vez um conjunto estruturado de dados com referências internas através de ponteiros (outra lista, por exemplo), a chamada à função LIBERE(saiu->info) só liberará o primeiro nível da estrutura (aquele apontado diretamente);
  - tudo o que for referenciado através de ponteiros em info permanecerá em algum lugar da memória, provavelmente inatingível (*garbage*);
  - para evitar isto pode-se criar uma função destrói(info) para o TipoInfo que será chamada no lugar de LIBERE.

Exemplo simplificado: Programa Principal

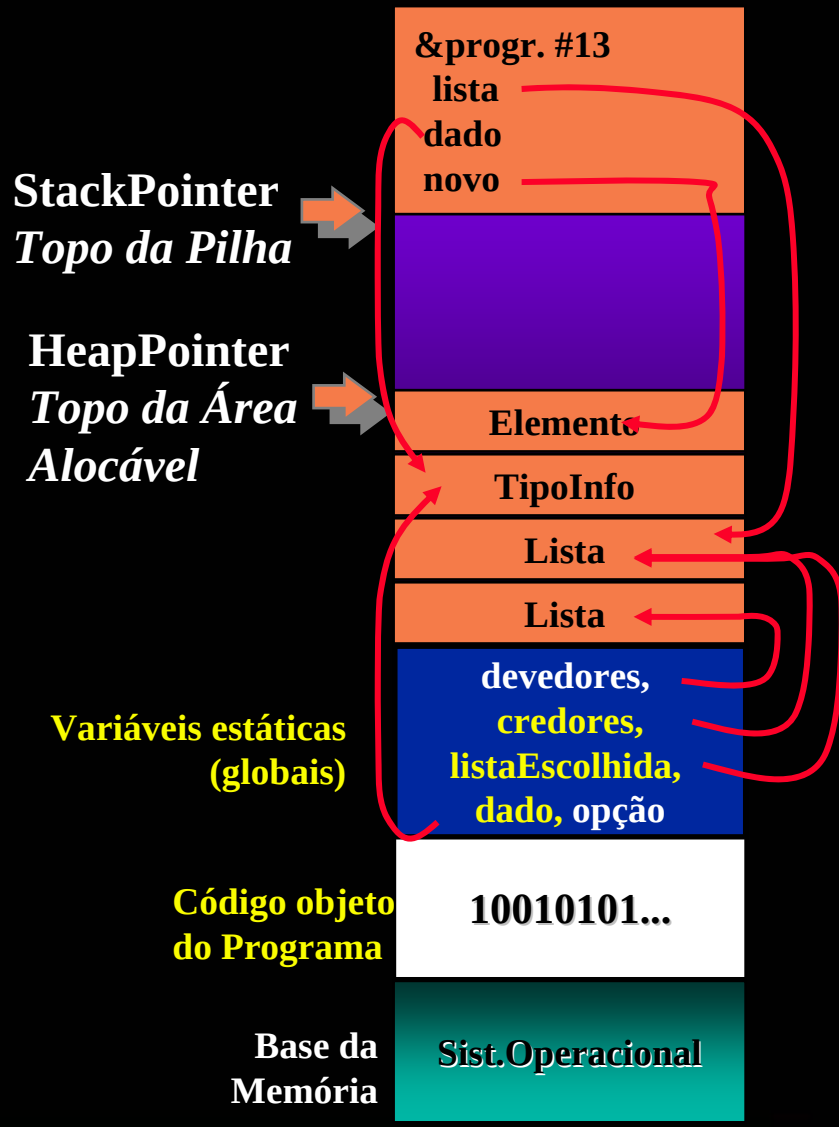
```
#inclua listaEnc.h
variáveis
    tLista *devedores, *credores, *listaEscolhida;
    TipoInfo *dado;
    caracter opção;
1 Programa Principal
2   início
3         devedores <- criaLista();
4         credores <- criaLista();
5         opção <- ´´;
6         ENQUANTO (opção ~= ´f´) ENTÃO
7                   escreveMenu();
8                   leia(opção);
9                   CASO opção SEJA
10                    ´c´: listaEscolhida <- credores;
11                    ´d´: listaEscolhida <- devedores;
12                    ´i´: dado <- leiaInfo();
13                              adicionaNoInício(listaEscolhida, dado);
--                    - - - -
--                    - - - -
--                  FIM CASO
--        FIM ENQUANTO
--  fim;
```

- Memória logo após o início do programa, quando o fluxo de execução se encontra na linha #2.

- Memória logo após as 2 chamadas à função criaLista(), quando o fluxo de execução do programa se encontra na linha #5.



