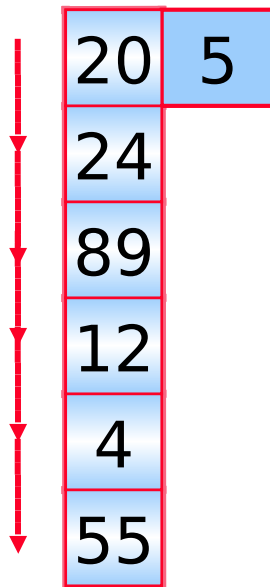


Estruturas de Dados

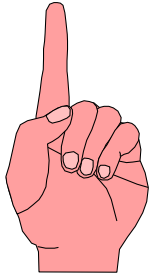
# Listas Encadeadas Simples

# Listas com Vetores: Desvantagens



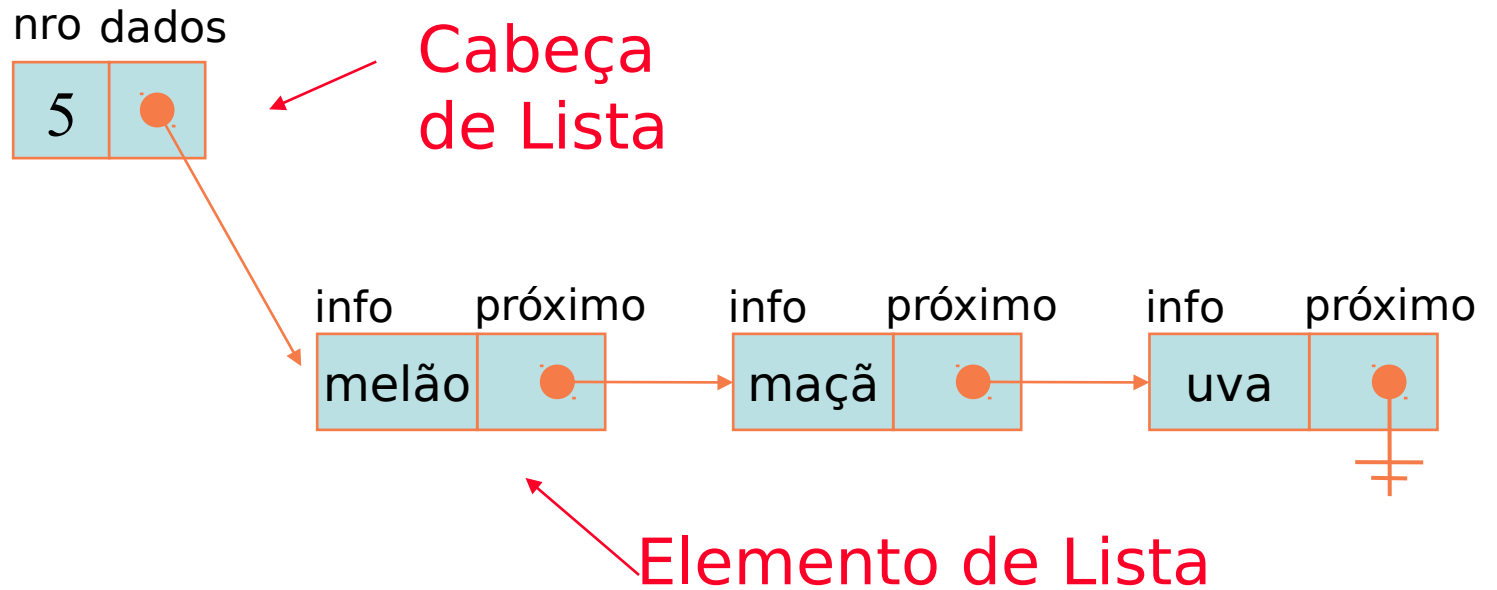
- Tamanho máximo fixo;
- mesmo vazias ocupam um grande espaço de memória:
  - mesmo que utilizemos um vetor de ponteiros, se quisermos prever uma lista de 10.000 elementos, teremos 40.000 bytes desperdiçados;
- operações podem envolver muitos deslocamentos de dados:
  - inclusão em uma posição ou no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

# Listas Encadeadas



- São listas onde cada elemento está armazenado em um objeto chamada elemento de lista;
- cada elemento de lista referencia o próximo e só é alocado dinamicamente quando necessário;
- para referenciar o primeiro elemento utilizamos um objeto 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:

```
classe 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:

```
class tElemento {  
    tElemento *próximo;  
    T info;  
};
```

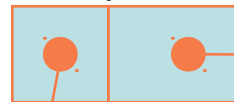
# Listas Encadeadas: Modelagem

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

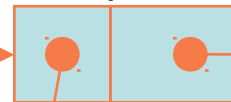
nro dados



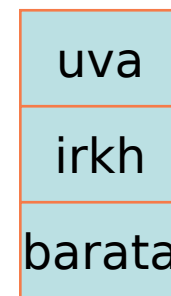
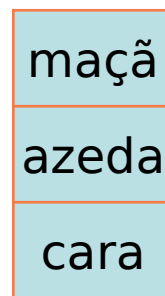
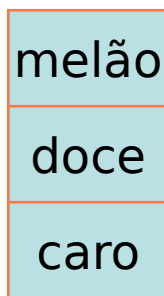
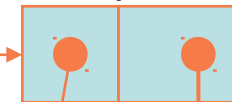
info próximo



info próximo



info próximo




Elemento de Informação (tipoInfo)

# Modelagem: Elemento de Lista

- Pseudo-código da Classe elemento de lista:

```
class tElemento {  
    tElemento *próximo;  
    T *info;  
};
```

Ao invés de colocar a Informação no elemento de lista, usamos um ponteiro para ela.

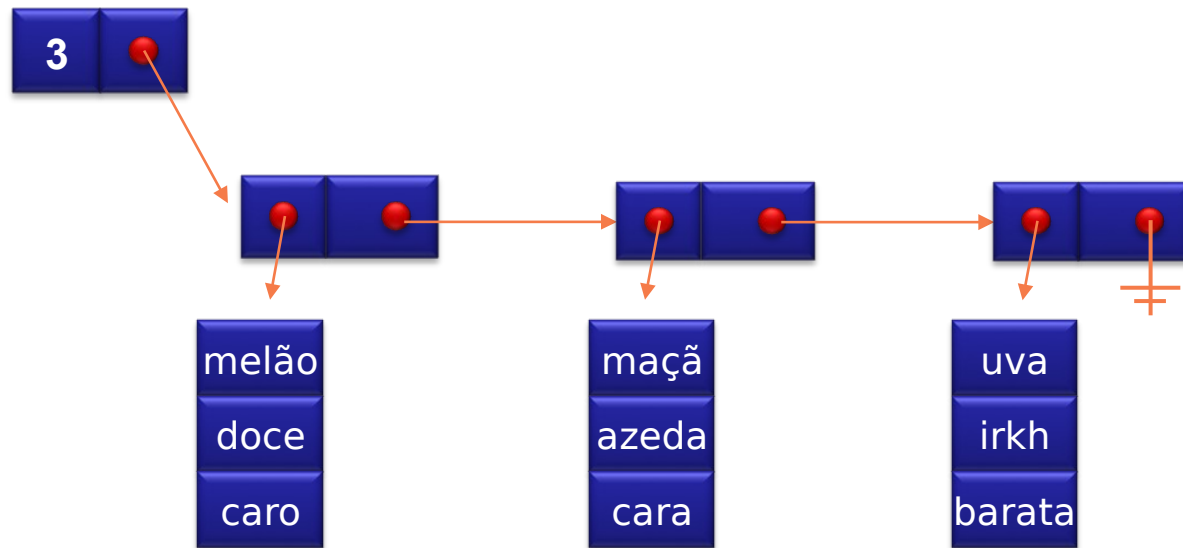


- Local: **lista.h**
  - T necessita de um destrutor próprio
  - Assim como a lista (neste caso a cabeça) vai precisar de um também



# Modelagem: Aspecto Funcional

- 3 Categorias de Operações:
  - colocar e retirar dados da lista;
  - testar se a lista está vazia e outros testes;
  - inicializá-la e garantir a ordem dos elementos.



# Modelagem: Operações da Lista

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

# Modelagem: Operações da Lista

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

# Algoritmo CriaLista

