

# **RENATA CARINA SOARES**

## **ESTRUTURA DE DADOS**

ORIENTADOR:  
**PROFº DRº DANILO SIPOLI SANCHES**

**2015**

Dedico este trabalho primeiramente a **Deus**, ao meu pai **Osvaldo Sebastião Soares**, a minha mãe **Rosinha Leni de Jesuz Soares**, ao meu irmão **Renan Francisco Soares** e ao meu namorado **Willian Cardoso Horikoshi**. *Agradeço* ao professor orientador **Danilo Sipoli Sanches**.

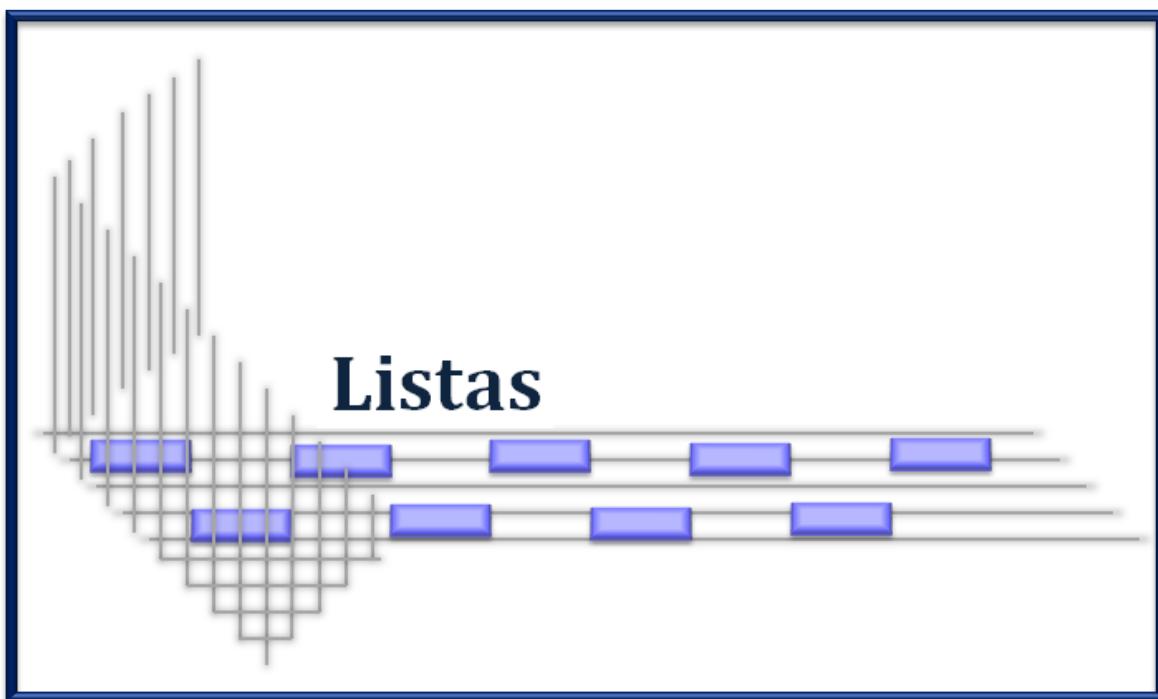
## SUMÁRIO

<b>1. LISTAS .....</b>	<b>1</b>
<b>1.1. Listas Simplesmente Encadeadas .....</b>	<b>3</b>
<b>1.1.1. <i>Conceitos</i>.....</b>	<b>3</b>
<b>1.1.2. <i>Criação e Manipulação</i> .....</b>	<b>4</b>
<b>1.1.2.1. Criação de um tipo abstrato de dado (TAD) .....</b>	<b>4</b>
<b>1.1.2.2. Criar a Lista .....</b>	<b>5</b>
<b>1.1.2.3. Inserir no início da lista.....</b>	<b>6</b>
<b>1.1.2.4. Averiguar se a lista está vazia .....</b>	<b>11</b>
<b>1.1.2.5. Imprimir os elementos.....</b>	<b>11</b>
<b>1.1.2.6. Buscar um determinado elemento na lista.....</b>	<b>14</b>
<b>1.1.2.7. Liberar todos os elementos.....</b>	<b>17</b>
<b>1.1.2.8. Retirar um nó.....</b>	<b>21</b>
<b>1.2. Listas Duplamente Encadeadas .....</b>	<b>33</b>
<b>1.2.1. <i>Conceitos</i>.....</b>	<b>33</b>
<b>1.2.2. <i>Criação e Manipulação</i> .....</b>	<b>34</b>
<b>1.2.2.1. Criação de um tipo abstrato de dado (TAD) .....</b>	<b>35</b>
<b>1.2.2.2. Criar a lista .....</b>	<b>35</b>
<b>1.2.2.3. Inserir no início da lista.....</b>	<b>36</b>
<b>1.2.2.4. Averiguar se a lista está vazia .....</b>	<b>42</b>
<b>1.2.2.5. Imprimir os elementos.....</b>	<b>43</b>
<b>1.2.2.6. Buscar um determinado elemento na lista.....</b>	<b>46</b>
<b>1.2.2.7. Liberar todos os elementos.....</b>	<b>48</b>
<b>1.2.2.8. Retirar um nó.....</b>	<b>52</b>
<b>1.3. Lista Circular .....</b>	<b>61</b>
<b>1.3.1. <i>Conceitos</i>.....</b>	<b>61</b>
<b>1.3.1.1. Vantagens e desvantagens .....</b>	<b>61</b>
<b>1.3.2. <i>Manipulação</i> .....</b>	<b>62</b>
<b>1.4. Exercícios .....</b>	<b>63</b>
<b>2. FILAS .....</b>	<b>65</b>
<b>2.1. Fila com Lista Simplesmente Encadeada .....</b>	<b>67</b>
<b>2.1.1. <i>Conceitos</i>.....</b>	<b>67</b>
<b>2.1.2. <i>Criação e Manipulação</i> .....</b>	<b>68</b>