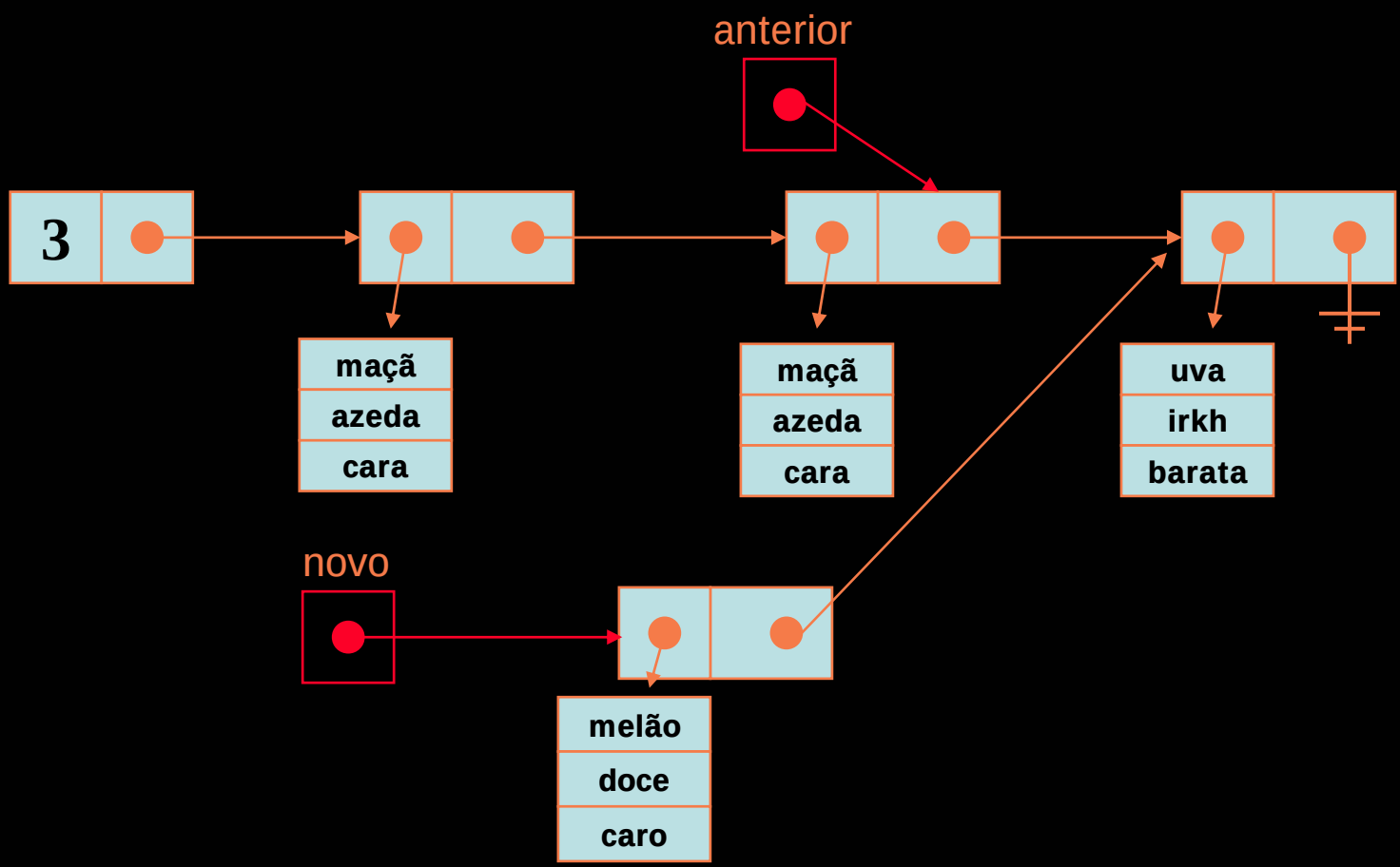
- Memória imediatamente antes de retornar de uma chamada à função adicionaNoInício(), quando a listaEscolhida é a dos credores e o fluxo de execução do programa se encontra na última linha da função adicionaNoInício() e retornará ao programa principal para a linha #13.