```
Lista* MÉTODO 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* MÉTODO 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* MÉTODO 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 MÉTODO listaVazia()  
  início  
    SE (tamanho = 0) ENTÃO  
      RETORNE (Verdadeiro)  
    SENÃO  
      RETORNE (Falso) ;  
  fim;
```

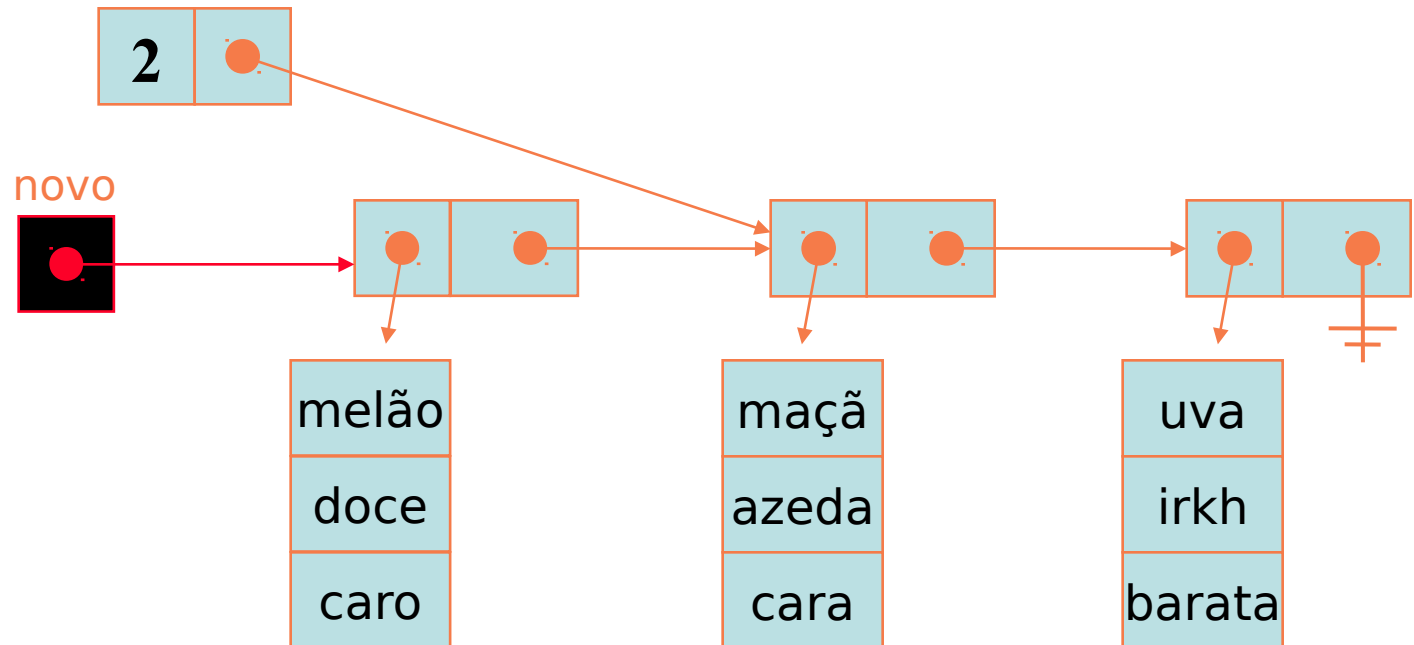
Um algoritmo ListaCheia não existe aqui;  
verificar se houve espaço na memória para um novo  
elemento será responsabilidade de cada operação  
de adição.

# Algoritmo AdicionaNoInicio

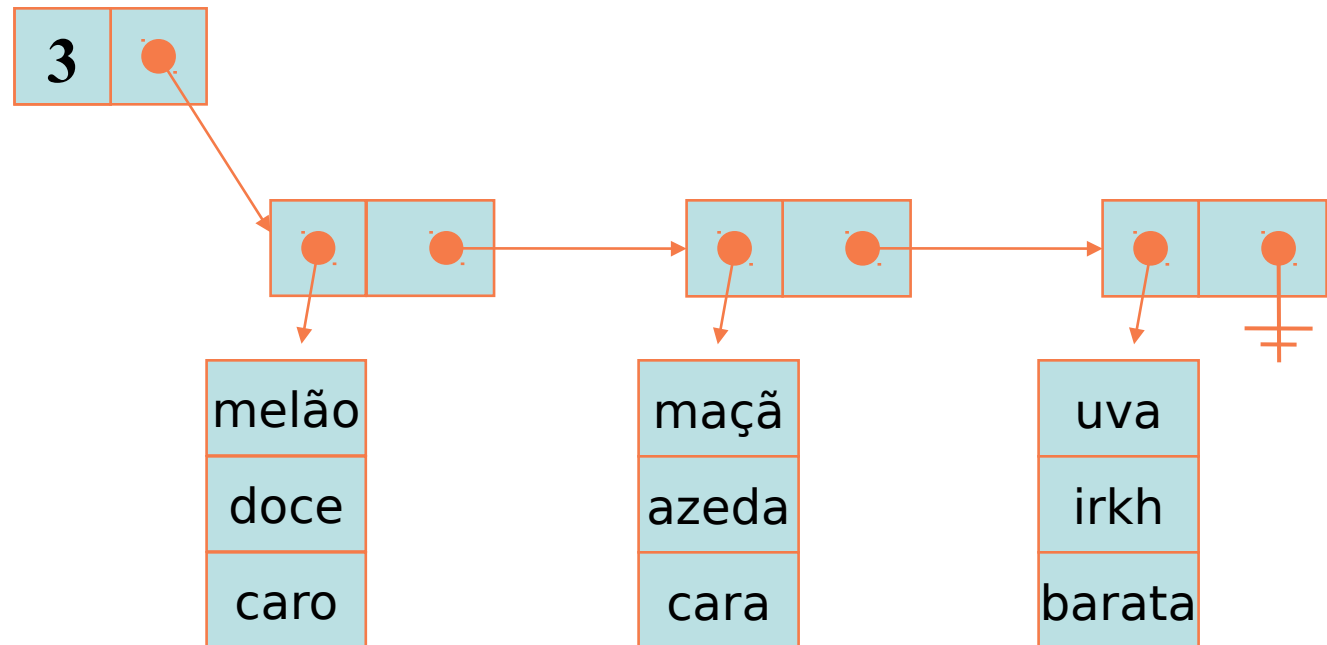
- Procedimento:
  - testamos se é possível alocar um elemento;
  - fazemos o próximo deste novo elemento ser o primeiro da lista;
  - fazemos a cabeça de lista apontar para o novo elemento.
- Parâmetros:
  - O (ponteiro do) dado a ser inserido;



# Algoritmo AdicionaNoInicio



# Algoritmo AdicionaNoInicio



# Algoritmo AdicionaNoInício

