Algoritmos e Estrutura de Dados II

Prof. Fellipe Guilherme Rey de Souza

Aula 07 – Lista (Implementação)

Agenda

- Possíveis implementações
- Lista com Vetor
- Lista Encadeada
- Lista Duplamente Encadeada

Possíveis Implementações

- Existem duas principais formas de implementarmos uma Lista:
  - Usando um vetor para armazenar os valores
  - Usando alocação dinâmica de memória para armazenar os valores
    - Aqui, podemos ter duas possíveis implementações: Listas Encadeadas e Listas Duplamente Encadeadas.

- A seguir, veremos um pouco mais das duas implementações, incluindo os seus prós e contras.

Possíveis Implementações

- Importante salientar que a Lista é um **tipo abstrato de dado**. Isso quer dizer que existe mais de uma implementação possível para a Lista.

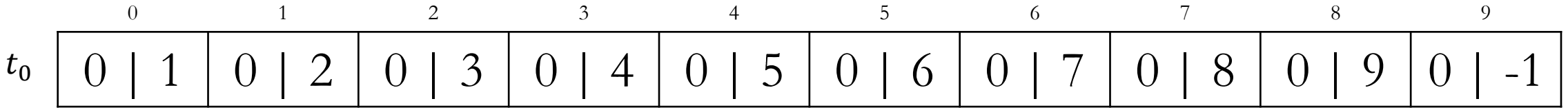
- Os conteúdos que serão abordados na aula de hoje mostram **uma possível implementação** de Lista, seja utilizando vetores ou seja utilizando alocação dinâmica de memória.

Lista com Vetores

- A implementação da Lista com vetores utiliza-se da estrutura já existente das linguagens de programação (vetor/array) para armazenar seus valores.

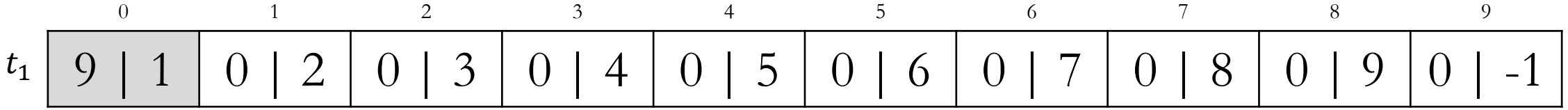
- A implementação é simples: Basta usar um vetor (do tipo que for a Lista) e criar três novas variáveis adicionais: **comeco** e **disponivel**.

Listas com Vetores



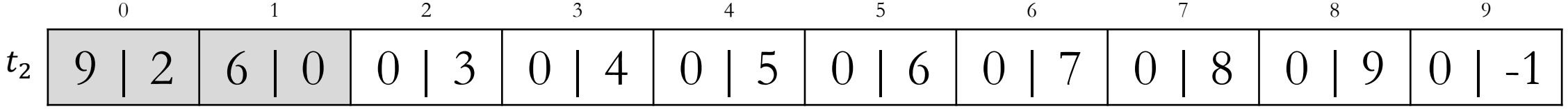
Começo: -1 / **Disponível**: 0

Ação: Inserir o item "9"



Começo: 0 / **Disponível**: 1

Ação: Inserir o item "6"



Começo: 1 / **Disponível**: 2

Listas com Vetores

Ação: Ver o primeiro (Retorna o "6" sem mexer na Lista)

t_3

0	1	2	3	4	5	6	7	8	9
9 | 2	6 | 0	0 | 3	0 | 4	0 | 5	0 | 6	0 | 7	0 | 8	0 | 9	0 | -1

Começo: 1 / **Disponível**: 2

Ação: Remover o item 9

t_4

0	1	2	3	4	5	6	7	8	9
0 | -1	6 | 2	0 | 3	0 | 4	0 | 5	0 | 6	0 | 7	0 | 8	0 | 9	0 | 0

Começo: 1 / **Disponível**: 2

Ação: Inserir o item "8"

t_5

0	1	2	3	4	5	6	7	8	9
0 | -1	6 | 2	8 | 3	0 | 4	0 | 5	0 | 6	0 | 7	0 | 8	0 | 9	0 | 0

Começo: 1 / **Disponível**: 3

Listas com Vetores

```c
#define TAM 8

typedef struct {
    int info;
    int prox;
} Node;

int comeco, disponivel;
```

- Definição da Lista
  - Defini um tamanho máximo para a Lista (neste exemplo, o tamanho é 8)
  - Criei uma Lista do tipo Node (com info e prox).
  - Criei duas variáveis globais auxiliares: comeco (para definir o índice que a lista começa) e disponivel (para definir o próximo índice disponível para inserir)

Listas com Vetores

- Iniciar a Lista

```
void iniciaLista(Node *lista, int *comeco, int *disponivel) {
    for (int i = 0; i < TAM-1; i++) {
        lista[i].info = 0;
        lista[i].prox = i + 1;
    }
    lista[TAM-1].info = 0;
    lista[TAM-1].prox = -1;
    *comeco = -1;
    *disponivel = 0;
}
```

  - Para iniciar a lista, receberemos a lista, comeco e disponivel.
  - Para cada elemento da lista, faremos com que o valor (info) seja zero e o prox seja o índice + 1.
  - Para o último elemento, o valor do prox será -1 (não tem mais posições disponíveis), começo vai ser -1 e disponível será 0.

Listas com Vetores

- Inserir

```c
void adicionaNaLista(Node *lista, int *comeco, int *disponivel, int valor) {
    if (*disponivel == -1) {
        printf("Lista cheia! Impossivel adicionar o valor %d na lista \n", valor);
    } else {
        if (*comeco == -1) {
            *comeco = *disponivel;
        }
        lista[*disponivel].info = valor;
        *disponivel = lista[*disponivel].prox;
        printf("Valor %d adicionado com sucesso na lista!\n", valor);
    }
}
```

  - Para Inserir, precisamos receber a Lista, comeco, disponivel e o valor.
  - Se o disponível for igual a -1, significa que a lista está cheia.
  - Caso contrário, se o começo for -1, significa que a lista está vazia. Vamos fazer com que o começo seja agora disponível (onde iremos inserir). Depois, basta atribuirmos o novo valor ao indice disponível.