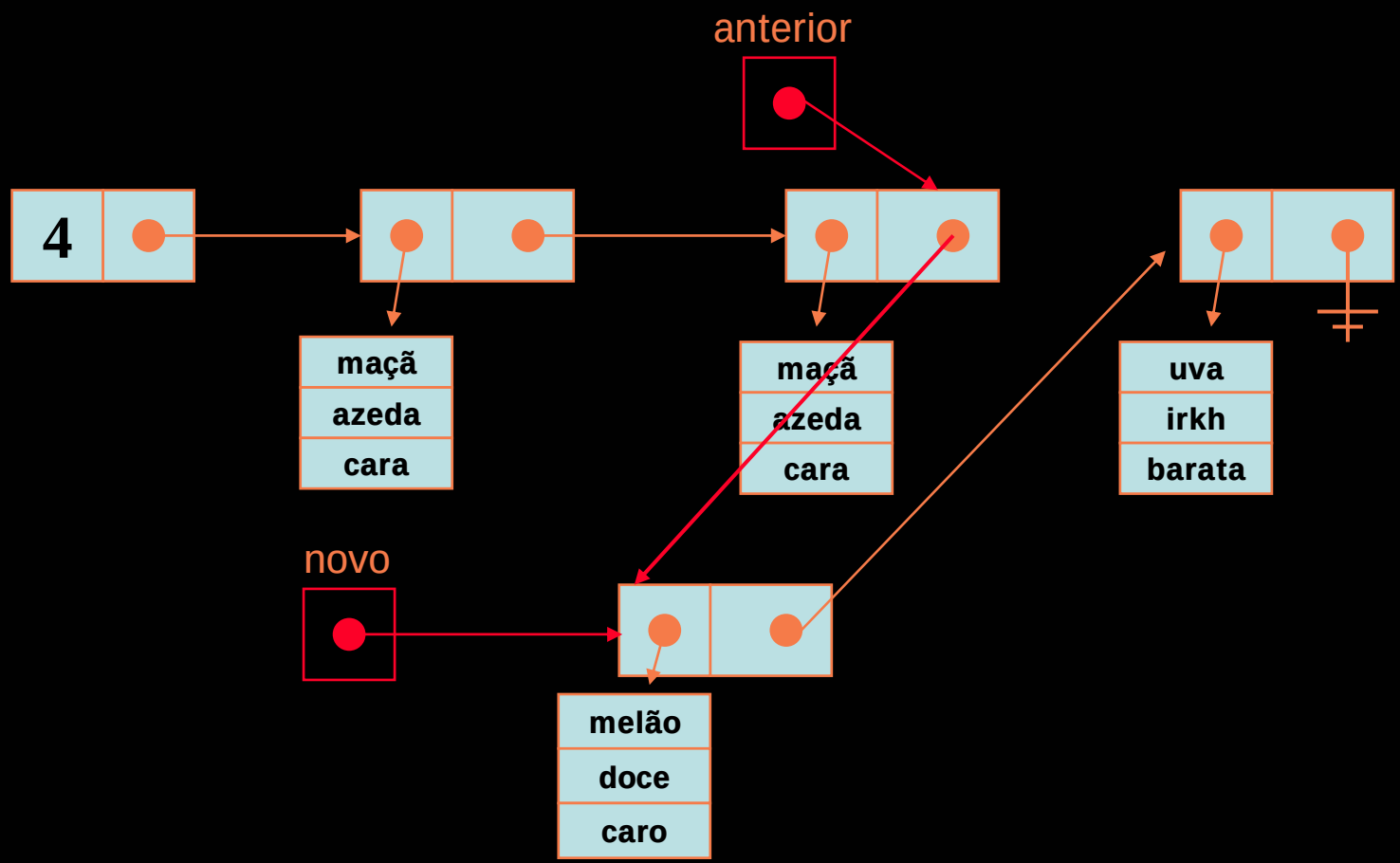
Algoritmo AdicionaNaPosição

- Procedimento:
  - testamos se a posição existe e se é possível alocar elemento;
  - caminhamos até a posição;
  - adicionamos o novo dado na posição;
  - incrementamos o tamanho.
- Parâmetros:
  - o dado a ser inserido;
  - a posição onde inserir;
  - a Lista.

Algoritmo AdicionaNaPosição



Algoritmo AdicionaNaPosição



Algoritmo AdicionaNaPosição

```
Inteiro FUNÇÃO adicionaNaPosição(tLista *aLista, TipoInfo *info,
                                 inteiro posição)
    //Adiciona novo elemento na posição informada.
    //Retorna o novo número de elementos da lista ou erro.
    variáveis
      tElemento *novo, *anterior; //Ponteiros auxiliares.
    início
      SE (posição > aLista->tamanho + 1) ENTÃO
          RETORNE(ErroPosição)
      SENÃO
          SE (posição = 1) ENTÃO
            RETORNE(adicionaNoInício(aLista, info);
          SENÃO
            novo <- aloque(tElemento);
            SE (novo = NULO) ENTÃO
              RETORNE(ErroListaCheia);
            SENÃO
              anterior <- aLista->dados;
              REPITA (posição - 2) VEZES
                    anterior <- anterior->próximo;
              novo->próximo <- anterior->próximo;
              novo->info <- info;
              anterior->próximo <- novo;
              aLista->tamanho <- aLista->tamanho + 1;
              RETORNE(aLista->tamanho);
            FIM SE
        FIM SE
      FIM SE
    fim;
```

Algoritmo AdicionaNaPosição

```
Inteiro FUNÇÃO adicionaNaPosição(tLista *aLista, TipoInfo *info,
                                 inteiro posição)
    //Adiciona novo elemento na posição informada.
    //Retorna o novo número de elementos da lista ou erro.
    variáveis
      tElemento *novo, *anterior; //Ponteiros auxiliares.
    início
      SE (posição > aLista->tamanho + 1) ENTÃO
          RETORNE(ErroPosição)
      SENÃO
          SE (posição = 1) ENTÃO
            RETORNE(adicionaNoInício(aLista, info);
          SENÃO
            novo <- aloque(tElemento);
            SE (novo = NULO) ENTÃO
              RETORNE(ErroListaCheia);
            SENÃO
              anterior <- aLista->dados;
              REPITA (posição - 2) VEZES
                    anterior <- anterior->próximo;
              novo->próximo <- anterior->próximo;
              novo->info <- info;
              anterior->próximo <- novo;
              aLista->tamanho <- aLista->tamanho + 1;
              RETORNE(aLista->tamanho);
            FIM SE
        FIM SE
      FIM SE
    fim;
```

Algoritmo AdicionaNaPosição

```
Inteiro FUNÇÃO adicionaNaPosição(tLista *aLista, TipoInfo *info,
                                 inteiro posição)
    //Adiciona novo elemento na posição informada.
    //Retorna o novo número de elementos da lista ou erro.
    variáveis
      tElemento *novo, *anterior; //Ponteiros auxiliares.
    início
      SE (posição > aLista->tamanho + 1) ENTÃO
          RETORNE(ErroPosição)
      SENÃO
          SE (posição = 1) ENTÃO
            RETORNE(adicionaNoInício(aLista, info);
          SENÃO
            novo <- aloque(tElemento);
            SE (novo = NULO) ENTÃO
              RETORNE(ErroListaCheia);
            SENÃO
              anterior <- aLista->dados;
              REPITA (posição - 2) VEZES
                    anterior <- anterior->próximo;
              novo->próximo <- anterior->próximo;
              novo->info <- info;
              anterior->próximo <- novo;
              aLista->tamanho <- aLista->tamanho + 1;
              RETORNE(aLista->tamanho);
            FIM SE
        FIM SE
      FIM SE
    fim;
```