```
MÉTODO adicionaNoInício(T *dado)
    variáveis
        tElemento *novo; //Variável auxiliar.
    início
        novo <- alocue(tElemento);
        SE (novo = NULO) ENTÃO
            THROW(ErroListaCheia);
        SENÃO
            novo->próximo <- dados;
            novo->info <- dado;
            dados <- novo;
            tamanho <- tamanho + 1;

        FIM SE
    fim;
```

# Algoritmo AdicionaNoInício

```
MÉTODO adicionaNoInício(T *dado)
    variáveis
        tElemento *novo; //Variável auxiliar.
    início
        novo <- alocue(tElemento);
        SE (novo = NULO) ENTÃO
            THROW(ErroListaCheia);
        SENÃO
            novo->próximo <- dados;
            novo->info <- dado;
            dados <- novo;
            tamanho <- tamanho + 1;
        FIM SE
    fim;
```

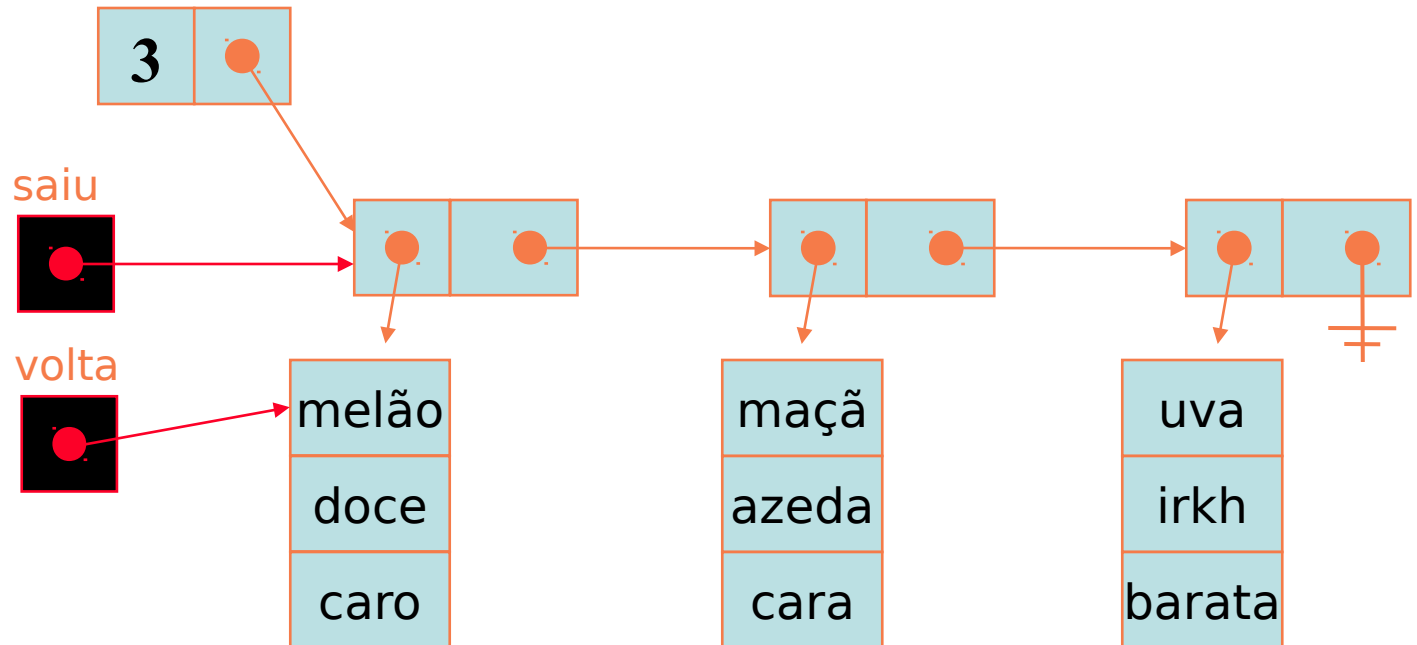
# Algoritmo AdicionaNoInício