2.1.2.1.	Criação de um tipo abstrato de dado (TAD) .....	68
2.1.2.2.	Criar a Fila.....	69
2.1.2.3.	Inserir elementos na fila .....	70
2.1.2.4.	Averiguar se a lista está vazia .....	76
2.1.2.5.	Imprimir os elementos.....	77
2.1.2.6.	Buscar um determinado elemento na fila .....	80
2.1.2.7.	Liberar todos os elementos.....	82
2.1.2.8.	Retirar um nó.....	86
2.2.	<b>Fila com Lista Duplamente Encadeada</b> .....	92
2.2.1.	<i>Conceitos</i> .....	92
2.2.2.	<i>Criação e Manipulação</i> .....	93
2.2.2.1.	Criação de um tipo abstrato de dado (TAD) .....	93
2.2.2.2.	Criar a Fila.....	94
2.2.2.3.	Inserir elementos .....	95
2.2.2.3.1.	<i>Inserir no fim da Fila Dupla</i> .....	96
2.2.2.3.2.	<i>Inserir no início da Fila Dupla</i> .....	102
2.2.2.4.	Averiguar se a lista está vazia .....	107
2.2.2.5.	Imprimir os elementos.....	107
2.2.2.6.	Buscar um determinado elemento na fila .....	110
2.2.2.7.	Liberar todos os elementos.....	114
2.2.2.8.	Retirar elemento .....	118
2.2.2.8.1.	<i>Retirar do início da Fila Dupla</i> .....	118
2.2.2.8.2.	<i>Retirar do final da Fila Dupla</i> .....	122
2.3.	<b>Fila Circular</b> .....	126
2.3.1.	<i>Conceitos</i> .....	126
2.3.2.	<i>Manipulação</i> .....	126
2.4.	<b>Exercícios</b> .....	128
3.	<b>PILHA</b> .....	129
3.1.	<b>Conceitos</b> .....	129
3.2.	<b>Criação e Manipulação</b> .....	130
3.2.1.	<i>Criação de um tipo abstrato de dado (TAD)</i> .....	131
3.2.2.	<i>Criar a pilha</i> .....	132
3.2.3.	<i>Inserir na pilha</i> .....	133
3.2.4.	<i>Averiguar se a pilha está vazia</i> .....	136

3.2.5. <i>Imprimir os elementos</i> .....	137
3.2.6. <i>Buscar um determinado elemento na pilha</i> .....	140
3.2.7. <i>Liberar todos os elementos</i> .....	143
3.2.8. <i>Retirar um nó</i> .....	147
3.1. <b>Exercícios</b> .....	152
<b>4. ÁRVORES</b> .....	153
4.1. <b>Árvore Binária</b> .....	157
4.1.1. <i>Conceitos</i> .....	157
4.1.2. <i>Criação e Manipulação</i> .....	157
4.1.2.1. Criação de um tipo abstrato de dado (TAD) .....	157
4.1.2.2. Criar a árvore binária.....	158
4.1.2.3. Inserir elementos .....	159
4.1.2.4. Averiguar se a árvore binária está vazia.....	167
4.1.2.5. Imprimir os elementos.....	168
4.1.2.6. Verificar se um determinado elemento pertence a árvore .....	186
4.1.2.6. Libear todos os elementos.....	190
4.2. <b>Árvore binária de busca</b> .....	203
4.2.1. <i>Conceitos</i> .....	203
4.2.2. <i>Criação e Manipulação</i> .....	203
4.2.2.1. Criação de um tipo abstrato de dado (TAD) .....	203
4.2.2.2. Criar a árvore binária de busca.....	204
4.2.2.3. Inserir elementos .....	205
4.2.2.4. Averiguar se a árvore binária de busca está vazia.....	208
4.2.2.5. Imprimir os elementos.....	209
4.2.2.6. Verificar se um determinado elemento pertence a árvore .....	217
4.2.2.7. Libear todos os elementos.....	220
4.2.2.8. Retirar um elemento .....	228
4.3. <b>Árvore AVL</b> .....	234
4.3.2. <i>Conceitos</i> .....	234
4.3.2.1. Fator de balanceamento.....	234
4.3.2.2. Tipos de rotações.....	234
4.3.2.2.1. <i>Exemplo</i> .....	235
4.4. <b>Árvore B</b> .....	248
4.4.2. <i>Conceitos</i> .....	248

4.4.2.1.	Inserção de dados (chaves).....	248
4.4.2.1.1.	Exemplo .....	250
4.5.	<b>Exercícios</b> .....	255
	<b>Referências</b> .....	257



## 1. LISTAS

Toda lista é formada por um conjunto de elementos, denominados de nós da lista. Estes são ligados entre si, porém, a conexão entre os elementos se difere conforme a classificação da lista, a qual possui três configurações:

- **Lista simplesmente encadeada.** Seus elementos estão dispostos da *esquerda para direita*. Esta direção é determinada pelos campos do endereço seguinte, os quais são ponteiros, que partem de um nó para seu sucessor na lista, como mostra a figura 1.

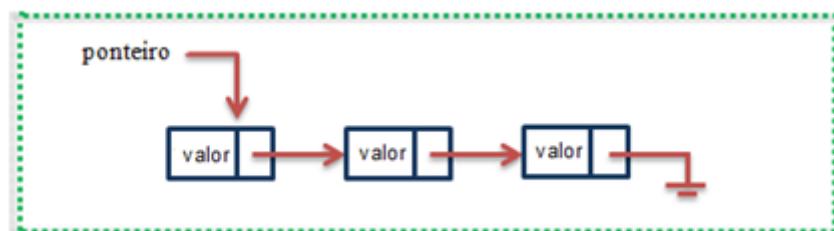


Figura 1. Representação de uma lista simplesmente encadeada.

- **Lista duplamente encadeada.** Os elementos podem ser manipulados em duas direções, tanto da *esquerda para direita* quanto da *direita para esquerda*. Estas direções são determinadas pelos campos do endereço seguinte e anterior, os quais são ponteiros, que partem de um nó para seu sucessor e outro para seu anterior na lista dupla, respectivamente, como mostra a figura 2.

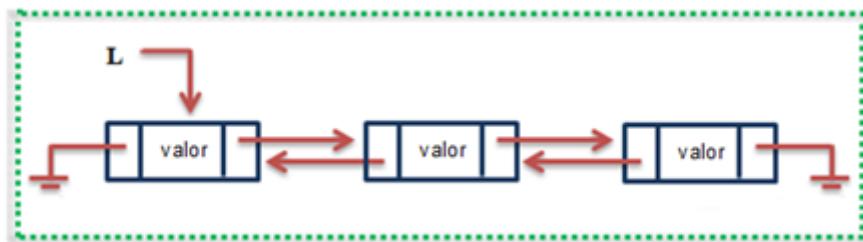


Figura 2. Representação de uma lista duplamente encadeada.