Algoritmo AdicionaNaPosição

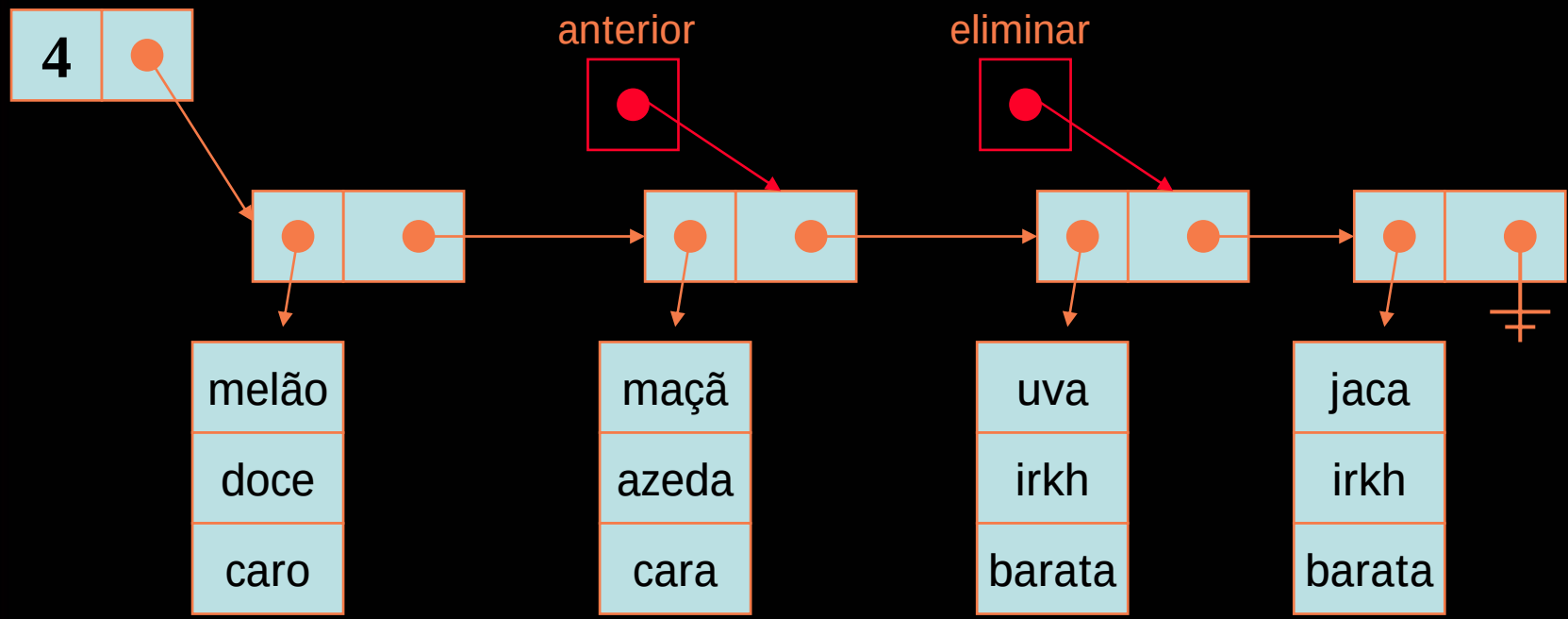
```
Inteiro FUNÇÃO adicionaNaPosição(tLista *aLista, TipoInfo *info,
                                 inteiro posição)
    //Adiciona novo elemento na posição informada.
    //Retorna o novo número de elementos da lista ou erro.
    variáveis
      tElemento *novo, *anterior; //Ponteiros auxiliares.
    início
      SE (posição > aLista->tamanho + 1) ENTÃO
          RETORNE(ErroPosição)
      SENÃO
          SE (posição = 1) ENTÃO
            RETORNE(adicionaNoInício(aLista, info);
          SENÃO
            novo <- aloque(tElemento);
            SE (novo = NULO) ENTÃO
              RETORNE(ErroListaCheia);
            SENÃO
              anterior <- aLista->dados;
              REPITA (posição - 2) VEZES
                    anterior <- anterior->próximo;
              novo->próximo <- anterior->próximo;
              novo->info <- info;
              anterior->próximo <- novo;
              aLista->tamanho <- aLista->tamanho + 1;
              RETORNE(aLista->tamanho);
            FIM SE
        FIM SE
      FIM SE
    fim;
```

Algoritmo RetiraDaPosição

- Procedimento:
  - testamos se a posição existe;
  - caminhamos até a posição;
  - retiramos o dado da posição;
  - decrementamos o tamanho.
- Parâmetros:
  - a posição de onde retirar;
  - a Lista.