```
MÉTODO adicionaNoInício(T *dado)
    variáveis
        tElemento *novo; //Variável auxiliar.
    início
        novo <- aloque(tElemento);
        SE (novo = NULO) ENTÃO
            THROW(ErroListaCheia);
        SENÃO
            novo->próximo <- dados;
            novo->info <- dado;
            dados <- novo;
            tamanho <- tamanho + 1;
        FIM SE
    fim;
```

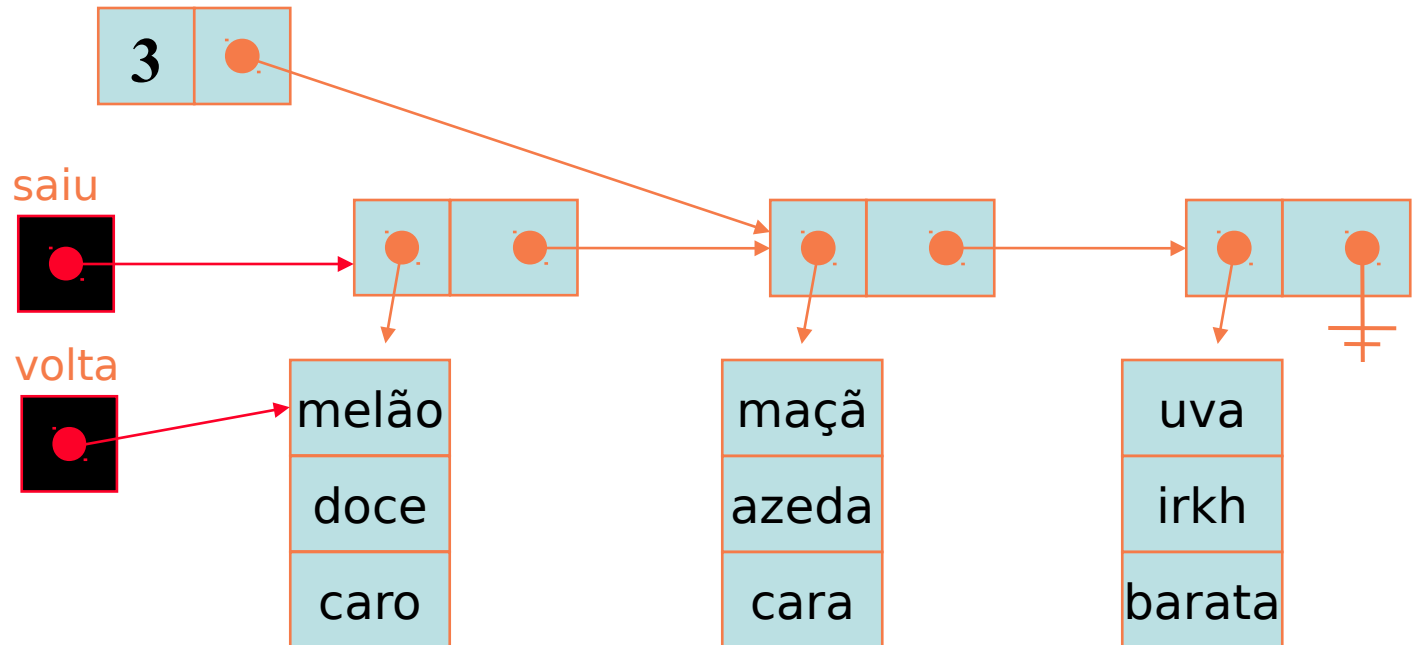
# Algoritmo RetiraDoInício

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

# Algoritmo RetiraDoInicio

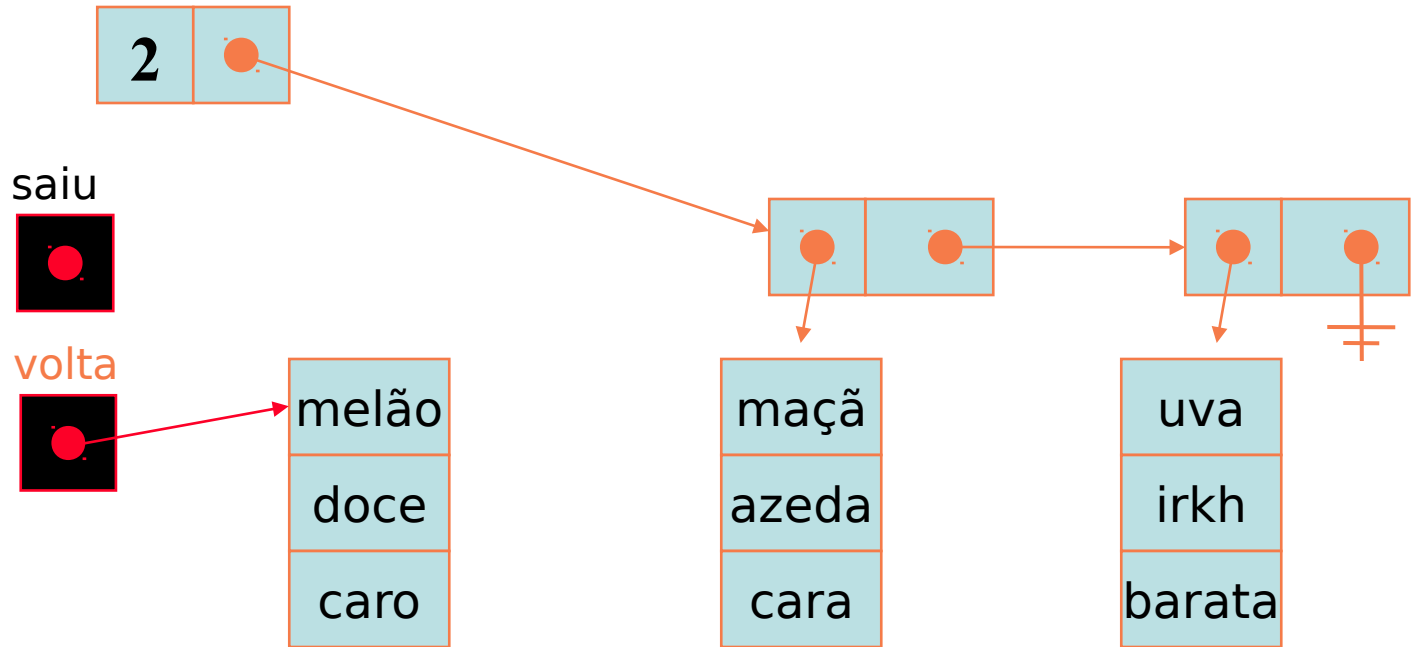


# Algoritmo RetiraDoInicio





# Algoritmo RetiraDoInicio



# Algoritmo RetiraDoInício

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

# Algoritmo EliminaDoInício