- **Lista circular.** O último elemento tem como próximo o primeiro nó da lista, formando um ciclo, como mostra a figura 3. A lista pode ser representada por um ponteiro para um elemento inicial qualquer da lista.

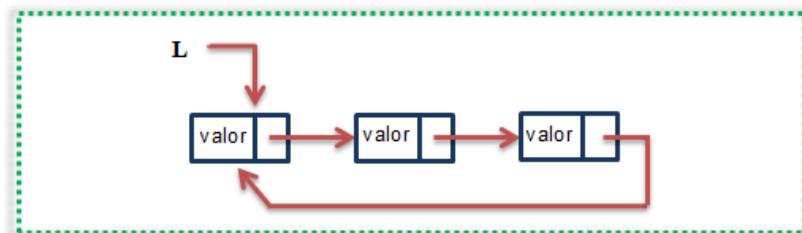


Figura 3. Representação de uma lista circular.

## 1.1. Listas Simplesmente Encadeadas

### 1.1.1. Conceitos

Uma lista simplesmente encadeada é um conjunto de elementos, denominados de nós da lista. Cada nó contém dois campos, como mostra a figura 4:

- *campo de informação*: armazena o real elemento da lista.
- *campo do endereço seguinte*: usado para acessar o próximo nó. Este campo dentro de um nó pode ser visto como uma ligação ou um ponteiro para o próximo nó.

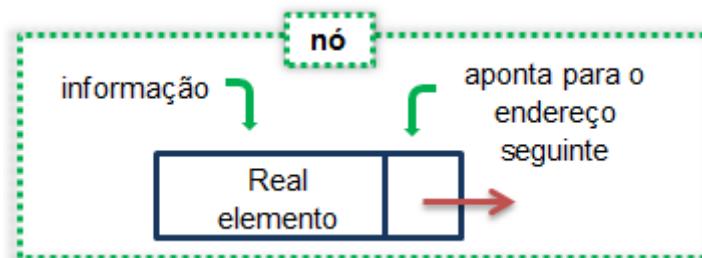


Figura 4. Composição de um nó da lista.

A lista inteira é acessada a partir de um ponteiro externo denominado **L**, este aponta para o primeiro nó da lista.

O *campo do endereço seguinte* do último nó da lista contém um valor especial, conhecido como NULL, que não é um endereço válido. Esse ponteiro nulo (NULL) é usado para indicar o final de uma lista. Como apresentado na figura 5.

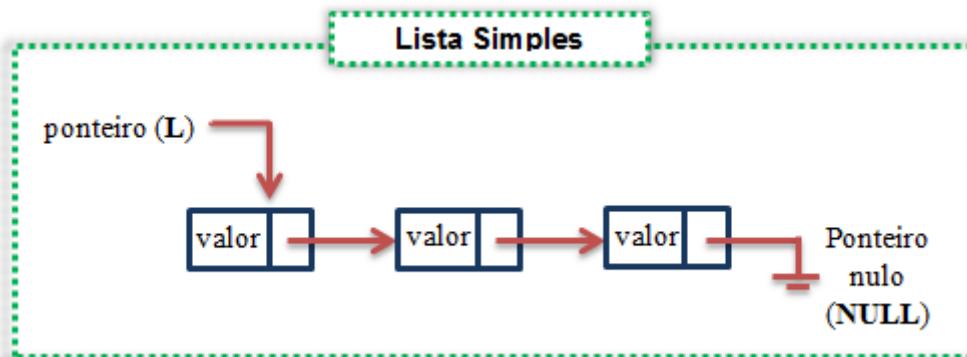


Figura 5. Representação de uma lista simplesmente encadeada.

#### Observação

Por ponteiro “externo” entende-se aquele que não está incluso dentro de um nó e, sim, seu valor pode ser acessado diretamente por referência a uma variável.

A lista *sem* nós é chamada *lista vazia* ou *lista nula* (figura 6). O valor do ponteiro externo, para este tipo de lista, é o ponteiro nulo.

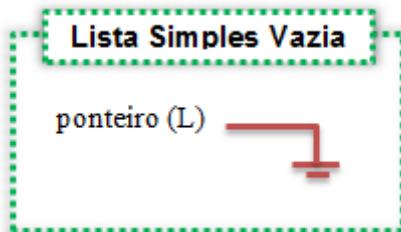


Figura 6. Representação de uma lista simples vazia.

Uma lista simplesmente encadeada mantém seus elementos em uma dada direção, como visto na figura 5, da *esquerda para direita*. Esta direção é determinada pelos campos do endereço seguinte, os quais são ponteiros, que partem de um nó para seu sucessor na lista.

### 1.1.2. Criação e Manipulação

Pode-se programar a criação da lista simples e estabelecer as devidas funcionalidades a ela, entre estas se tem:

- 1.1.2.1. Criação de um Tipo Abstrato de Dado (TAD).
- 1.1.2.2. Criar a lista.
- 1.1.2.3. Inserir elementos no início da lista.
- 1.1.2.4. Averiguar se a lista está vazia.
- 1.1.2.5. Imprimir os elementos.
- 1.1.2.6. Buscar um determinado elemento na lista.
- 1.1.2.7. Liberar todos os elementos.
- 1.1.2.8. Retirar um nó.

#### 1.1.2.1. Criação de um tipo abstrato de dado (TAD)

Criação de um novo tipo de dado, como mostra a figura 7.

```

    graph TD
        A["typedef struct lista{"] --- B[int info]
        B --- C["struct lista* prox;"]
        C --- D["}ListaSimp;"]
    
```

Figura 7. Estrutura de um novo tipo de dado do tipo ListaSimp.

*Descrição das funcionalidades* apresentadas na figura 7:

1. Declarou-se a informação (info) do *campo de informação*, do tipo inteiro, o qual armazena o valor contido no nó. (figura 8).
2. Declarou-se o ponteiro (prox) do *campo do endereço seguinte*, sendo do tipo struct lista (figura 8).



Figura 8. Regiões declaradas como info e outra como prox.

#### Observação

Para facilitar, com a utilização do `typedef`, em vez de usar a designação `struct lista` troca-se por `ListaSimp`. Entretanto, como visto no marcador 2 não se pode declarar o ponteiro prox do tipo `ListaSimp`, pois este ainda não havia sido referenciado.