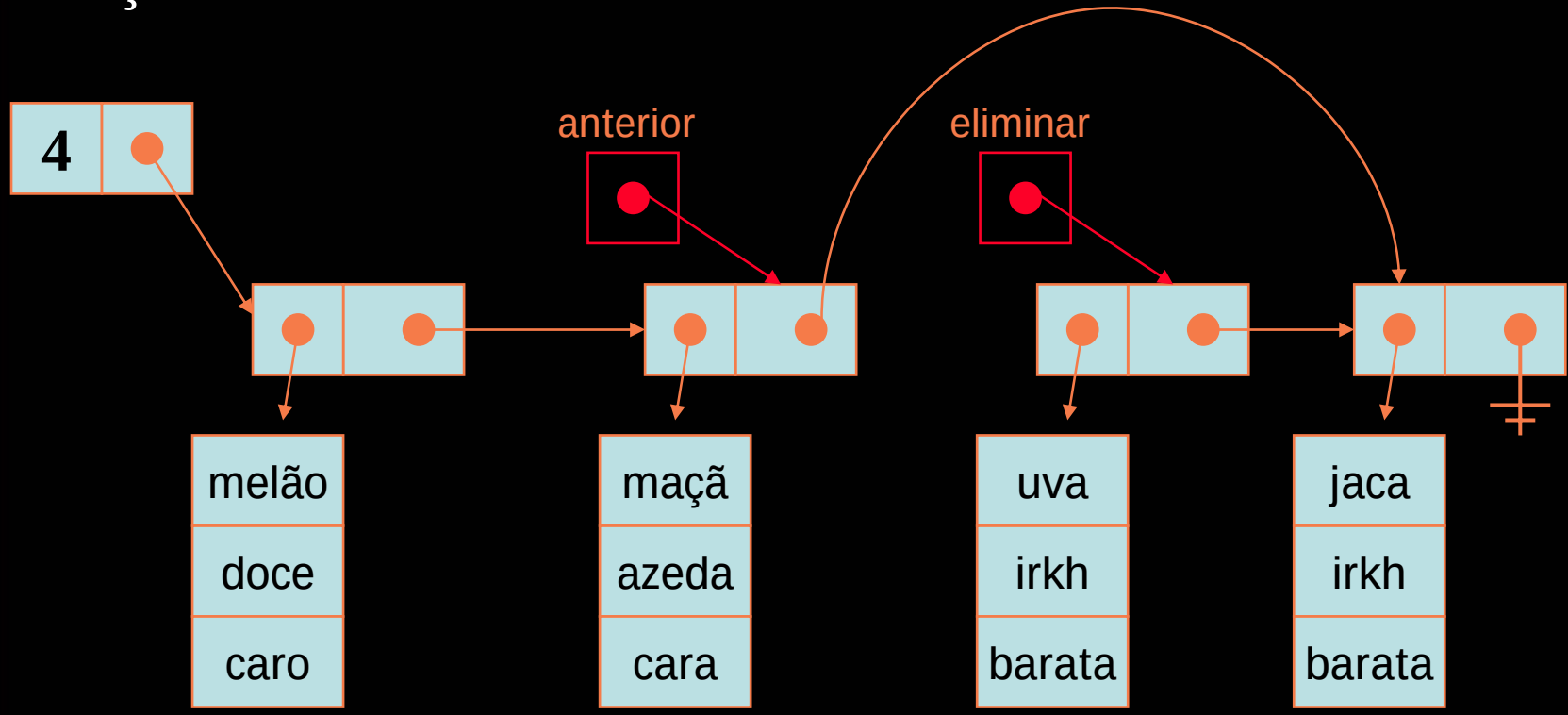
Algoritmo RetiraDaPosição

Posições > 1



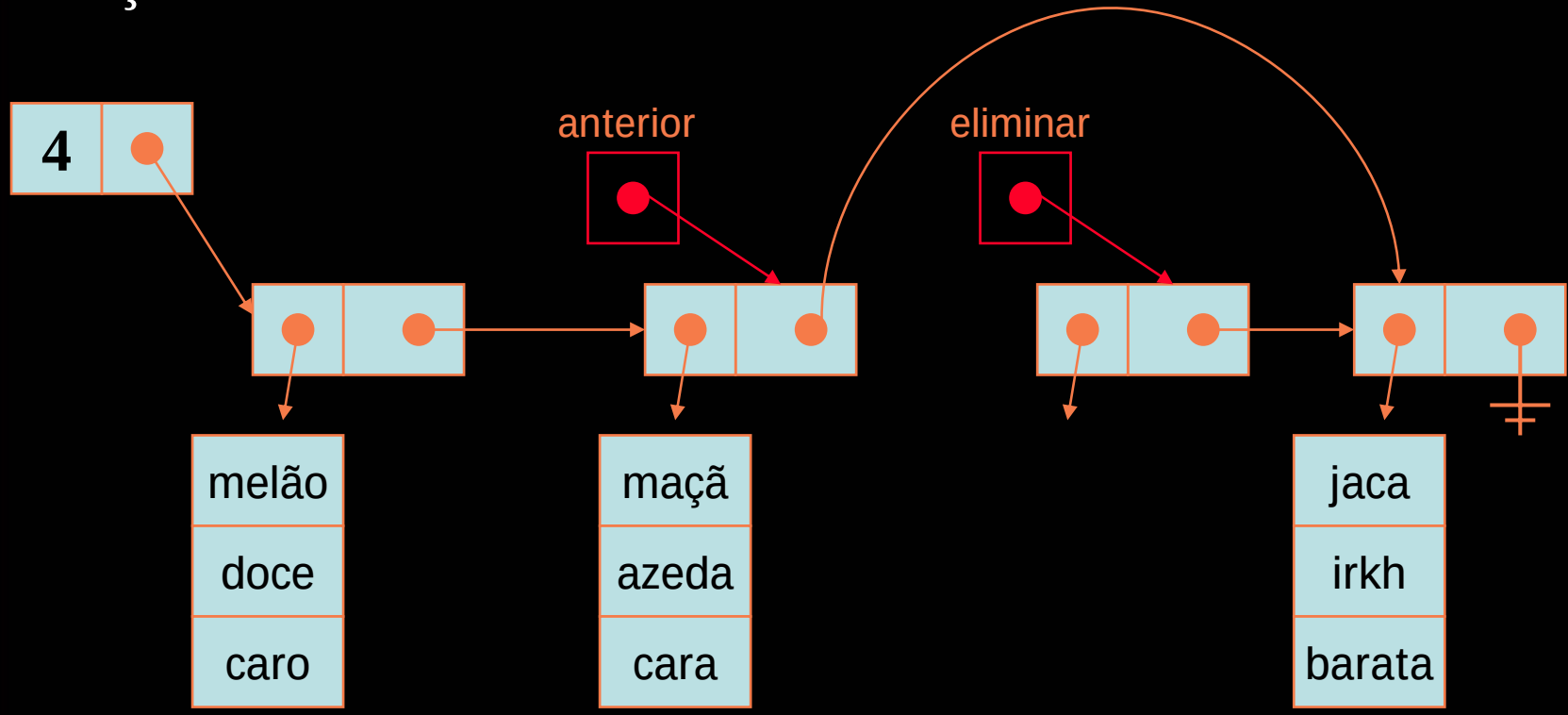
Algoritmo RetiraDaPosição

Posições > 1



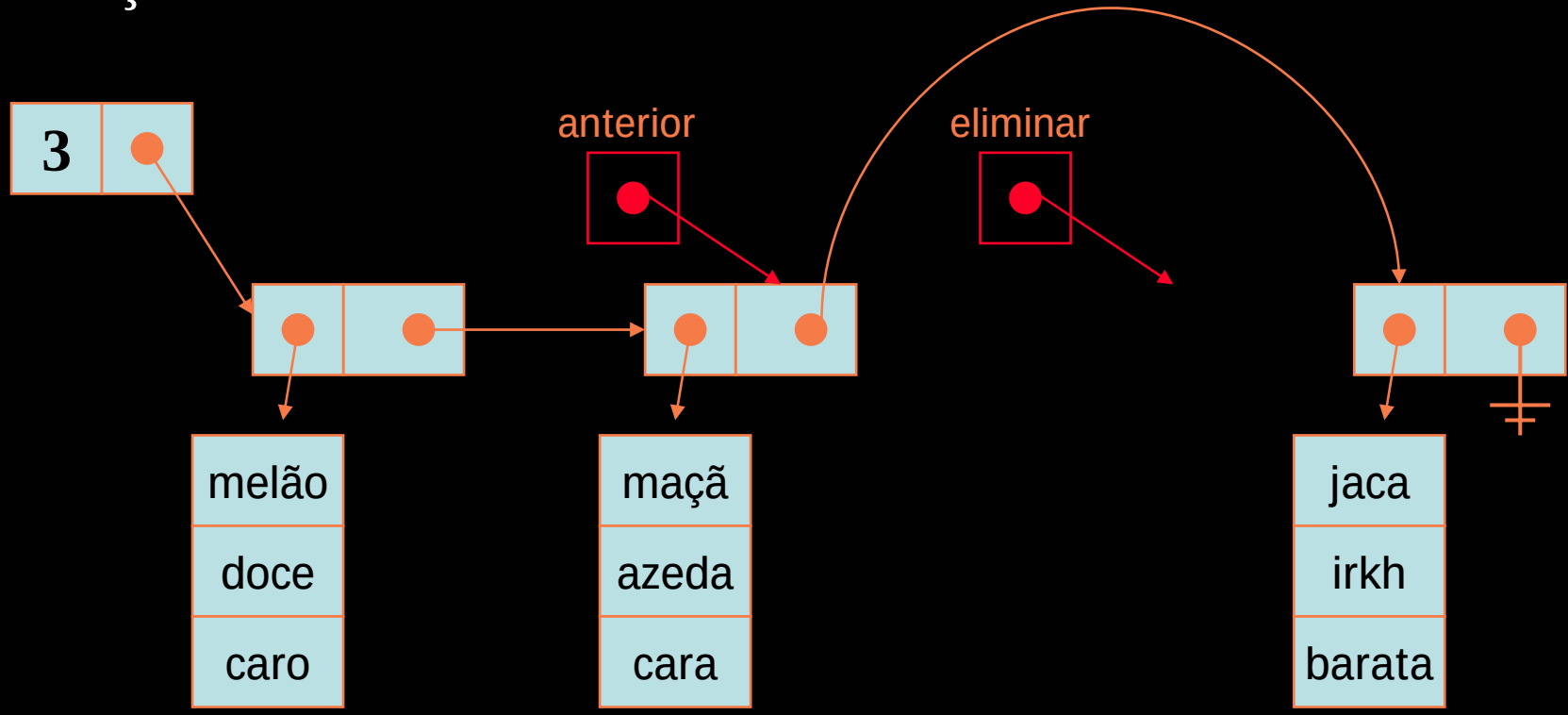
Algoritmo RetiraDaPosição

Posições > 1



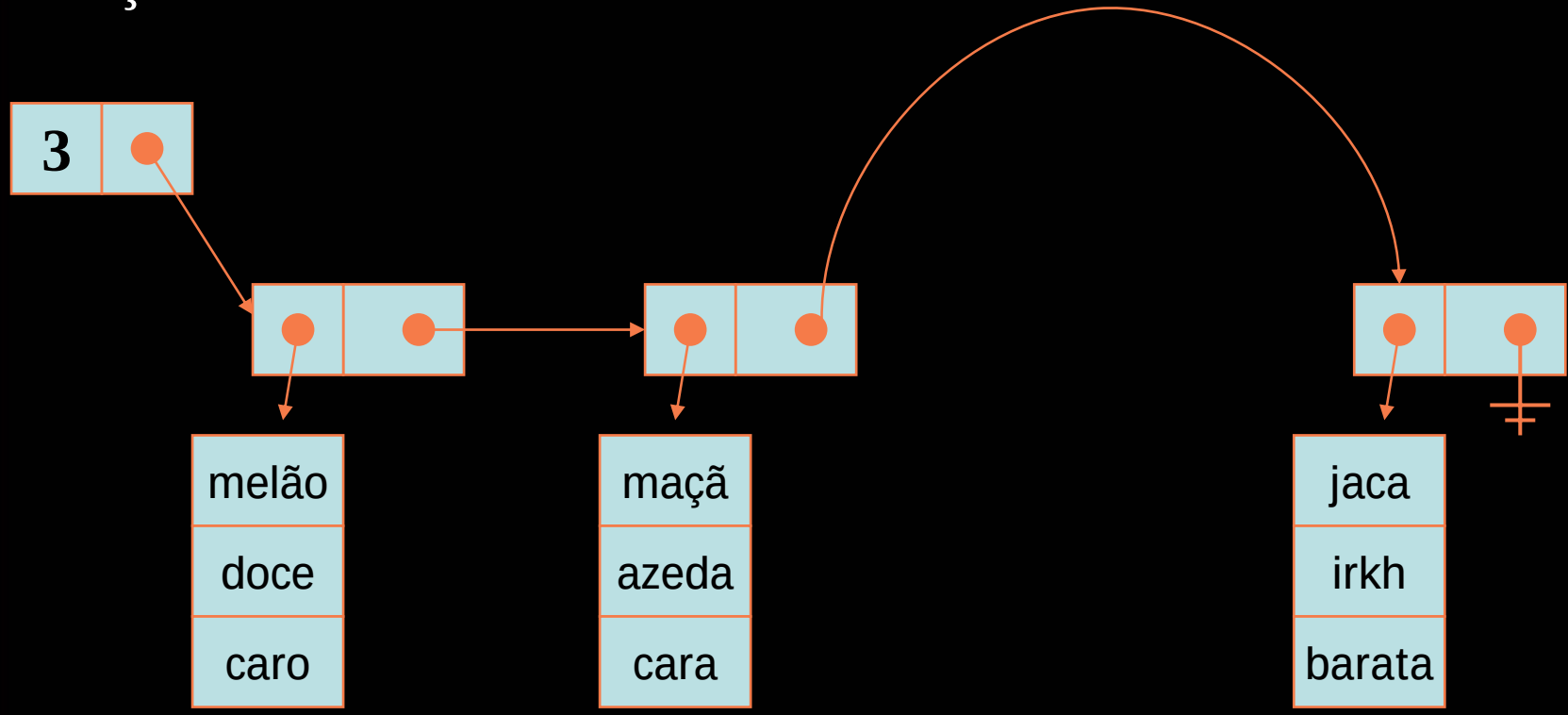
Algoritmo RetiraDaPosição

Posições > 1



Algoritmo RetiraDaPosição

Posições > 1



Algoritmo RetiraDaPosição

```
TipoInfo* FUNÇÃO retiraDaPosição(tLista *aLista, inteiro posição)
    //Elimina o elemento da posição de uma lista.
    //Retorna a informação do elemento eliminado ou NULO.
    variáveis
          tElemento *anterior, *eliminar; //Variável auxiliar para elemento.
          TipoInfo *volta; //Variável auxiliar para o dado retornado.
    início