```
Inteiro MÉTODO eliminaDoInício()  
    //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()) ENTÃO  
            THROW(ErroListaVazia);  
        SENÃO  
            saiu <- dados;  
            dados <- saiu->próximo;  
            tamanho <- tamanho - 1;  
            LIBERE(saiu->info);  
            LIBERE(saiu);  
            RETORNE(tamanho);  
        FIM SE  
    fim;
```

# Algoritmo EliminaDoInício

- Observe que a linha **LIBERE(saiu->info)** possui um perigo:
  - se o T 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 T que será chamada no lugar de **LIBERE**.

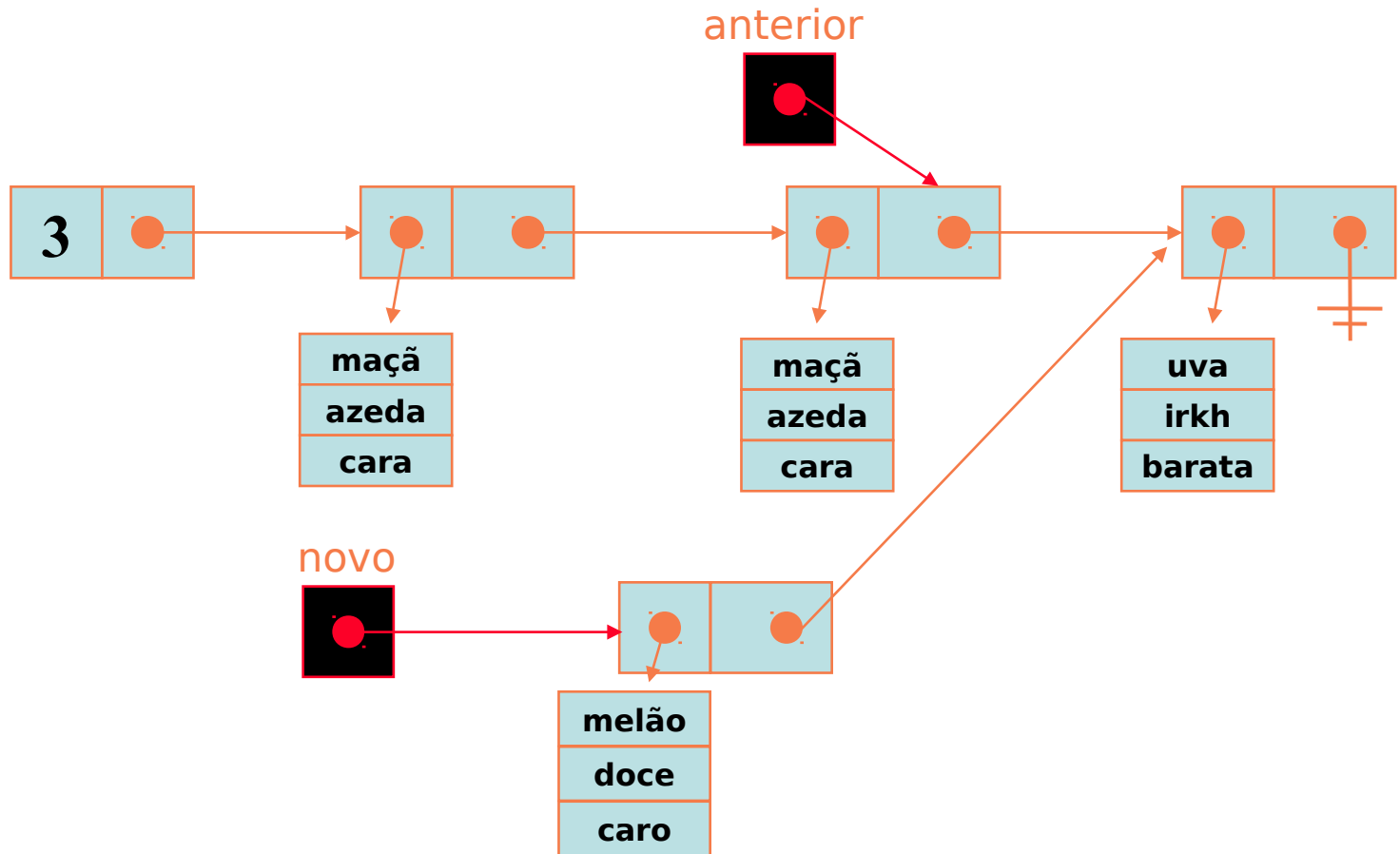
# Importância do Destrutor

- O destrutor diz como o objeto será destruído quando sair de escopo.
- No mínimo deve liberar a memória que foi alocada por chamadas “new” no construtor.
- Se nenhum destrutor for declarado será gerado um *default*, que aplicará o destrutor correspondente a cada dado da classe.
  - A recursão tem que ser garantida pelo objeto

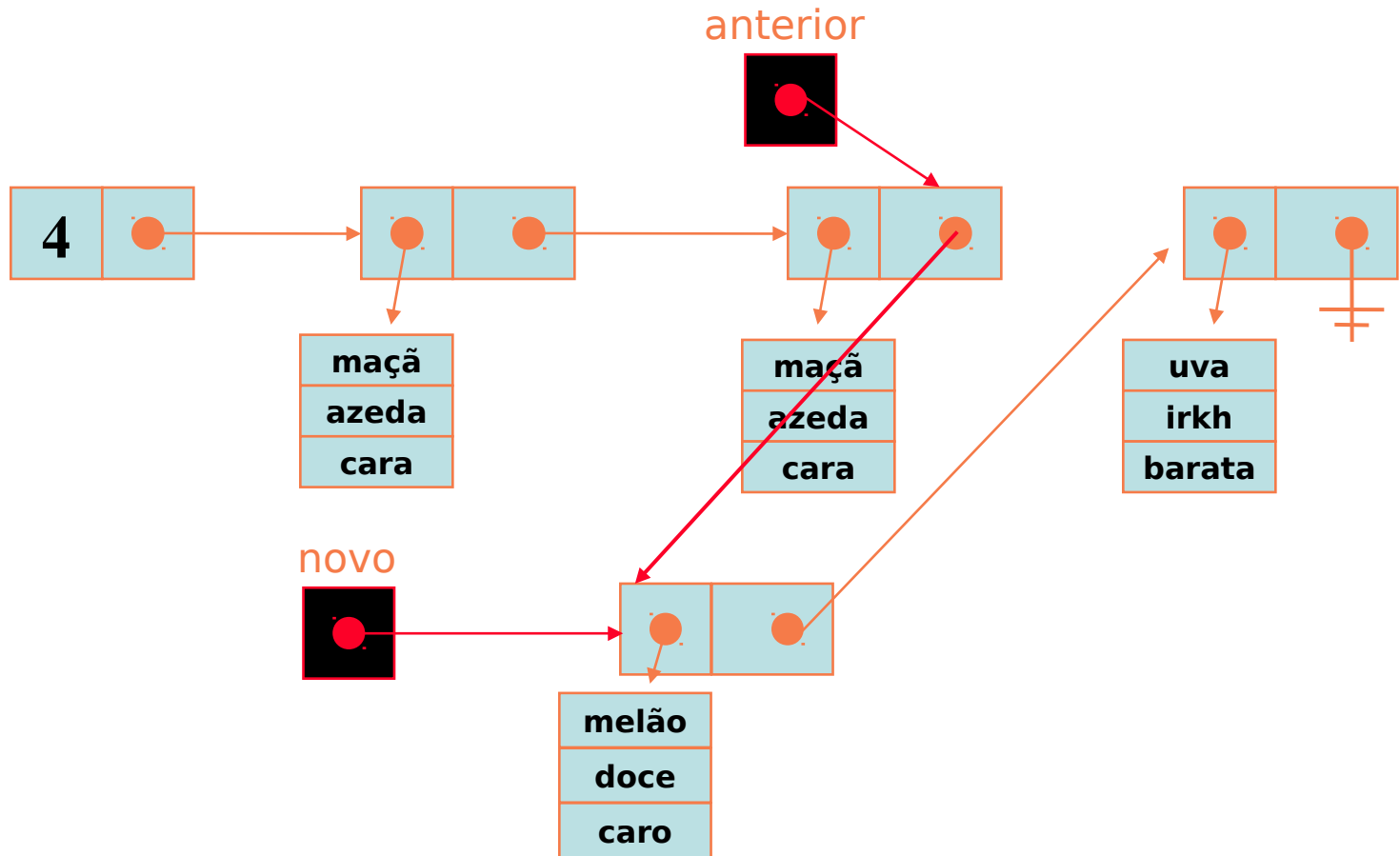
# Algoritmo AdicionaNaPosição

- 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;

# Algoritmo AdicionaNaPosição



# Algoritmo AdicionaNaPosição





# Algoritmo AdicionaNaPosição

```
Inteiro MÉTODO adicionaNaPosição(T *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 > tamanho + 1) ENTÃO
        TRHOW(ErroPosição)
    SENÃO
        SE (posição = 1) ENTÃO
            RETORNE(adicionaNoInício(info);
        SENÃO
            novo <- aloque(tElemento);
            SE (novo = NULO) ENTÃO
                THROW(ErroListaCheia);
            SENÃO
                anterior <- dados;
                REPITA (posição - 2) VEZES
                    anterior <- anterior->próximo;
                novo->próximo <- anterior->próximo;
                novo->info <- info;
                anterior->próximo <- novo;
                tamanho <- tamanho + 1;
                RETORNE(tamanho);
        FIM SE
    FIM SE
FIM SE
fim;
```

# Algoritmo AdicionaNaPosição

```
Inteiro MÉTODO adicionaNaPosição(T *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 > tamanho + 1) ENTÃO
        THROW(ErroPosição)
    SENÃO
        SE (posição = 1) ENTÃO
            RETORNE(adicionaNoInício(info));
        SENÃO
            novo <- aloque(tElemento);
            SE (novo = NULO) ENTÃO
                THROW(ErroListaCheia);
            SENÃO
                anterior <- dados;
                REPITA (posição - 2) VEZES
                    anterior <- anterior->próximo;
                novo->próximo <- anterior->próximo;
                novo->info <- info;
                anterior->próximo <- novo;
                tamanho <- tamanho + 1;
                RETORNE(tamanho);
        FIM SE
    FIM SE
FIM SE
fim;
```

# Algoritmo AdicionaNaPosição

```
Inteiro MÉTODO adicionaNaPosição(T *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 > tamanho + 1) ENTÃO
        THROW(ErroPosição)
    SENÃO
        SE (posição = 1) ENTÃO
            RETORNE(adicionaNoInício(info);
        SENÃO
            novo <- aloque(tElemento);
            SE (novo = NULO) ENTÃO
                THROW(ErroListaCheia);
            SENÃO
                anterior <- dados;
                REPITA (posição - 2) VEZES
                    anterior <- anterior->próximo;
                novo->próximo <- anterior->próximo;
                novo->info <- info;
                anterior->próximo <- novo;
                tamanho <- tamanho + 1;
                RETORNE(tamanho);
        FIM SE
    FIM SE
FIM SE
fim;
```

# Algoritmo AdicionaNaPosição