Listas com Vetores

- Remover
  - Para remover, temos duas possibilidades. A primeira é remover um item do início a lista e a outra é remover do meio ou fim.
  - Se a remoção for no início, ele faz o seguinte passo-a-passo:
    1. Atualiza o ponteiro *comeco* para o próximo nó.

```
void removerDoInicio(Node *lista, int *comeco, int *disponivel, int valor) {
    int fim = lista[*comeco].prox;
    while(lista[fim].prox != -1) {
        fim = lista[fim].prox;
    }
    lista[fim].prox = *comeco;
    int antigoComeco = *comeco;
    *comeco = lista[*comeco].prox;
    lista[antigoComeco].prox = -1;

    if (*disponivel == -1) {
        *disponivel = antigoComeco;
    }
}

void removerDoMeioOuFim(Node *lista, int *comeco, int *disponivel, int valor) {
    int atual = *comeco;
    int anterior = -1;
    while (lista[atual].info != valor && lista[atual].prox != -1) {
        anterior = atual;
        atual = lista[atual].prox;
    }
    int fim = atual;
    while (lista[fim].prox != -1) {
        fim = lista[fim].prox;
    }
    if (atual != fim) {
        lista[anterior].prox = lista[atual].prox;
        lista[fim].prox = atual;
        lista[atual].prox = -1;
    } else if (lista[atual].info != valor) {
    }
    if (*disponivel == -1) {
        *disponivel = atual;
    }
}

void removerDaLista(Node *lista, int *comeco, int *disponivel, int valor) {
    if (lista[*comeco].info == valor) {
        removerDoInicio(lista, &(*comeco), &(*disponivel), valor);
    } else {
        removerDoMeioOuFim(lista, &(*comeco), &(*disponivel), valor);
    }

}
```

Listas com Vetores

- Remover (*cont.*)
  - Se a remoção for no início, ele faz o seguinte:
    1. Atualiza o ponteiro *comeco* para o próximo nó.
    2. Desfaz o link do nó removido, fazendo com que ele não aponte para mais nada (por meio da definição de prox para -1).
    3. Conecta o último nó da lista ao primeiro nó, criando um ciclo.
    4. Marca o índice do nó removido como o próximo espaço disponível na lista.

```
void removerDoInicio(Node *lista, int *comeco, int *disponivel, int valor) {
    int fim = lista[*comeco].prox;
    while(lista[fim].prox != -1) {
        fim = lista[fim].prox;
    }
    lista[fim].prox = *comeco;
    int antigoComeco = *comeco;
    *comeco = lista[*comeco].prox;
    lista[antigoComeco].prox = -1;

    if (*disponivel == -1) {
        *disponivel = antigoComeco;
    }
}

void removerDoMeioOuFim(Node *lista, int *comeco, int *disponivel, int valor) {
    int atual = *comeco;
    int anterior = -1;
    while (lista[atual].info != valor && lista[atual].prox != -1) {
        anterior = atual;
        atual = lista[atual].prox;
    }
    int fim = atual;
    while (lista[fim].prox != -1) {
        fim = lista[fim].prox;
    }
    if (atual != fim) {
        lista[anterior].prox = lista[atual].prox;
        lista[fim].prox = atual;
        lista[atual].prox = -1;
    } else if (lista[atual].info != valor) {
    }
    if (*disponivel == -1) {
        *disponivel = atual;
    }
}

void removerDaLista(Node *lista, int *comeco, int *disponivel, int valor) {
    if (lista[*comeco].info == valor) {
        removerDoInicio(lista, &(*comeco), &(*disponivel), valor);
    } else {
        removerDoMeioOuFim(lista, &(*comeco), &(*disponivel), valor);
    }

}
```

Listas com Vetores

- Remover (*cont.*)
  - Se a remoção for no meio ou fim, ele faz o seguinte:
    1. Busca o nó com o valor especificado.
    2. Caso o nó esteja no meio, remove o nó e conecta o nó anterior ao próximo.
    3. Caso o nó esteja no final, ajusta o ponteiro do último nó.
    4. Marca o nó removido como disponível para reutilização.

```c
void removerDoInicio(Node *lista, int *comeco, int *disponivel, int valor) {
    int fim = lista[*comeco].prox;
    while(lista[fim].prox != -1) {
        fim = lista[fim].prox;
    }
    lista[fim].prox = *comeco;
    int antigoComeco = *comeco;
    *comeco = lista[*comeco].prox;
    lista[antigoComeco].prox = -1;

    if (*disponivel == -1) {
        *disponivel = antigoComeco;
    }
}

void removerDoMeioOuFim(Node *lista, int *comeco, int *disponivel, int valor) {
    int atual = *comeco;
    int anterior = -1;
    while (lista[atual].info != valor && lista[atual].prox != -1) {
        anterior = atual;
        atual = lista[atual].prox;
    }
    int fim = atual;
    while (lista[fim].prox != -1) {
        fim = lista[fim].prox;
    }
    if (atual != fim) {
        lista[anterior].prox = lista[atual].prox;
        lista[fim].prox = atual;
        lista[atual].prox = -1;
    } else if (lista[atual].info != valor) {
    }
    if (*disponivel == -1) {
        *disponivel = atual;
    }
}

void removerDaLista(Node *lista, int *comeco, int *disponivel, int valor) {
    if (lista[*comeco].info == valor) {
        removerDoInicio(lista, &(*comeco), &(*disponivel), valor);
    } else {
        removerDoMeioOuFim(lista, &(*comeco), &(*disponivel), valor);
    }

}
```

Listas com Vetores

- **Vantagens**:
  i. Acesso rápido aos elementos
     - Como um vetor é uma estrutura de dados contígua na memória, o acesso aos elementos é muito rápido, com complexidade **O(1)**, tanto para inserção quanto remoção.

Mesmas vantagens da implementação da Pilha com vetores!

Listas com Vetores

- **Desvantagens:**
  i. Tamanho fixo (em vetores estáticos)
     - Se o vetor for alocado com um tamanho fixo, ele pode levar a um desperdício de memória se o vetor não for completamente utilizado ou a necessidade de realocar o vetor quando ele se lota, o que pode ser custoso (complexidade **O(n)** na realocação e cópia).
  ii. Necessidade de percorrer toda lista na remoção do meio/fim
     - Ao removermos um nó que está no meio ou fim da lista, precisamos percorrer toda a lista para fazer com que o último nó removido seja agora o próximo nó do antigo último nó (ou seja, o nó removido será o *prox* do nó que tinha o *prox* como -1).