#### 1.1.2.2. Criar a Lista

##### O que fazer? Dica...

Toda lista deve inicializar vazia!!!

A função denominada **criar\_lista**, apresentada na figura 9, possui as seguintes características (graficamente, está função está representada na figura 6, apresentada anteriormente):

- A função não possui parâmetros;
- O retorno é do tipo **ListaSimp**, retornando o valor nulo (NULL).

```
ListaSimp* criar_lista(void){
    return NULL;
}
```

Figura 9. Função referente à atribuição NULL para a lista simples.

**Observação**

Na função principal (**main()**) a função criar a lista deve ser chamada como:

**L = criar\_lista();**

#### 1.1.2.3. Inserir no início da lista

**O que fazer? Dicas...**

- Deve alocar memória, de maneira dinâmica, para cada nó a ser inserido.
- O ponteiro L, aponta sempre para o primeiro nó da lista.
- A cada inserção o campo do endereço seguinte deste apontará para onde o ponteiro L aponta e, posteriormente, o ponteiro L apontará para o nó a ser inserido.

A função denominada **inserir\_elementos**, apresentada na figura 10, possui as seguintes características:

- A função possui como parâmetros o ponteiro L, do tipo **ListaSimp**, que aponta para o primeiro nó da lista, e o valor a ser inserido, este do tipo inteiro, no campo de informação.
- O retorno da função é do tipo **ListaSimp**, retornando o novo nó, assim, encadeando o elemento na lista existente.

```
ListaSimp* inserir_elementos(ListaSimp* L, int valor){
```

```
1      ListaSimp* novo = (ListaSimp*)malloc(sizeof(ListaSimp));
2      novo -> info = valor;
3      novo -> prox = L;
4      return novo;
    }
```

Figura 10. Função referente à inserir elementos no início da lista simples.

**Descrição das funcionalidades** apresentadas figura 10:

1. Utilizando o conceito de alocação dinâmica, aloca-se um nó denominado de *novo* do tipo **ListaSimp**.

2. O *novo -> info* recebe o valor, do tipo inteiro, fornecido por meio do parâmetro da função.
3. O *novo->prox* aponta para onde o ponteiro L aponta, se for para NULL apontará para NULL, senão, apontará para o nó designado como primeiro da lista.
4. Retorna o novo nó criado para o ponteiro L, assim, este aponta para o novo nó.

O ponteiro novo sendo do tipo ListaSimp tem acesso as regiões definidas em um nó. Assim, para referenciar o campo de informação do novo nó escreve-se *novo->info*, assim como, para o campo do endereço seguinte, o qual designa-se como *novo->prox*. Para melhor entendimento, graficamente, apresenta-se a figura 11.

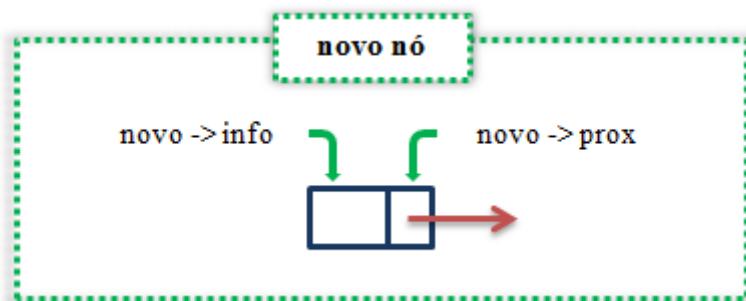


Figura 11. Regiões do nó denominado de novo.

Na função principal (**main()**) deve constar (figura 12):

```
main(void){
    1   ListoSimp* L;
    2   L = criar_lista();
    3   L = inserir_elementos(L, <valor a ser inserido>);
}
```

Figura 12. Função principal chamar a função criar\_lista e inserir\_elementos na lista simples.

**Descrição das funcionalidades** apresentadas na figura 12:

1. Cria-se um ponteiro L, do tipo ListaSimp, o qual referenciará o início da lista.
2. Chamar a função *criar\_lista* para atribuir o primeiro valor a lista, o qual é nulo. O ponteiro L declarado recebe o retorno da função *criar\_lista*.
3. Chamar a função *inserir\_elementos* passando como parâmetros o ponteiro L (apontando para o primeiro nó, caso houver, ou para NULL caso a lista esteja vazia) e o valor a ser inserido. O ponteiro L declarado recebe o retorno da função *inserir\_elementos*, o qual é um novo nó.

Para melhor entendimento, graficamente, realiza-se um exemplo a seguir, utilizando o conceito *teste de mesa*.

### Exemplo

Inserir os seguintes elementos 4, 2, e 7.

**1º PASSO:** Na função principal colocar os seguintes comandos apresentados na figura 13:

```
main(void){
    ListaSimp* L;
    L = criar_lista();
    L = inserir_elementos(L, 4);
    L = inserir_elementos(L, 2);
    L = inserir_elementos(L, 7);
}
```

Figura 13. Comandos que devem ser colocados na função principal.

**2º PASSO:** Quando inicializada a função principal o ponteiro L, tipo ListaSimp, é declarado.

**3º PASSO:** O ponteiro L recebe o retorno da função criar\_lista, o qual é NULL.



**4º PASSO:** a função inserir\_elementos(L, 4) realiza os seguintes procedimentos:

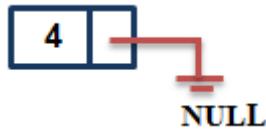
- Aloca-se um novo nó.



- Este nó recebe o valor 4 no campo de informação (novo -> info).



- O campo do endereço seguinte do nó criado (novo -> prox) aponta para onde o ponteiro L aponta (Como visto anteriormente, no passo 3, aponta para NULL).



- O ponteiro L, na função principal, recebe o novo nó, assim, deixa de apontar para NULL e aponta para o nó criado (figura 14).

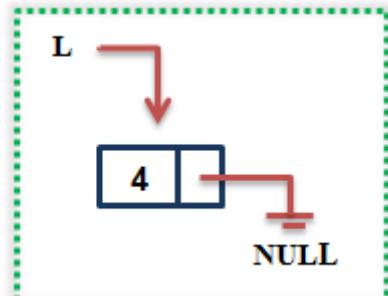


Figura 14. Inserção do elemento 4.