```
Inteiro MÉTODO adicionaNaPosição(T *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 > tamanho + 1) ENTÃO
        THROW(ErroPosição)
    SENÃO
        SE (posição = 1) ENTÃO
            RETORNE(adicionaNoInício(info);
        SENÃO
            novo <- aloque(tElemento);
            SE (novo = NULO) ENTÃO
                THROW(ErroListaCheia);
            SENÃO
                anterior <- dados;
                REPITA (posição - 2) VEZES
                    anterior <- anterior->próximo;
                novo->próximo <- anterior->próximo;
                novo->info <- info;
                anterior->próximo <- novo;
                tamanho <- tamanho + 1;
                RETORNE(tamanho);
        FIM SE
    FIM SE
FIM SE
fim;
```

# Algoritmo AdicionaNaPosição

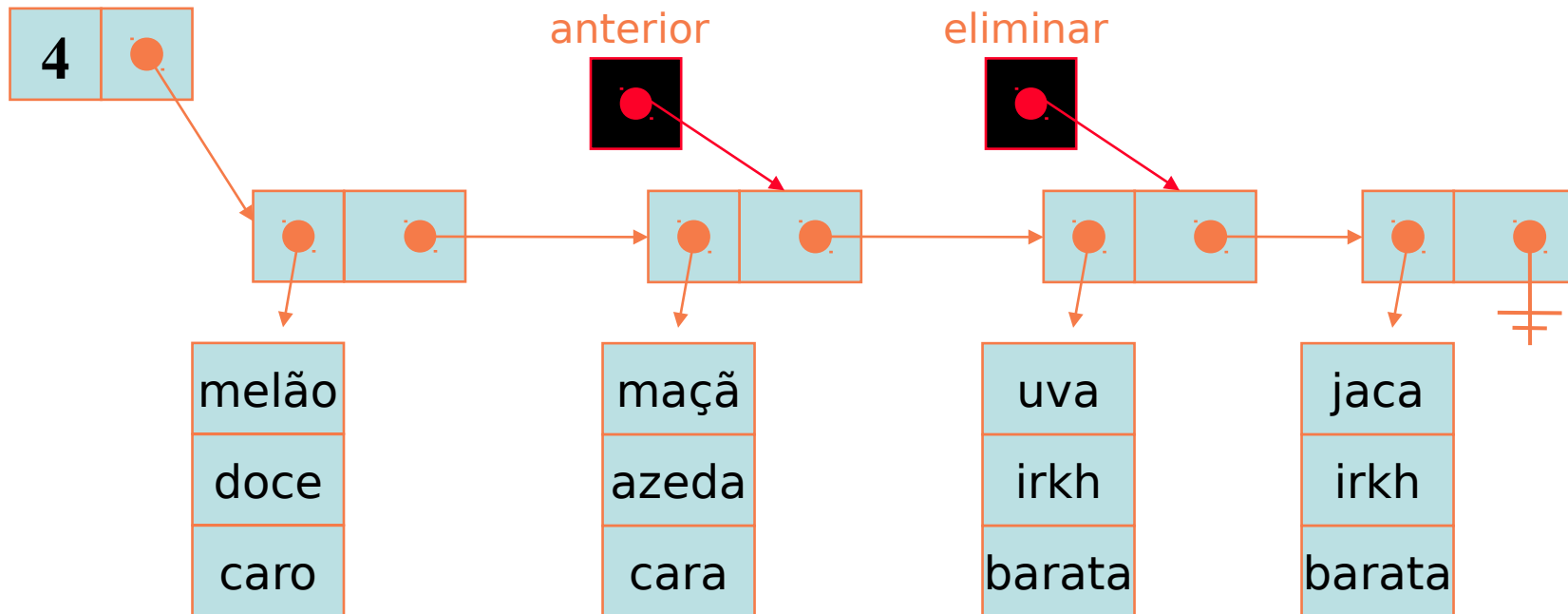
```
Inteiro MÉTODO adicionaNaPosição(T *info, inteiro posição)
//Adiciona novo elemento na posição informada.
//Retorna o novo número de elementos da lista ou erro.
variáveis
    tElemento *novo, *anterior; //Ponteiros auxiliares.
início
    SE (posição > tamanho + 1) ENTÃO
        THROW(ErroPosição)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