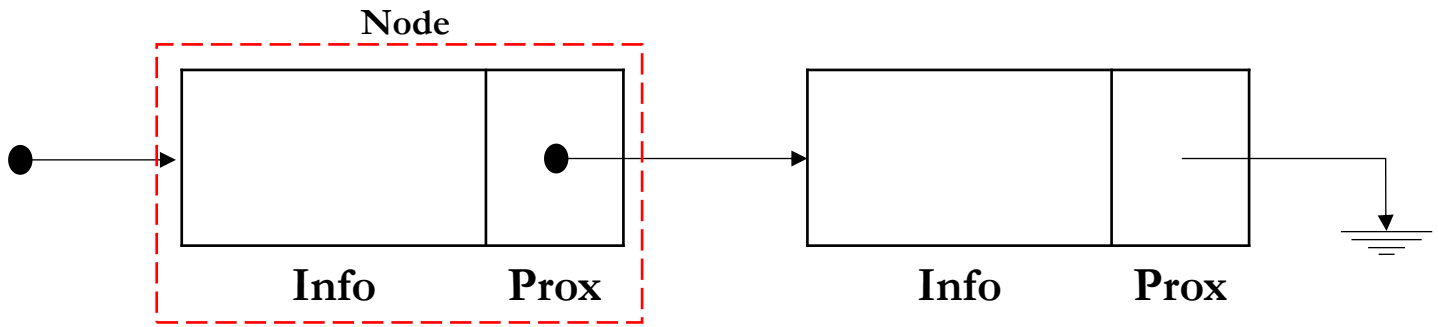
Listas Encadeada

- Como citado na nossa aula anterior, a implementação da Lista utilizando o vetor não é uma boa solução.
  - Diferentemente das implementações de Pilha e Fila usando vetores (que eram consideravelmente mais fáceis), a implementação da Lista adiciona um pouco mais de complexidade na implementação.
  - Esse acréscimo na complexidade de implementação não justifica o seu uso como uma opção viável.

Listas Encadeada

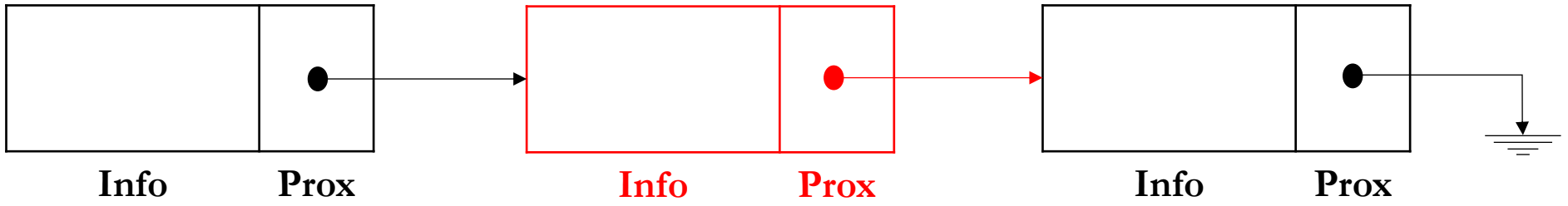
- Usando alocação dinâmica de memória, conseguimos implementar a nossa lista de duas maneiras diferentes.
  - **Lista Encadeada**
    - Também conhecida na literatura como Lista Ligada, Lista Simplesmente Ligada ou Lista Simplesmente Encadeada.
  - **Lista Duplamente Encadeada**
    - Também conhecida na literatura como Lista Duplamente Ligada.

Listas Encadeada



- Assim como fizemos para a Pilha e Fila, iremos definir a mesma estrutura chamada Node, que contém dois campos:
  - **Info**, que armazena o conteúdo de cada item da nossa Lista.
  - **Prox**, que armazena o endereço de memória do próximo item da nossa Lista.

Listas Encadeada



- Utilizando esta estrutura, a cada novo item a ser inserido, basta criar outro Node e realizar as atribuições para que este Node ocupe seu lugar na lista!
  - Seja no começo, meio ou fim, a depender do elemento a ser inserido.
  - Lembrando que estamos inserindo os itens de forma ordenada na lista ☺