      SE (posição > aLista->tamanho) ENTÃO
          RETORNE(NULO);
      SENÃO
          SE (posição = 1) ENTÃO
            RETORNE(retiraDoInício(aLista));
          SENÃO
            anterior <- aLista->dados;
            REPITA (posição - 2) VEZES
                   anterior <- anterior->próximo;
            eliminar <- anterior->próximo;
            volta <- eliminar->info;
            anterior->próximo <- eliminar->próximo;
            aLista->tamanho <- aLista->tamanho - 1;
            LIBERE(eliminar);
            RETORNE(volta);
          FIM SE
      FIM SE
    fim;
```

Algoritmo RetiraDaPosição

```
TipoInfo* FUNÇÃO retiraDaPosição(tLista *aLista, inteiro posição)
    //Elimina o elemento da posição de uma lista.
    //Retorna a informação do elemento eliminado ou NULO.
    variáveis
          tElemento *anterior, *eliminar; //Variável auxiliar para elemento.
          TipoInfo *volta; //Variável auxiliar para o dado retornado.
    início
      SE (posição > aLista->tamanho) ENTÃO
          RETORNE(NULO);
      SENÃO
          SE (posição = 1) ENTÃO
            RETORNE(retiraDoInício(aLista));
          SENÃO
            anterior <- aLista->dados;
            REPITA (posição - 2) VEZES
                    anterior <- anterior->próximo;
            eliminar <- anterior->próximo;
            volta <- eliminar->info;
            anterior->próximo <- eliminar->próximo;
            aLista->tamanho <- aLista->tamanho - 1;
            LIBERE(eliminar);
            RETORNE(volta);
          FIM SE
      FIM SE
    fim;
```