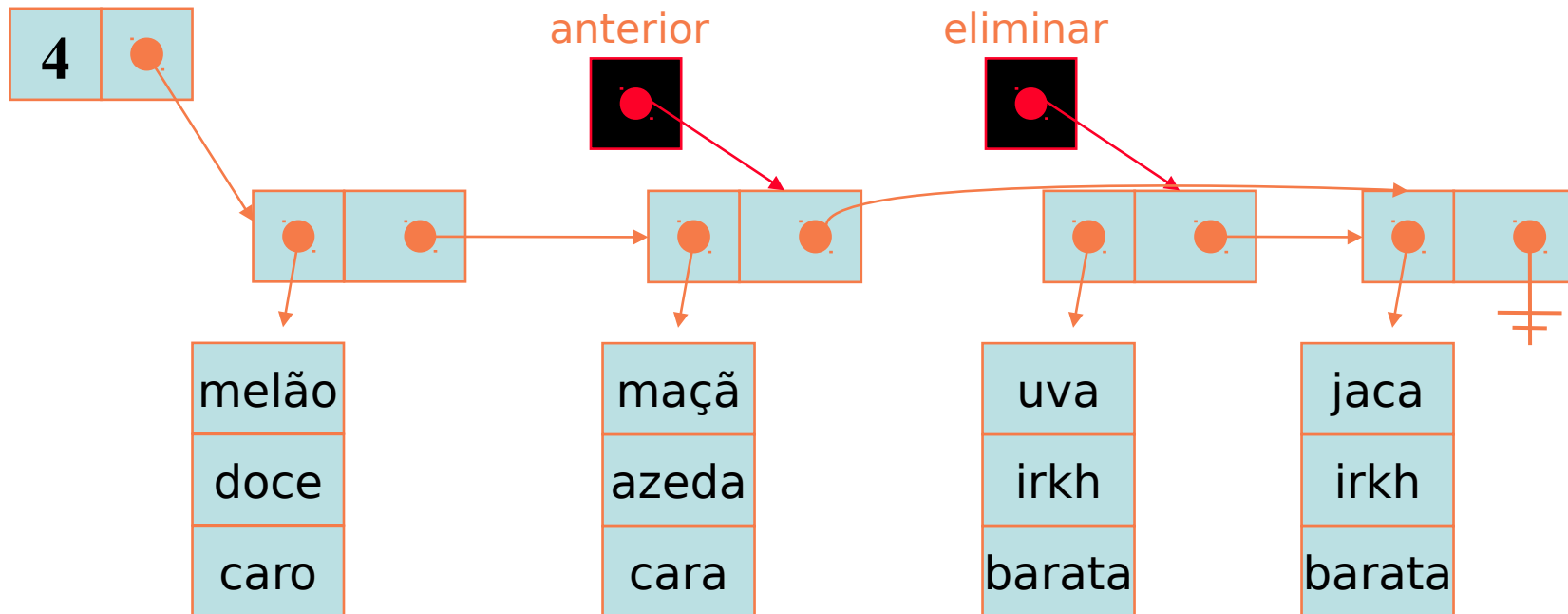
# Algoritmo RetiraDaPosição

Posições > 1



# Algoritmo RetiraDaPosição

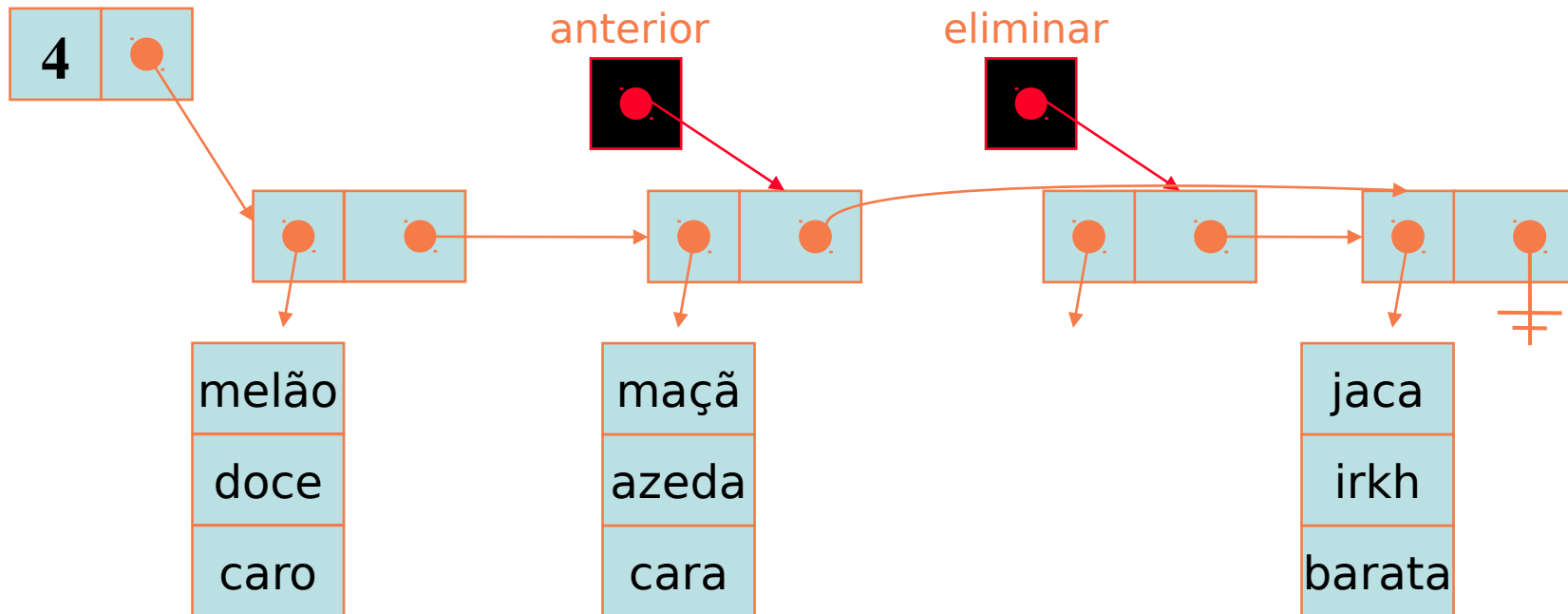
Posições > 1





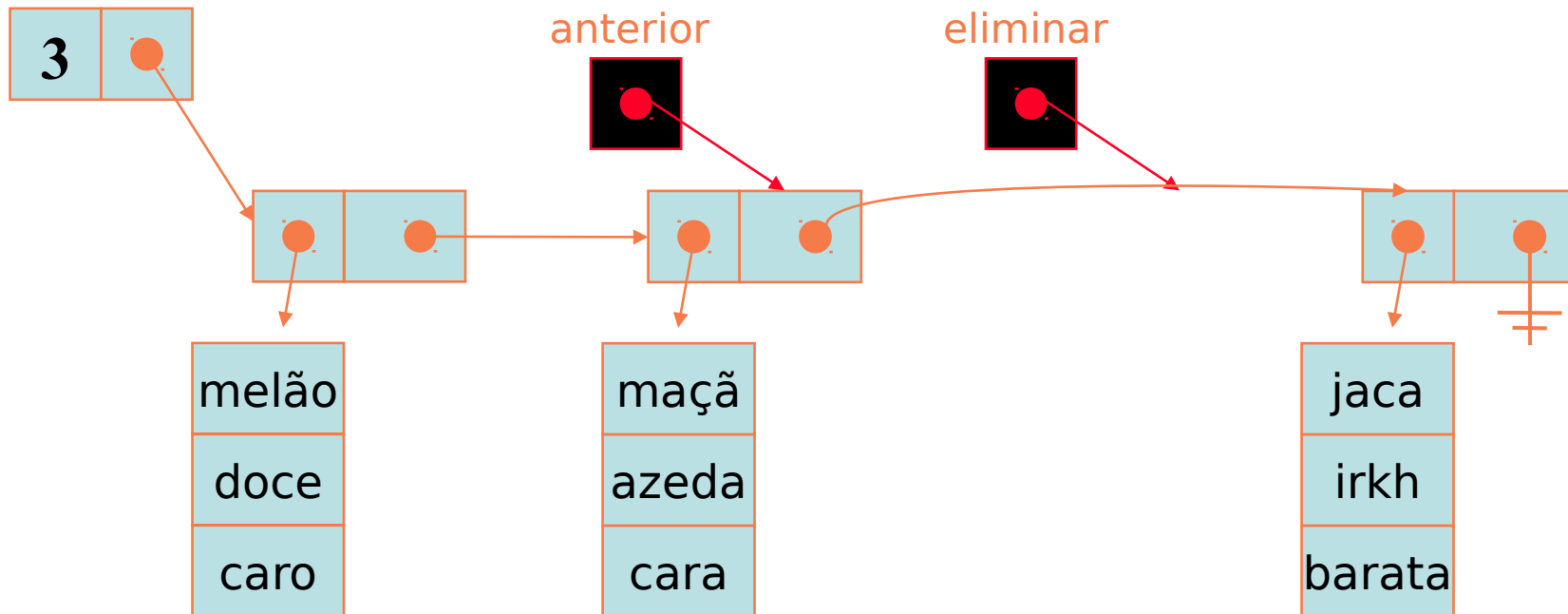
# Algoritmo RetiraDaPosição

Posições > 1



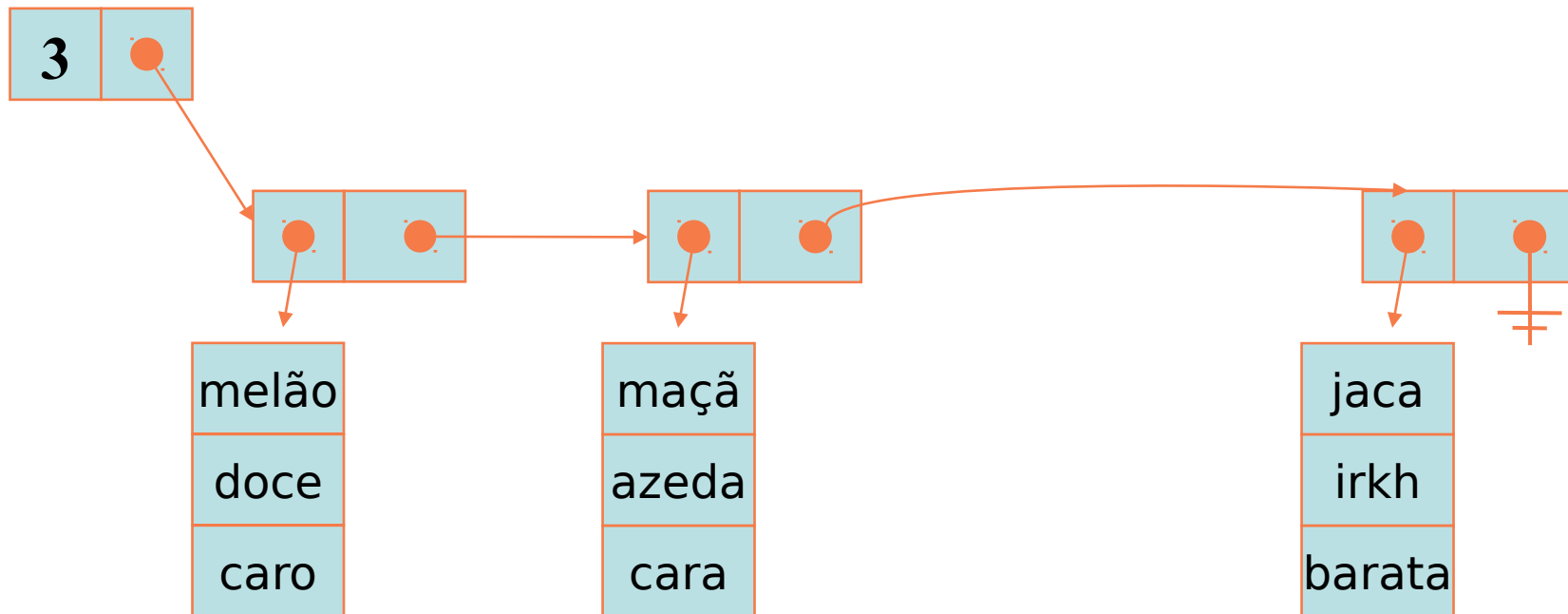
# Algoritmo RetiraDaPosição

Posições > 1



# Algoritmo RetiraDaPosição

Posições > 1



# Algoritmo RetiraDaPosição

```
T* MÉTODO retiraDaPosição(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.
    T *volta; //Variável auxiliar para o dado retornado.
início
    SE (posição > tamanho) ENTÃO
        RETORNE(NULO);
    SENÃO
        SE (posição = 1) ENTÃO
            RETORNE(retiraDoInício());
        SENÃO
            anterior <- dados;
            REPITA (posição - 2) VEZES
                anterior <- anterior->próximo;
            eliminar <- anterior->próximo;
            volta <- eliminar->info;
            anterior->próximo <- eliminar->próximo;
            tamanho <- tamanho - 1;
            LIBERE(eliminar);
            RETORNE(volta);
        FIM SE
    FIM SE
fim;
```

# Algoritmo RetiraDaPosição

```
T* MÉTODO retiraDaPosição(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.
    T *volta; //Variável auxiliar para o dado retornado.
início
    SE (posição > tamanho) ENTÃO
        RETORNE(NULO);
    SENÃO
        SE (posição = 1) ENTÃO
            RETORNE(retiraDoInício());
        SENÃO
            anterior <- dados;
            REPITA (posição - 2) VEZES
                anterior <- anterior->próximo;
            eliminar <- anterior->próximo;
            volta <- eliminar->info;
            anterior->próximo <- eliminar->próximo;
            tamanho <- tamanho - 1;
            LIBERE(eliminar);
            RETORNE(volta);
        FIM SE
    FIM SE
fim;
```

# Algoritmo RetiraDaPosição

```
T* MÉTODO retiraDaPosição(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.
    T *volta; //Variável auxiliar para o dado retornado.
início
    SE (posição > tamanho) ENTÃO
        RETORNE (NULO);
    SENÃO
        SE (posição = 1) ENTÃO
            RETORNE(retiraDoInício());
        SENÃO
            anterior <- dados;
            REPITA (posição - 2) VEZES
                anterior <- anterior->próximo;
            eliminar <- anterior->próximo;
            volta <- eliminar->info;
            anterior->próximo <- eliminar->próximo;
            tamanho <- tamanho - 1;
            LIBERE(eliminar);
            RETORNE(volta);
        FIM SE
    FIM SE
fim;
```

# Algoritmo RetiraDaPosição

```
T* MÉTODO retiraDaPosição(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.
    T *volta; //Variável auxiliar para o dado retornado.
início
    SE (posição > tamanho) ENTÃO
        RETORNE(NULO);
    SENÃO
        SE (posição = 1) ENTÃO
            RETORNE(retiraDoInício());
        SENÃO
            anterior <- dados;
            REPITA (posição - 2) VEZES
                anterior <- anterior->próximo;
            eliminar <- anterior->próximo;
            volta <- eliminar->info;
            anterior->próximo <- eliminar->próximo;
            tamanho <- tamanho - 1;