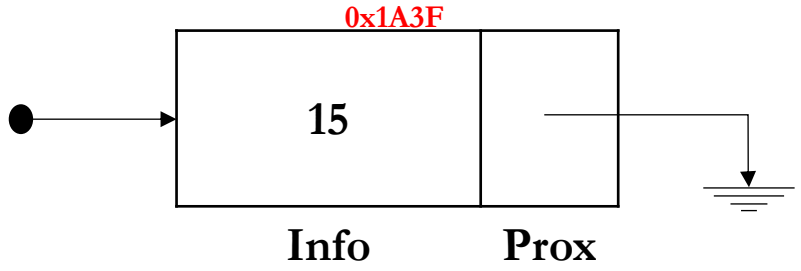
Listas Encadeada

t_0 **NULL**

Começo: NULL

Ação: Inserir o item 15

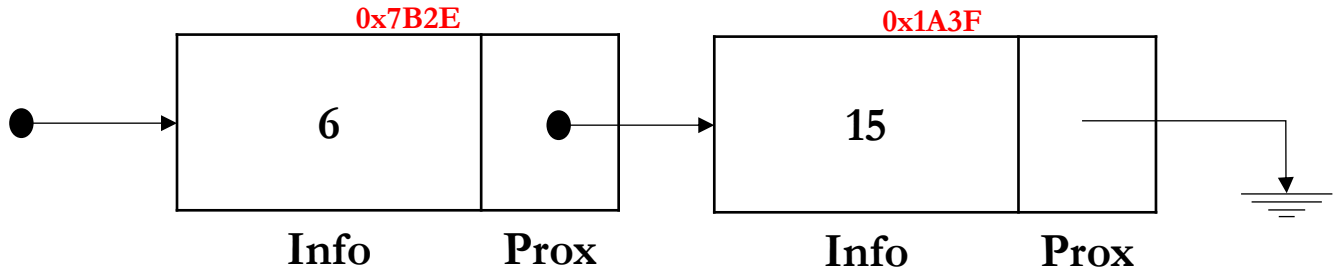
t_1



Começo: **0x1A3F**

Ação: Inserir o item 6

t_2

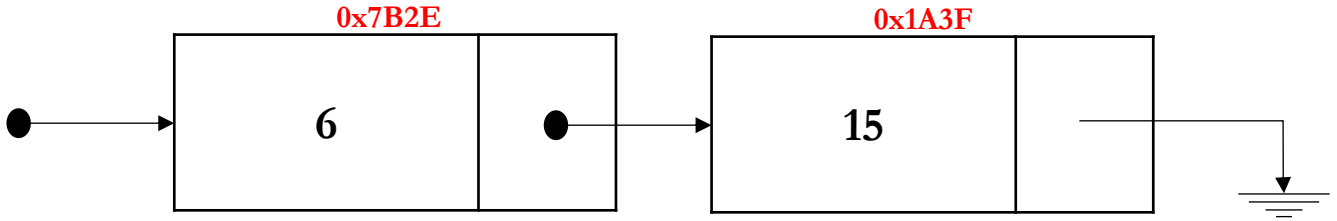


Começo: **0x7B2E**

Listas Encadeada

Ação: Buscar o elemento 15 (retorna 0x1A3F sem mexer na lista)

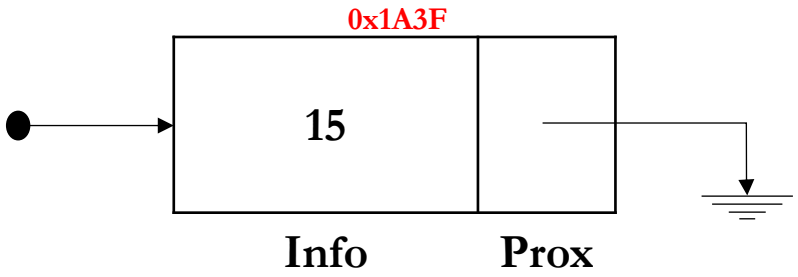
t_3



Começo: 0x7B2E

Ação: Remover o item 6 (Retorna 0x7B2E)

t_4



Começo: **0x1A3F**

Ação: Remover o item 15 (Retorna **0x1A3F**)

t_5 **NULL**

Começo: **NULL**

Listas Encadeada

```c
#define TRUE 1
#define FALSE 0

typedef struct no {
    int info;
    struct no *prox;
} Node;

typedef struct {
    struct no *comeco;
} Lista;
```

- Estrutura
  - Para criar a estrutura, precisamos criar dois tipos novos: Node e Lista.
  - O Node possui dois atributos: info (do tipo dos elementos da Lista) e o prox (do tipo no) – exatamente a mesma estrutura usada no exemplo da Pilha e Fila.
  - A Lista possui um no inicial, chamado comeco.

Listas Encadeada

- Inserir
  - Para inserir um elemento, primeiramente criamos duas variáveis auxiliares:
    - **ant**, que armazenará o valor anterior quando formos percorrer a lista e;
    - **aux**, que receberá o começo da lista (para podemos caminhar na lista sem modificar seu começo).

```
void insere(Lista *lista, int valor) {
    printf("Inserindo valor %d na lista\n", valor);
    Node *ant = NULL;
    Node *aux = lista->comeco;
    Node *novoNoh = (Node*) malloc(sizeof(Node));
    if (!novoNoh) {
        printf("Impossível adicionar. Lista cheia!\n");
    } else if (aux == NULL || aux->info > valor) {
        // LISTA VAZIA ou o valor a ser inserido
        // é menor que o primeiro
        novoNoh->info = valor;
        novoNoh->prox = lista->comeco;
        lista->comeco = novoNoh;
    } else { // LISTA NÃO VAZIA
        novoNoh->info = valor;
        while (aux != NULL && aux->info < valor) {
            ant = aux;
            aux = aux->prox;
        }
        novoNoh->prox = ant->prox;
        ant->prox = novoNoh;
    }
}
```

Listas Encadeada

- Inserir (*cont.*)
  - Criamos o *novoNoh* (mesma ideia do *newElement*) e fazemos uma verificação: Se não foi possível alocar memória para ele, printamos uma mensagem de erro.
  - Caso contrário, vamos para a nosso primeiro caso: A lista está vazia ou o elemento a ser inserido é menor que o primeiro da lista.

```
void insere(Lista *lista, int valor) {
    printf("Inserindo valor %d na lista\n", valor);
    Node *ant = NULL;
    Node *aux = lista->comeco;
    Node *novoNoh = (Node*) malloc(sizeof(Node));
    if (!novoNoh) {
        printf("Impossível adicionar. Lista cheia!\n");
    } else if (aux == NULL || aux->info > valor) {
        // LISTA VAZIA ou o valor a ser inserido
        // é menor que o primeiro
        novoNoh->info = valor;
        novoNoh->prox = lista->comeco;
        lista->comeco = novoNoh;
    } else { // LISTA NÃO VAZIA
        novoNoh->info = valor;
        while (aux != NULL && aux->info < valor) {
            ant = aux;
            aux = aux->prox;
        }
        novoNoh->prox = ant->prox;
        ant->prox = novoNoh;
    }
}
```

Listas Encadeada

- Inserir (*cont.*)
  - Neste cenário, atribuimos o valor ao novoNoh, definimos o antigo começo da lista como o próximo elemento e definimos que o novoNoh agora é o começo da lista.
  - No próximo cenários, lidamos com uma inserção de um elemento que está no meio ou fim da lista.

```c
void insere(Lista *lista, int valor) {
    printf("Inserindo valor %d na lista\n", valor);
    Node *ant = NULL;
    Node *aux = lista->comeco;
    Node *novoNoh = (Node*) malloc(sizeof(Node));
    if (!novoNoh) {
        printf("Impossível adicionar. Lista cheia!\n");
    } else if (aux == NULL || aux->info > valor) {
        // LISTA VAZIA ou o valor a ser inserido
        // é menor que o primeiro
        novoNoh->info = valor;
        novoNoh->prox = lista->comeco;
        lista->comeco = novoNoh;
    } else { // LISTA NÃO VAZIA
        novoNoh->info = valor;
        while (aux != NULL && aux->info < valor) {
            ant = aux;
            aux = aux->prox;
        }
        novoNoh->prox = ant->prox;
        ant->prox = novoNoh;
    }
}
```