**5º PASSO:** a função inserir\_elementos(L, 2) realiza os seguintes procedimentos:

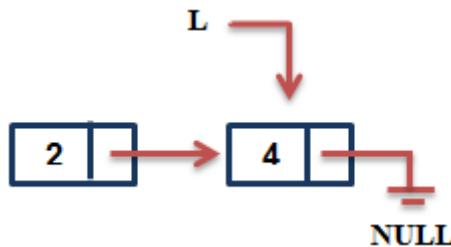
- Aloca-se um novo nó.



- Este nó recebe o valor 2 no campo de informação (*novo -> info*).



- O campo do endereço seguinte do nó criado (*novo -> prox*) aponta para onde o ponteiro L aponta (Como visto anteriormente, na figura 14, aponta para o nó de valor 4).



- O ponteiro L, na função principal, recebe o novo nó, assim, deixa de apontar para o nó de valor 4 e aponta para o novo nó criado de valor 2 (figura 15).

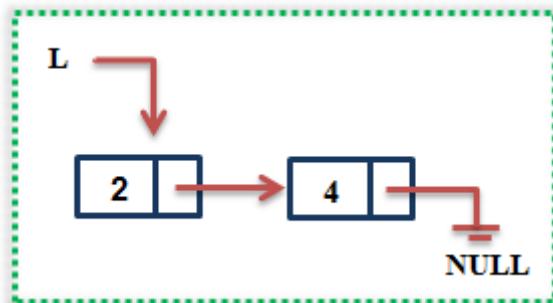


Figura 15. Inserção do elemento 2.

**6º PASSO:** a função inserir\_elementos(L, 7) realiza os seguintes procedimentos:

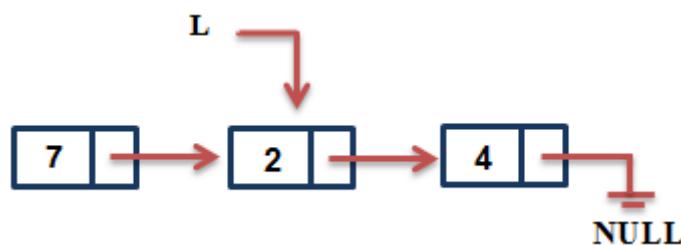
- Aloca-se um novo nó.



- Este nó recebe o valor 7 no campo de informação (novo -> info).



- O campo do endereço seguinte do nó criado (novo -> prox) aponta para onde o ponteiro L aponta (Como visto anteriormente, na figura 15, aponta para nó com valor 2).



- O ponteiro L, na função principal, recebe o novo nó, assim, deixa de apontar para o nó com valor 2 e aponta para o novo nó criado com valor 7 (figura 16).

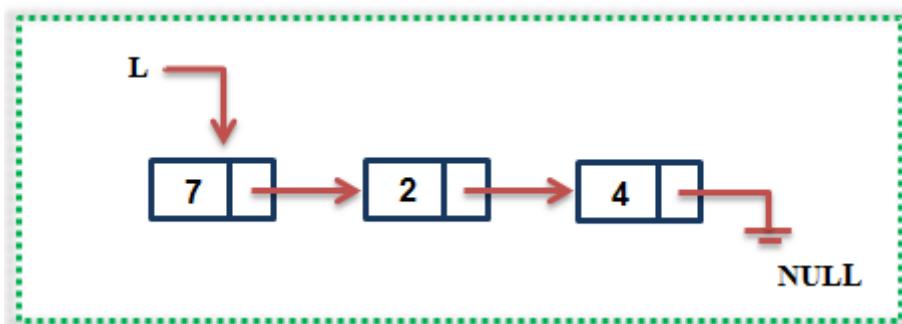


Figura 16. Inserção do elemento 7.

**Reflexão...**

Pode-se inserir no meio ou no fim, contudo, deve realizar modificações do código referente a inserir na lista.

O uso de desenhos auxilia na formação da lógica necessária para a formação do código.

**1.1.2.4. Averiguar se a lista está vazia****O que fazer? Dicas...**

Verificar se a lista aponta para o valor nulo (NULL).

A função denominada **lista\_vazia**, apresentada na figura 17, possui as seguintes características:

- A função possui como parâmetro o ponteiro L, o qual aponta para o primeiro nó da lista.
- O retorno é do tipo **booleano**, retornando valor 1 (verdadeiro) quando a lista estiver vazia, caso contrário, retorna valor 0 (falso);

```
int lista_vazia(ListaSimp* L){  
    return (L == NULL);  
}
```

Figura 17. Função que verifica se a lista está vazia.

**1.1.2.5. Imprimir os elementos****O que fazer? Dicas...**

- Percorrer a lista imprimindo os valores de cada nó.
- Não manipular o ponteiro L para não perder a referência do início da lista.

A função denominada **imprimir**, apresentada na figura 18, possui as seguintes características:

- A função possui como parâmetro o ponteiro L, o qual aponta para o primeiro nó da lista.
- O retorno é do tipo void, retornando na tela os valores contidos na lista.

```

void imprimir(ListaSimp* L){

    1      ListoSimp* p;

    2      if(lista_vazia(L))
    2          printf("\nLista vazia!\n\n");

    3      else{
    3          for (p = L; p != NULL; p = p -> prox)
    3              printf(" %d", p -> info);
    3
    3      }
    3      printf("\n\n");
    3
}

```

Figura 18. Função imprimir.

*Descrição das funcionalidades* apresentadas na figura 18:

1. Declara-se um ponteiro auxiliar denominado de p do tipo ListaSimp.
2. Dentro da condicional chama-se a função `lista_vazia`, caso retorne o valor 1 a lista está vazia e, assim, imprimi-se “Lista vazia!”.
3. Senão entra no laço for:
  - Utiliza-se um ponteiro auxiliar denominado de p, o qual aponta para onde o ponteiro L aponta, ou seja, para o primeiro nó.
  - Enquanto o ponteiro p for diferente de NULL ainda há elemento(s) na lista.
  - O incremento do ponteiro p é representado por  $p = p \rightarrow prox$ .
  - A impressão é baseada no valor armazenado na região do campo de informação (info).

#### Observação

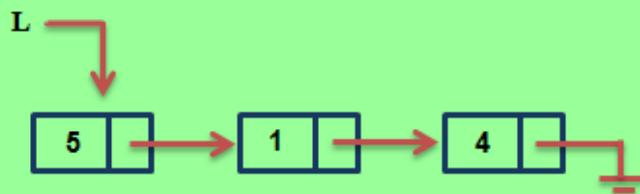
Na função principal (`main()`) a função imprimir a lista deve ser chamada como:

`imprimir(L);`

Para melhor entendimento, graficamente, realiza-se um exemplo a seguir, utilizando o conceito *teste de mesa*.