Algoritmo RetiraDaPosição

```
TipoInfo* FUNÇÃO retiraDaPosição(tLista *aLista, inteiro posição)
    //Elimina o elemento da posição de uma lista.
    //Retorna a informação do elemento eliminado ou NULO.
    variáveis
          tElemento *anterior, *eliminar; //Variável auxiliar para elemento.
          TipoInfo *volta; //Variável auxiliar para o dado retornado.
    início
      SE (posição > aLista->tamanho) ENTÃO
          RETORNE(NULO);
      SENÃO
          SE (posição = 1) ENTÃO
            RETORNE(retiraDoInício(aLista));
          SENÃO
            anterior <- aLista->dados;
            REPITA (posição - 2) VEZES
                   anterior <- anterior->próximo;
            eliminar <- anterior->próximo;
            volta <- eliminar->info;
            anterior->próximo <- eliminar->próximo;
            aLista->tamanho <- aLista->tamanho - 1;
            LIBERE(eliminar);
            RETORNE(volta);
          FIM SE
      FIM SE
    fim;
```

Algoritmo RetiraDaPosição

```
TipoInfo* FUNÇÃO retiraDaPosição(tLista *aLista, inteiro posição)
    //Elimina o elemento da posição de uma lista.
    //Retorna a informação do elemento eliminado ou NULO.
    variáveis
          tElemento *anterior, *eliminar; //Variável auxiliar para elemento.
          TipoInfo *volta; //Variável auxiliar para o dado retornado.
    início
      SE (posição > aLista->tamanho) ENTÃO
          RETORNE(NULO);
      SENÃO
          SE (posição = 1) ENTÃO
            RETORNE(retiraDoInício(aLista));
          SENÃO
            anterior <- aLista->dados;
            REPITA (posição - 2) VEZES
                    anterior <- anterior->próximo;
            eliminar <- anterior->próximo;
            volta <- eliminar->info;
            anterior->próximo <- eliminar->próximo;
            aLista->tamanho <- aLista->tamanho - 1;
            LIBERE(eliminar);
            RETORNE(volta);
          FIM SE
      FIM SE
    fim;
```

Algoritmo RetiraDaPosição

```
TipoInfo* FUNÇÃO retiraDaPosição(tLista *aLista, inteiro posição)
    //Elimina o elemento da posição de uma lista.
    //Retorna a informação do elemento eliminado ou NULO.
    variáveis
          tElemento *anterior, *eliminar; //Variável auxiliar para elemento.
          TipoInfo *volta; //Variável auxiliar para o dado retornado.
    início
      SE (posição > aLista->tamanho) ENTÃO
          RETORNE(NULO);
      SENÃO
          SE (posição = 1) ENTÃO
            RETORNE(retiraDoInício(aLista));
          SENÃO
            anterior <- aLista->dados;
            REPITA (posição - 2) VEZES
                   anterior <- anterior->próximo;
            eliminar <- anterior->próximo;
            volta <- eliminar->info;
            anterior->próximo <- eliminar->próximo;
            aLista->tamanho <- aLista->tamanho - 1;
            LIBERE(eliminar);
            RETORNE(volta);
          FIM SE
      FIM SE
    fim;
```

Modelagem do Tipo Info

- Para inserção em ordem e para achar um elemento determinado, necessitamos da capacidade de comparar informações associadas aos elementos;
  - estas operações de comparação fazem parte do TAD TipoInfo e não da lista;
  - devem ser implementadas como tal.
- Operações: testar AS INFORMAÇÕES:
  - Igual(dado1, dado2)
  - Maior(dado1, dado2)
  - Menor(dado1, dado2)

Algoritmo AdicionaEmOrdem

- Procedimento:
  - necessitamos de uma função para comparar os dados (maior);
  - procuramos pela posição onde inserir comparando dados;
  - chamamos **`adicionaNaPosição()`**.
- Parâmetros:
  - o dado a ser inserido;
  - a Lista.

Algoritmo AdicionaEmOrdem

```
Inteiro FUNÇÃO adicionaEmOrdem(tLista *aLista, TipoInfo dado)
   variáveis
     tElemento *atual; //Variável auxiliar para caminhar.
     inteiro posição;
   início
     SE (listaVazia(aLista)) ENTÃO
       RETORNE(adicionaNoInício(aLista, dado));
     SENÃO
       atual <- aLista->dados;
       posição <- 1;
       ENQUANTO (atual->próximo ~= NULO E
                 maior(dado, atual->info)) FAÇA
          //Encontrar posição para inserir.
          atual <- atual->próximo;
          posição <- posição + 1;
        FIM ENQUANTO
        SE maior(dado, atual->info) ENTÃO //Parou porque acabou a lista.
          RETORNE(adicionaNaPosição(aLista, dado, posição + 1));
        SENÃO
          RETORNE(adicionaNaPosição(aLista, dado, posição));
        FIM SE
     FIM SE
   fim;
```

Algoritmos Restantes

- Por conta do aluno:
  - Adiciona(lista, dado)
  - Retira(lista)
  - RetiraEspecífico(lista, dado)
- Operações - inicializar ou limpar:
  - DestróiLista(lista)

Algoritmo DestróiLista

```
FUNÇÃO destróiLista(tLista *aLista)
   variáveis
     tElemento *atual, *anterior; //Variável auxiliar para caminhar.
   início
     SE (listaVazia(aLista)) ENTÃO
        LIBERE(aLista);
     SENÃO
        atual <- aLista->dados;
        ENQUANTO (atual ~= NULO) FAÇA
          //Eliminar até o fim.
          anterior <- atual;
          //Vou para o próximo mesmo que seja nulo.
          atual <- atual->próximo;
          //Liberar primeiro a Info.
          LIBERE(anterior->info);
          //Liberar o elemento que acabei de visitar.
          LIBERE(anterior);
        FIM ENQUANTO
        LIBERE(aLista);
     FIM SE
   fim;
```