Listas Encadeada

- Inserir (*cont.*)
  - Neste cenário, atribuimos o valor ao *novoNoh* e percorremos a lista enquanto o ponteiro (aux) não for nulo ou o valor do nó atual for menor que o valor a ser inserido.
  - Após achar onde o nó precisa ser inserido, nós definimos o próximo do nó como o próximo do anterior e o próximo do anterior como o novo nó.

```
void insere(Lista *lista, int valor) {
    printf("Inserindo valor %d na lista\n", valor);
    Node *ant = NULL;
    Node *aux = lista->comeco;
    Node *novoNoh = (Node*) malloc(sizeof(Node));
    if (!novoNoh) {
        printf("Impossível adicionar. Lista cheia!\n");
    } else if (aux == NULL || aux->info > valor) {
        // LISTA VAZIA ou o valor a ser inserido
        // é menor que o primeiro
        novoNoh->info = valor;
        novoNoh->prox = lista->comeco;
        lista->comeco = novoNoh;
    } else { // LISTA NÃO VAZIA
        novoNoh->info = valor;
        while (aux != NULL && aux->info < valor) {
            ant = aux;
            aux = aux->prox;
        }
        novoNoh->prox = ant->prox;
        ant->prox = novoNoh;
    }
}
```

Listas Encadeada

- Remover
  - Antes de remover, verificamos se nossa lista é vazia. Se ela for, não é possível remover e retornamos FALSE (0).
  - Caso a lista não seja vazia, usaremos a mesma ideia das variáveis *ant* e *atual* (*aux* na inserção) para percorrermos a lista.

```
int removeDaLista(Lista *lista, int valor) {
    if (lista->comeco == NULL) {
        printf("Lista vazia. Impossível remover!\n");
        return FALSE;
    }
    printf("Removendo valor %d da lista\n", valor);
    Node *ant = NULL;
    Node *atual = lista->comeco;
    if (atual->info == valor) {
        lista->comeco = atual->prox;
        free(atual);
        return TRUE;
    }
    while (atual != NULL) {
        if (atual->info == valor) {
            ant->prox = atual->prox;
            free(atual);
            return TRUE;
        }
        ant = atual;
        atual = atual->prox;
    }
    return FALSE;
}
```

Listas Encadeada

- Remover (*cont.*)
  - Se o valor a ser removido for o primeiro da lista, definimos o próximo item como o começo da lista e retornamos TRUE (1).
  - Caso contrário, vamos percorrer a nossa lista até encontrar o elemento a ser removido. Quando encontrar, vamos "ligar" a lista (ant e prox) e vamos retornar TRUE (1).

```c
int removeDaLista(Lista *lista, int valor) {
    if (lista->comeco == NULL) {
        printf("Lista vazia. Impossível remover!\n");
        return FALSE;
    }
    printf("Removendo valor %d da lista\n", valor);
    Node *ant = NULL;
    Node *atual = lista->comeco;
    if (atual->info == valor) {
        lista->comeco = atual->prox;
        free(atual);
        return TRUE;
    }
    while (atual != NULL) {
        if (atual->info == valor) {
            ant->prox = atual->prox;
            free(atual);
            return TRUE;
        }
        ant = atual;
        atual = atual->prox;
    }
    return FALSE;
}
```

Listas Encadeada

- Remover (*cont.*)
  - Caso o item a ser removido não exista na lista, retornaremos FALSE (0).

```c
int removeDaLista(Lista *lista, int valor) {
    if (lista->comeco == NULL) {
        printf("Lista vazia. Impossível remover!\n");
        return FALSE;
    }
    printf("Removendo valor %d da lista\n", valor);
    Node *ant = NULL;
    Node *atual = lista->comeco;
    if (atual->info == valor) {
        lista->comeco = atual->prox;
        free(atual);
        return TRUE;
    }
    while (atual != NULL) {
        if (atual->info == valor) {
            ant->prox = atual->prox;
            free(atual);
            return TRUE;
        }
        ant = atual;
        atual = atual->prox;
    }
    return FALSE;
}
```

Listas Encadeada

```c
int procuraElemento(Lista *lista, int valor) {
    Node *atual = lista->comeco;
    while (atual != NULL) {
        if (atual->info == valor) {
            return TRUE;
        }
        atual = atual->prox;
    }
    return FALSE;
}
```

- Busca
  - Usa uma variável auxiliar chamada atual para percorrer a lista.
  - Enquanto percorre a lista, verifica se o elemento atual possui o valor que estamos procurando. Caso encontre, retornamos TRUE (1).
  - Caso a lista chegue ao fim e não encontramos o valor, retornamos FALSE (0).

Listas Encadeada

- **Vantagens:**
  i. Flexibilidade de tamanho
     - A Lista pode crescer e encolher conforme necessário, sem se preocupar com a capacidade inicial ou a realocação constante. Isso elimina o risco de ter um tamanho fixo e proporciona um uso mais eficiente da memória. Cada novo elemento é alocado de forma dinâmica conforme a necessidade.

  ii. Eficiência de uso de memória
     - Ao usar alocação dinâmica de memória, a memória é alocada exatamente para o número de elementos armazenados, sem desperdício. Não é necessário reservar espaço extra como em vetores dinâmicos ou arrays de tamanho fixo.