### Exemplo

Considerando a seguinte lista:



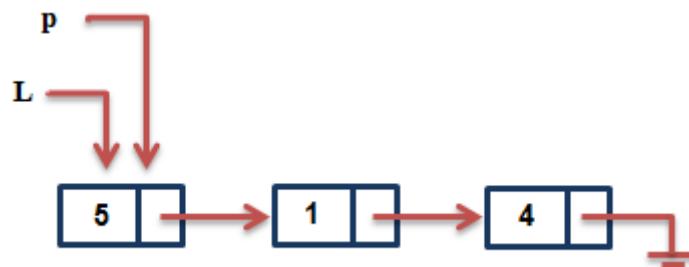
**1º PASSO:** Declara-se um ponteiro auxiliar denominado de p.

**2º PASSO:** Entra na condicional if.

- Chama-se a função lista\_vazia, a lista está vazia? NÃO, pois o ponteiro L aponta para o nó de valor 5, ou seja, é diferente de NULL.

**3º PASSO:** Entra na condicional else. Percorre-se o laço for.

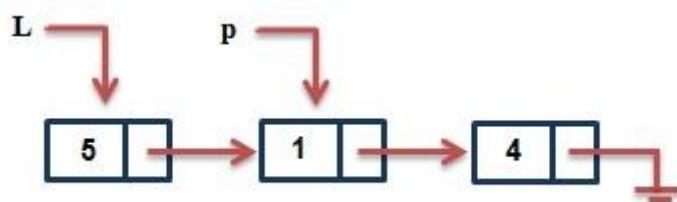
- O ponteiro p aponta para onde o ponteiro L aponta.



- O ponteiro p é diferente de NULL, ou seja, não aponta para NULL? SIM, assim imprime-se o valor 5 contido no elemento apontado pelo ponteiro p ( $p \rightarrow \text{info}$ ).

**4º PASSO:** Retorna ao laço for:

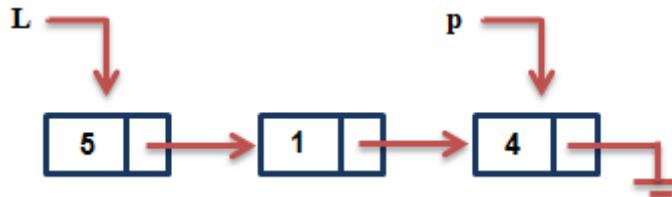
- O incremento do ponteiro p é dado por  $p = p \rightarrow \text{prox}$ , onde  $p \rightarrow \text{prox}$  é o elemento de valor 1. Assim, o ponteiro p aponta para este nó.



- O ponteiro  $p$  é diferente de NULL, ou seja, não aponta para NULL? SIM, assim imprime-se o valor 1 contido no elemento apontado pelo ponteiro  $p$  ( $p \rightarrow \text{info}$ ).

**5º PASSO:** Retorna ao laço for:

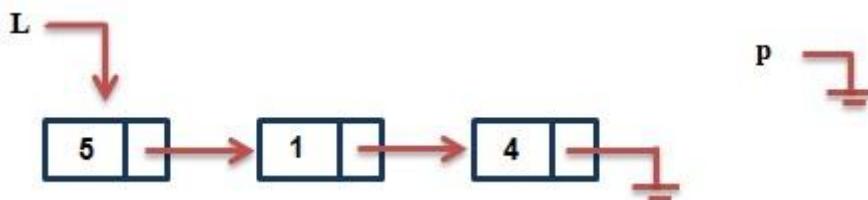
- O incremento do ponteiro  $p$  é dado por  $p = p \rightarrow \text{prox}$ , onde  $p \rightarrow \text{prox}$  é o elemento de valor 4. Assim, o ponteiro  $p$  aponta para este nó.



- O ponteiro  $p$  é diferente de NULL, ou seja, não aponta para NULL? SIM, assim imprime-se o valor 4 contido no elemento apontado pelo ponteiro  $p$  ( $p \rightarrow \text{info}$ ).

**6º PASSO:** Retorna ao laço for:

- O incremento do ponteiro  $p$  é dado por  $p = p \rightarrow \text{prox}$ , onde  $p \rightarrow \text{prox}$  possui valor nulo. Assim, o ponteiro  $p$  aponta para NULL.



- O ponteiro  $p$  é diferente de NULL, ou seja, não aponta para NULL? NÃO, ou seja, imprimiram-se todos os elementos contidos na lista.

**7º PASSO:** Termina o laço, assim, encerra a impressão!!

#### 1.1.2.6. Buscar um determinado elemento na lista

##### O que fazer? Dicas...

- Percorrer a lista comparando o valor que o nó possui com o valor a ser buscado.
- Caso encontre, retorna o nó que contém o valor. Caso contrário, retorna NULL.

A função denominada **buscar\_elemento**, apresentada na figura 19, possui as seguintes características:

- A função possui como parâmetro o ponteiro L, o qual aponta para o primeiro nó da lista, e o valor a ser procurado.
- O retorno é do tipo ListaSimp. Caso o elemento seja encontrado o ponteiro p aponta para ele, assim, retorna este nó, caso contrário, retorna o valor nulo (NULL).

```
ListaSimp* buscar_elemento (ListaSimp* L, int valor) {
```

```
    ListaSimp* p;
    for(p = L; p != NULL; p = p -> prox) {
        if(p -> info == valor)
            return p;
    }
    return NULL;
}
```

Figura 19. Função buscar elemento na Lista.

*Descrição das funcionalidades* apresentadas na figura 19:

1. O ponteiro L aponta para o primeiro nó da lista, este não deve ser manipulado para não perder a referência, então, utiliza-se um ponteiro auxiliar denominado de p, o qual aponta para onde L aponta, ou seja, para o primeiro nó.
2. Enquanto p for diferente de NULL ainda há elemento(s) na lista.
3. O incremento do p é representado por  $p = p \rightarrow prox$ .
4. Se o campo de informação do nó apontado pelo ponteiro p ( $p->info$ ) for igual ao valor procurado retorna-se o ponteiro p, o qual terá acesso ao campo de informação (info) e ao campo do endereço seguinte (prox), já que p é do tipo ListaSimp.
5. Caso não encontrado, retorna-se o valor nulo (NULL).