            LIBERE(eliminar);
            RETORNE(volta);
        FIM SE
    FIM SE
fim;
```

# Algoritmo RetiraDaPosição

```
T* MÉTODO retiraDaPosição(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.
    T *volta; //Variável auxiliar para o dado retornado.
início
    SE (posição > tamanho) ENTÃO
        RETORNE (NULO);
    SENÃO
        SE (posição = 1) ENTÃO
            RETORNE (retiraDoInício());
        SENÃO
            anterior <- dados;
            REPITA (posição - 2) VEZES
                anterior <- anterior->próximo;
            eliminar <- anterior->próximo;
            volta <- eliminar->info;
            anterior->próximo <- eliminar->próximo;
            tamanho <- tamanho - 1;
            LIBERE (eliminar);
            RETORNE (volta);
        FIM SE
    FIM SE
fim;
```



# Modelagem do T

- 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 Classe T e não da lista;
  - devem ser implementadas como tal.
- Operações: testar AS INFORMAÇÕES:
  - Igual(dado1, dado2) → operator=
  - Maior(dado1, dado2) → operator>
  - Menor(dado1, dado2) → operator<

# Algoritmo AdicionaEmOrdem

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

# Algoritmo AdicionaEmOrdem

```
Inteiro MÉTODO adicionaEmOrdem(T *dado)
    variáveis
        tElemento *atual; //Variável auxiliar para caminhar.
        inteiro posição;
    início
        SE (listaVazia()) ENTÃO
            RETORNE(adicionaNoInício(dado));
        SENÃO
            atual <- 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(dado, posição + 1));
            SENÃO
                RETORNE(adicionaNaPosição(dado, posição));
            FIM SE
        FIM SE
    fim;
```

# Algoritmos Restantes

- Por conta do aluno:
  - Adiciona(dado)
    - AdicionaNaPosicao(tamanho);
  - Retira()
    - RetiraDaPosicao(tamanho);
  - RetiraEspecífico(dado)
- Operações - inicializar ou limpar:
  - DestróiLista()

# 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;
```

# Exercício

- Implemente uma classe ListaEncadeada com todas as operações vistas;
- Implemente a lista usando Templates
- Implemente a lista com um numero de elementos variável definido na instanciação
- Use as melhores práticas de orientação a objetos
- Documente todas as classes, métodos e atributos.
- Aplique os testes unitários disponíveis no moodle da disciplina para validar sua estrutura de dados.



## Atribuição-Uso Não-Comercial-Compartilhamento pela Licença 2.5 Brasil

### *Você pode:*

- copiar, distribuir, exibir e executar a obra
- criar obras derivadas

### *Sob as seguintes condições:*

Atribuição — Você deve dar crédito ao autor original, da forma especificada pelo autor ou licenciante.

Uso Não-Comercial — Você não pode utilizar esta obra com finalidades comerciais.

Compartilhamento pela mesma Licença — Se você alterar, transformar, ou criar outra obra com base nesta, você somente poderá distribuir a obra resultante sob uma licença idêntica a esta.

Para ver uma cópia desta licença, visite <http://creativecommons.org/licenses/by-nc-sa/2.5/br/> ou mande uma carta para Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.