Listas Encadeada

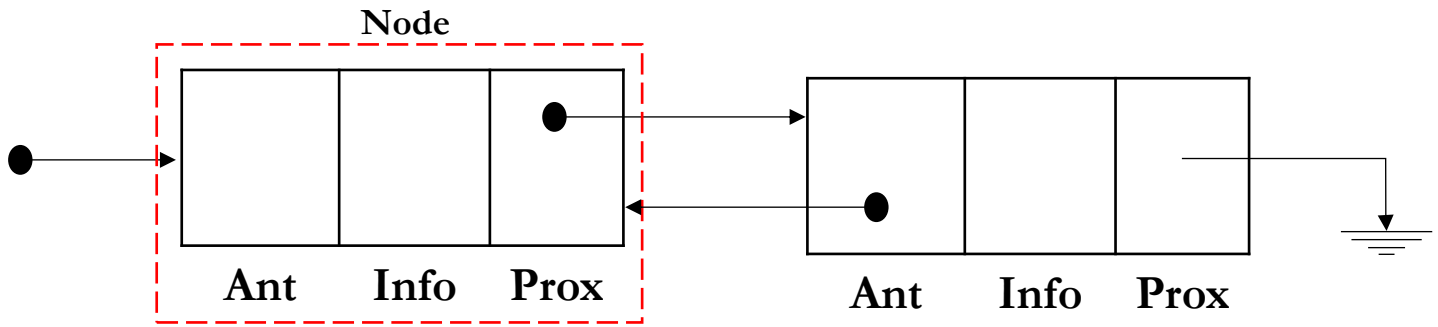
- **Desvantagens:**
  i. Acesso Sequencial
     - Para acessar um elemento em uma posição específica, é necessário percorrer a lista desde o início (ou de algum ponto conhecido) até o elemento desejado. Isso resulta em uma complexidade de tempo $O(n)$ para a busca, onde n é o número de elementos na lista. Em comparação, arrays oferecem acesso direto a qualquer posição em $O(1)$.

  ii. Fragmentação de memória
     - A memória pode ser fragmentada ao longo do tempo, o que pode tornar o uso ineficiente. Isso é especialmente importante em sistemas com recursos limitados, como sistemas embarcados, onde a fragmentação pode levar a um uso não ideal da memória.

Listas Encadeada

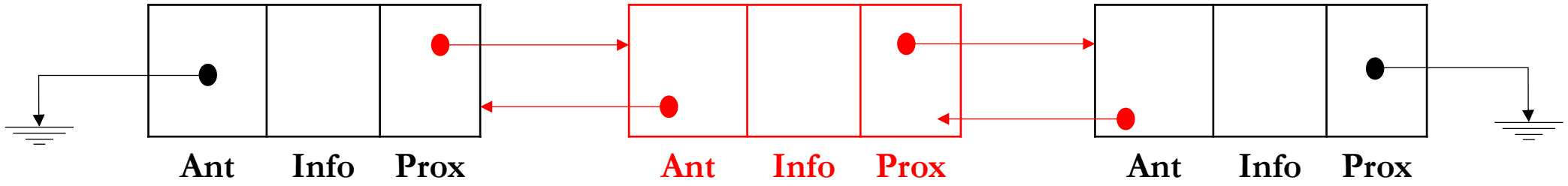
- **Desvantagens** (cont.):
  - iii. Gerenciamento de memória mais complexo
    - É necessário garantir que a memória seja liberada corretamente após o uso para evitar vazamentos de memória. A alocação e desalocação dinâmica de memória podem ser mais lentas do que simplesmente manipular um vetor estático.
  - iv. Ponteiro extra para manipulação
    - Para manipular a lista (inserir, remover, percorrer), é necessário manter e atualizar ponteiros de forma cuidadosa, o que pode resultar em maior complexidade de implementação e aumentar o risco de erros, como ponteiros nulos ou inválidos.

Listas Duplamente Encadeada



- Para a lista duplamente encadeada, vamos acrescentar um novo campo na nossa estrutura Node. Assim, teremos os seguintes campos:
  - **Info**, que armazena o conteúdo de cada item da nossa Lista.
  - **Prox**, que armazena o endereço de memória do próximo item da nossa Lista.
  - **Ant**, que armazena o endereço de memória do item anterior da nossa Lista.

Listas Duplamente Encadeada



- Utilizando esta estrutura, a cada novo item a ser inserido, basta criar outro Node e realizar as atribuições para que este Node ocupe seu lugar na lista (modificando os valores de **ant** e **prox**)!
  - Seja no começo, meio ou fim, a depender do elemento a ser inserido.
  - Lembrando que estamos inserindo os itens de forma ordenada na lista ☺

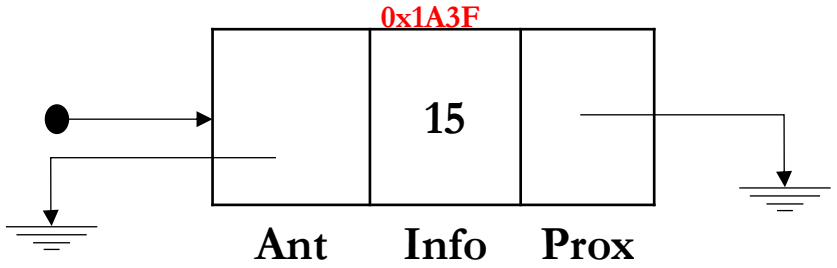
Listas Duplamente Encadeada

t_0 **NULL**

Começo: NULL

Ação: Inserir o item 15

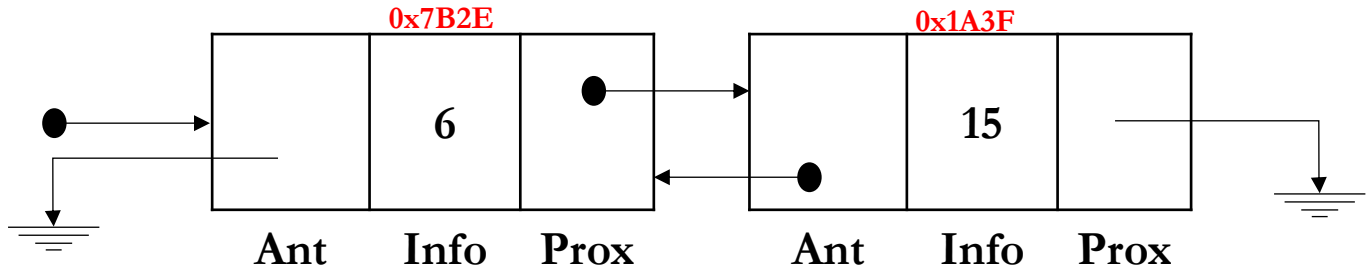
t_1



Começo: **0x1A3F**

Ação: Inserir o item 6

t_2

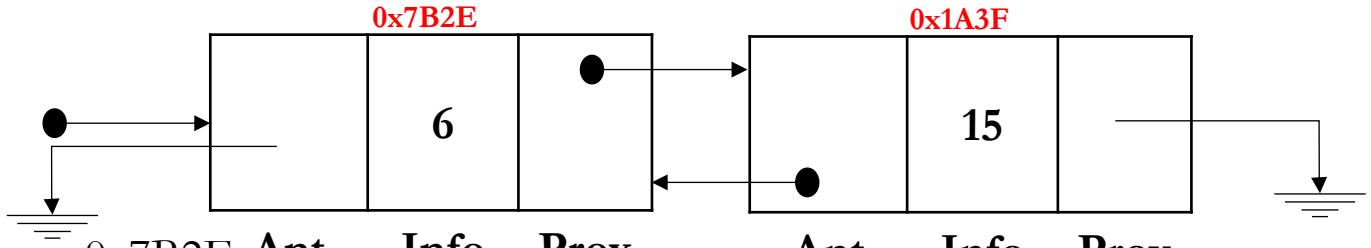


Começo: **0x7B2E**

Listas Duplamente Encadeada

Ação: Buscar o elemento 15 (retorna 0x1A3F sem mexer na lista)