#### Observação

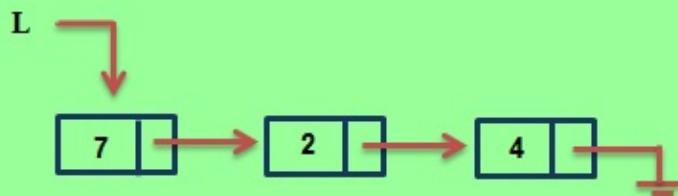
Na função principal (**main()**) a função buscar deve ser chamada como:

```
ListaSimp* aux = buscar_elemento(L, <valor a ser procurado>);
```

Para melhor entendimento, graficamente, realiza-se um exemplo a seguir, utilizando o conceito *teste de mesa*.

**Exemplo**

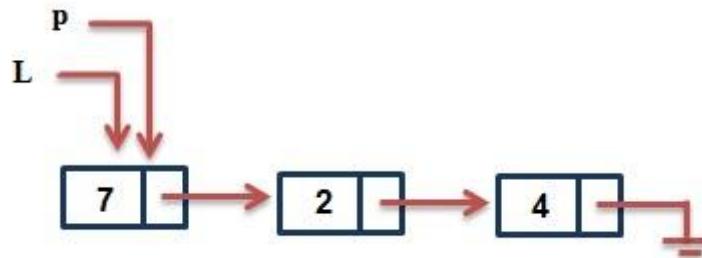
Considera-se a seguinte lista e o número 4 a ser procurado:



**1º PASSO:** Declara-se um ponteiro auxiliar denominado de p.

**2º PASSO:** Percorrer o laço for:

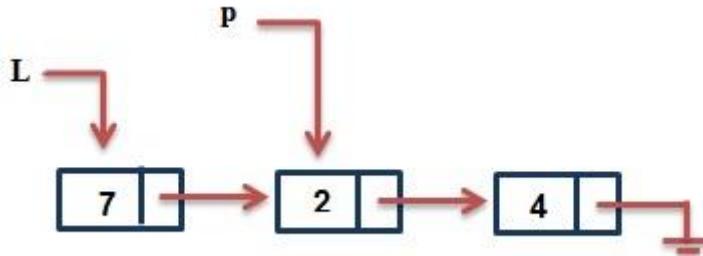
- O ponteiro p aponta para onde o ponteiro L aponta, ou seja, para o primeiro nó.



- O ponteiro p é diferente de NULL? SIM!! O ponteiro p aponta para o nó de valor 7.
- Entra na condicional if.
- O ponteiro p que na região info ( $p->info$ ) vale 7, é igual ao valor procurado que é 4, passado como parâmetro? NÃO!!

**3º PASSO:** Retorna ao laço for:

- O incremento do ponteiro p é dado por  $p = p \rightarrow prox$ , onde  $p \rightarrow prox$  é o elemento de valor 2. Assim, o ponteiro p aponta para este nó.

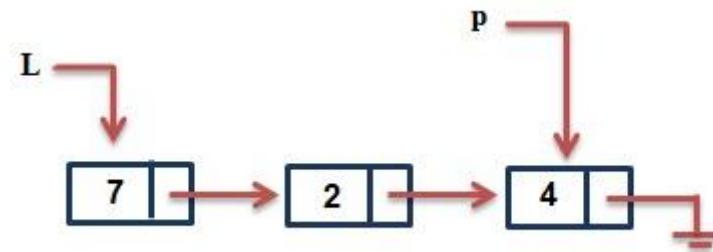


- O ponteiro p é diferente de NULL? SIM!! O ponteiro p aponta para o nó de valor 2.

- Entra na condicional if.
- O ponteiro p que na região info ( $p->info$ ) vale 2, é igual ao valor procurado que é 4, passado como parâmetro? NÃO!!

**4º PASSO:** Retorna ao laço for:

- O incremento do ponteiro p é dado por  $p = p -> prox$ , onde  $p -> prox$  é o elemento de valor 4. Assim, o ponteiro p aponta para este nó.



- O ponteiro p é diferente de NULL? SIM!! O ponteiro p aponta para o nó de valor 4.
- Entra na condicional if.
- O ponteiro p que na região info ( $p->info$ ) vale 4, é igual ao valor procurado que é 4, passado como parâmetro? SIM!!

**5º PASSO:** Retorna o nó apontado pelo ponteiro p, o qual possui valor 4.

#### 1.1.2.7. Liberar todos os elementos

##### O que fazer? Dicas...

- Percorrer a lista desalocando cada nó.
- Utilizar um ponteiro auxiliar que aponte para onde o ponteiro L aponta e, ainda, utilizar um outro ponteiro que aponte para o nó seguinte do primeiro ponteiro auxiliar criado.

A função denominada **liberar\_list**, apresentada na figura 20, possui as seguintes características:

- A função possui como parâmetro o ponteiro L.
- Não há retorno para a função principal, é do tipo void.

```

void liberar_lista (ListaSimp* L){

    1     ListoSimp* p = L;

    2     while (p != NULL) {

        3         ListoSimp* t = p -> prox;
        4         free(p);
        5         p = t;
        6     }
    }

```

Figura 20. Função liberar a Lista.

*Descrição das funcionalidades* apresentadas na figura 20:

1. O ponteiro L aponta para o primeiro nó da lista, este não deve ser manipulado para não perder a referência, então, utiliza-se um ponteiro auxiliar denominado de p, o qual aponta para onde o ponteiro L aponta, ou seja, para o primeiro nó.
2. No laço while, enquanto o ponteiro p for diferente de NULL ainda há elemento(s) na lista, então, executa-se os comandos encontrados dentro do laço while.
3. Utiliza-se um ponteiro auxiliar denominado de t que aponta para o nó a frente do ponteiro p ou para NULL, caso o ponteiro p aponte para o último nó.
4. Desaloca o nó apontado pelo ponteiro p.
5. O ponteiro p aponta para onde o ponteiro t aponta.
6. Após desalocar todos os elementos desaloca-se o ponteiro L.