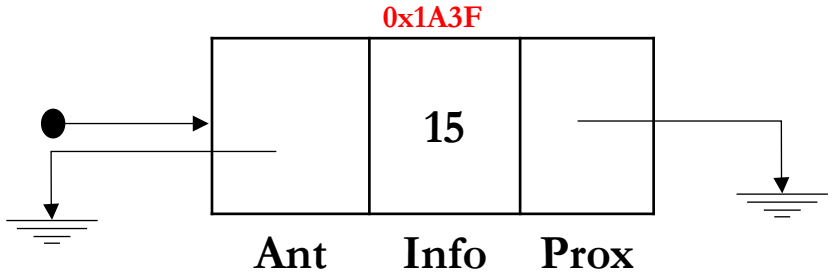
t_3



Começo: 0x7B2E

Ação: Remover o item 6 (Retorna 0x7B2E)

t_4



Começo: **0x1A3F**

Ação: Remover o item 15 (Retorna **0x1A3F**)

t_5 **NULL**

Começo: **NULL**

Listas Duplamente Encadeada

- Estrutura

```c
#define TRUE 1
#define FALSE 0

typedef struct no {
    struct no *ant;
    int info;
    struct no *prox;
} Node;

typedef struct {
    struct no *comeco;
    struct no *fim;
} Lista;
```

  - Para criar a estrutura, precisamos criar dois tipos novos: Node e Lista.
  - O Node possui três atributos: info (do tipo dos elementos da Lista) e o prox (do tipo *no*) e ant (do tipo *no*)
  - A Lista possui dois *no* iniciais, chamado *comeco* e *fim*.

Listas Duplamente Encadeada

- Inserir
  - Antes de inserir um elemento, primeiramente tentaremos alocar memória para ele. Caso não tenha memória suficiente, retornaremos uma mensagem de erro.
  - O primeiro caso de inserção é quando a lista é vazia. Se for, vamos atribuir os valores ao info, ant e prox e definir o *novoNoh* como começo e fim da lista.

```
void insere(Lista *lista, int valor) {
    Node *novoNoh = (Node*) malloc(sizeof(Node));
    if (!novoNoh) {
        printf("Impossível inserir. Lista cheia!");
    } else if (lista->comeco == NULL) {
        novoNoh->ant = NULL;
        novoNoh->info = valor;
        novoNoh->prox = NULL;
        lista->comeco = novoNoh;
        lista->fim = novoNoh;
    } else if (lista->comeco->info > valor) {
        novoNoh->ant = NULL;
        novoNoh->info = valor;
        novoNoh->prox = lista->comeco;
        lista->comeco->ant = novoNoh;
        lista->comeco = novoNoh;
    } else {
        novoNoh->info = valor;
        Node *aux = lista->comeco;
        while (aux->prox != NULL && aux->prox->info < valor) {
            aux = aux->prox;
        }
        novoNoh->prox = aux->prox;
        if (aux->prox != NULL) {
            novoNoh->prox->ant = novoNoh;
        } else {
            lista->fim = novoNoh;
        }
        novoNoh->ant = aux;
        aux->prox = novoNoh;
    }
}
```

Listas Duplamente Encadeada

- Inserir (*cont.*)
  - O segundo caso é quando o novo item é menor que o primeiro elemento. Nesse caso, vamos definir o valor de info, prox será setado para o antigo começo, e o antigo começo receberá o novoNoh como ant.
  - Não mexemos no fim, então não precisamos atualizá-lo.

```c
void insere(Lista *lista, int valor) {
    Node *novoNoh = (Node*) malloc(sizeof(Node));
    if (!novoNoh) {
        printf("Impossível inserir. Lista cheia!");
    } else if (lista->comeco == NULL) {
        novoNoh->ant = NULL;
        novoNoh->info = valor;
        novoNoh->prox = NULL;
        lista->comeco = novoNoh;
        lista->fim = novoNoh;
    } else if (lista->comeco->info > valor) {
        novoNoh->ant = NULL;
        novoNoh->info = valor;
        novoNoh->prox = lista->comeco;
        lista->comeco->ant = novoNoh;
        lista->comeco = novoNoh;
    } else {
        novoNoh->info = valor;
        Node *aux = lista->comeco;
        while (aux->prox != NULL && aux->prox->info < valor) {
            aux = aux->prox;
        }
        novoNoh->prox = aux->prox;
        if (aux->prox != NULL) {
            novoNoh->prox->ant = novoNoh;
        } else {
            lista->fim = novoNoh;
        }
        novoNoh->ant = aux;
        aux->prox = novoNoh;
    }
}
```

Listas Duplamente Encadeada

- Inserir (*cont.*)
  - O último caso é inserção no meio ou fim da lista. Para isso, precisamos percorrer a lista até encontrar onde o item será inserido. Caso a inserção seja no meio da lista, além de atualizar o prox do anterior, também atualizaremos o ant do próximo como sendo o novoNoh (não necessário se for o último).

```
void insere(Lista *lista, int valor) {
    Node *novoNoh = (Node*) malloc(sizeof(Node));
    if (!novoNoh) {
        printf("Impossível inserir. Lista cheia!");
    } else if (lista->comeco == NULL) {
        novoNoh->ant = NULL;
        novoNoh->info = valor;
        novoNoh->prox = NULL;
        lista->comeco = novoNoh;
        lista->fim = novoNoh;
    } else if (lista->comeco->info > valor) {
        novoNoh->ant = NULL;
        novoNoh->info = valor;
        novoNoh->prox = lista->comeco;
        lista->comeco->ant = novoNoh;
        lista->comeco = novoNoh;
    } else {
        novoNoh->info = valor;
        Node *aux = lista->comeco;
        while (aux->prox != NULL && aux->prox->info < valor) {
            aux = aux->prox;
        }
        novoNoh->prox = aux->prox;
        if (aux->prox != NULL) {
            novoNoh->prox->ant = novoNoh;
        } else {
            lista->fim = novoNoh;
        }
        novoNoh->ant = aux;
        aux->prox = novoNoh;
    }
}
```

Listas Duplamente Encadeada

- Remover
  - Antes de remover, verificamos se nossa lista é vazia. Se ela for, não é possível remover e retornamos FALSE (0).
  - Caso contrário, vamos verificar se o item a ser removido é o começo. Se for, removeremos o item e definiremos o próximo como começo da lista (e seu ant como NULL). Caso a lista após remoção só tenha um elemento, também atualizaremos o fim.

```
int removeDaLista(Lista *lista, int valor) {
    if (lista->comeco == NULL) {
        printf("Lista vazia. Impossível remover!\n");
        return FALSE;
    }
    Node *atual = lista->comeco;
    if (atual->info == valor) {
        lista->comeco = atual->prox;
        if (lista->comeco != NULL) {
            lista->comeco->ant = NULL;
        } else {
            lista->fim = NULL;
        }
        free(atual);
        return TRUE;
    }
    while (atual != NULL) {
        if (atual->info == valor) {
            if (atual->ant != NULL) {
                atual->ant->prox = atual->prox;
            }
            if (atual->prox != NULL) {
                atual->prox->ant = atual->ant;
            }
            if (atual == lista->fim) {
                lista->fim = atual->ant;
            }
            free(atual);
            return TRUE;
        }
        atual = atual->prox;
    }

    return FALSE;
}
```

Listas Duplamente Encadeada

- Remover (*cont.*)
  - Por fim, vamos remover um item do meio ou fim da lista. Vamos percorrer a lista até encontrar onde vamos inserir o item. Se for no meio, precisaremos atualizar os nós anterior e próximo para se "ligarem" sem o nó que está sendo removido. Se for no fim, precisamos somente atualizar o nó anterior e atualizar o fim da lista.

```c
int removeDaLista(Lista *lista, int valor) {
    if (lista->comeco == NULL) {
        printf("Lista vazia. Impossível remover!\n");
        return FALSE;
    }
    Node *atual = lista->comeco;
    if (atual->info == valor) {
        lista->comeco = atual->prox;
        if (lista->comeco != NULL) {
            lista->comeco->ant = NULL;
        } else {
            lista->fim = NULL;
        }
        free(atual);
        return TRUE;
    }
    while (atual != NULL) {
        if (atual->info == valor) {
            if (atual->ant != NULL) {
                atual->ant->prox = atual->prox;
            }
            if (atual->prox != NULL) {
                atual->prox->ant = atual->ant;
            }
            if (atual == lista->fim) {
                lista->fim = atual->ant;
            }
            free(atual);
            return TRUE;
        }
        atual = atual->prox;
    }

    return FALSE;
}
```

Listas Duplamente Encadeada

- Remover (*cont.*)
  - Caso nenhum item seja encontrado com o valor passado como parâmetro, a função retorna FALSE (0).

```
int removeDaLista(Lista *lista, int valor) {
    if (lista->comeco == NULL) {
        printf("Lista vazia. Impossível remover!\n");
        return FALSE;
    }
    Node *atual = lista->comeco;
    if (atual->info == valor) {
        lista->comeco = atual->prox;
        if (lista->comeco != NULL) {
            lista->comeco->ant = NULL;
        } else {
            lista->fim = NULL;
        }
        free(atual);
        return TRUE;
    }
    while (atual != NULL) {
        if (atual->info == valor) {
            if (atual->ant != NULL) {
                atual->ant->prox = atual->prox;
            }
            if (atual->prox != NULL) {
                atual->prox->ant = atual->ant;
            }
            if (atual == lista->fim) {
                lista->fim = atual->ant;
            }
            free(atual);
            return TRUE;
        }
        atual = atual->prox;
    }

    return FALSE;
}
```

Listas Duplamente Encadeada

```c
int procuraElemento(Lista *lista, int valor) {
    Node *atual = lista->comeco;
    while (atual != NULL) {
        if (atual->info == valor) {
            return TRUE;
        }
        atual = atual->prox;
    }
    return FALSE;
}
```

- Busca
  - Mesma implementação da Lista Encadeada.

Listas Duplamente Encadeada

- **Vantagens:**
  i. Acesso bidirecional
     - É possível percorrer a lista em duas direções: tanto do início para o fim quanto do fim para o início. Isso é útil quando você precisa acessar elementos a partir de ambas as extremidades da lista ou realizar manipulações rápidas de ambos os lados

  ii. Eficiência em remoções e inserções no meio e fim
     - Se você já tem um ponteiro para o nó a ser manipulado, você pode acessar tanto o nó anterior quanto o nó seguinte diretamente.

Listas Duplamente Encadeada

- **Desvantagens:**
  i. Maior consumo de memória
     - Cada nó de uma lista duplamente encadeada requer **dois ponteiros**: um para o próximo nó e outro para o nó anterior. Isso significa que cada nó usa mais memória, especialmente em grandes listas.
  ii. Complexidade adicional na implementação:
     - Implementação mais complexa, porque é preciso manter e manipular dois ponteiros para cada nó. Isso torna a manipulação de inserções, remoções e navegação mais propensa a erros e mais difícil de implementar corretamente.

Listas Duplamente Encadeada

- **Desvantagens** (cont.):
  - iii. Mais operações para manter a consistência
    - Ao inserir ou remover nós, você precisa ajustar dois ponteiros: tanto o ponteiro do nó anterior quanto o ponteiro do nó seguinte. Se você esquecer de atualizar um desses ponteiros, pode causar referências inválidas ou *segmentation faults*.

Algoritmos e Estrutura de Dados II

Prof. Fellipe Guilherme Rey de Souza

Aula 07 – Lista (Implementação)