#### Observação

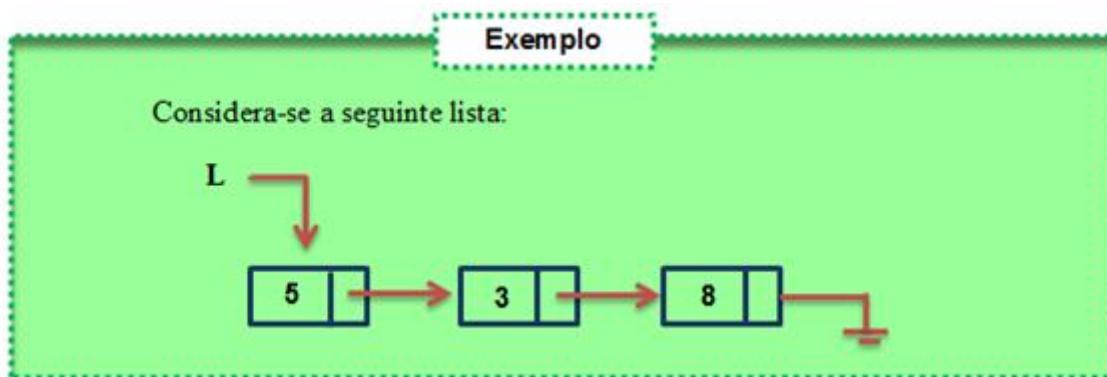
O ponteiro L é desalocado, assim, antes de inserir novos elementos deve-se chamar a função `cria_lista()`, na função principal (`main()`), para inicializar o ponteiro L com o valor nulo (NULL).

#### Observação

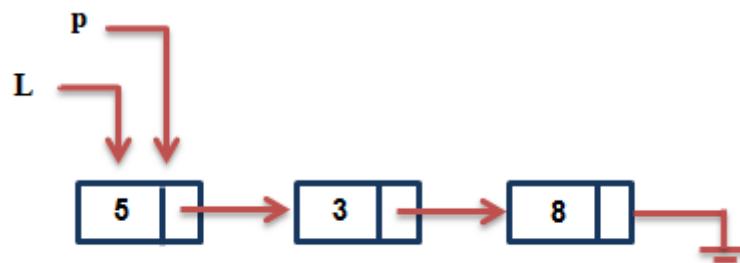
Na função principal (`main()`) a função liberar deve ser chamada como:

`liberar_lista(L);`

Para melhor entendimento, graficamente, realiza-se um exemplo a seguir, utilizando o conceito *teste de mesa*.

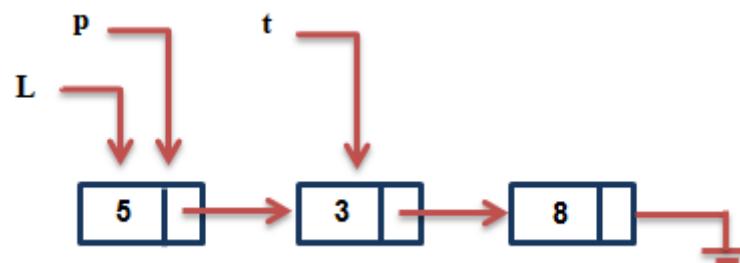


**1º PASSO:** O ponteiro p aponta para onde o ponteiro L aponta.

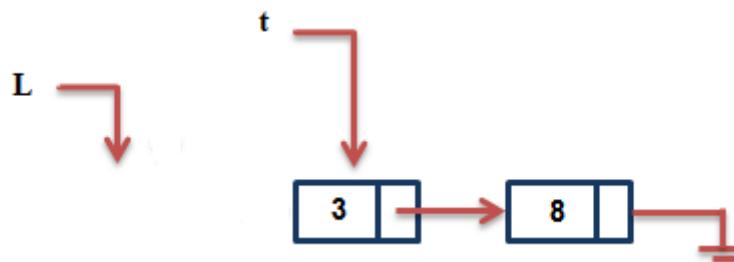


**2º PASSO:** Entra no laço while.

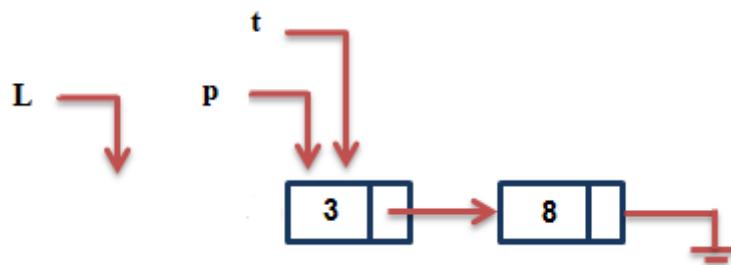
- O ponteiro p é diferente de NULL? SIM!! O ponteiro p aponta para o nó de valor 5.
- O ponteiro t aponta para o próximo nó do ponteiro p ( $p->prox$ ). No caso, apontará para o nó de valor 3.



- Desaloca o nó apontado pelo ponteiro p.

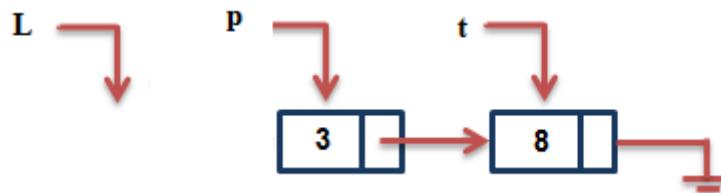


- O Ponteiro p aponta para onde o ponteiro t aponta.

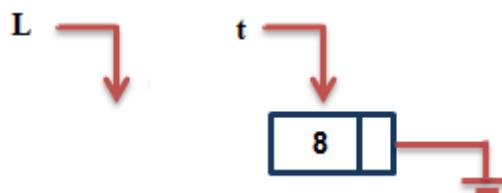


**3º PASSO:** Retorna ao laço while.

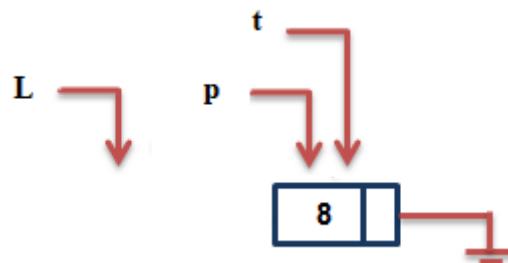
- O ponteiro p é diferente de NULL? SIM!! O ponteiro p aponta para o nó de valor 3.
- O ponteiro t aponta para o próximo nó do ponteiro p ( $p->prox$ ). No caso, apontará para o nó de valor 8.



- Desaloca o nó apontado pelo ponteiro p.



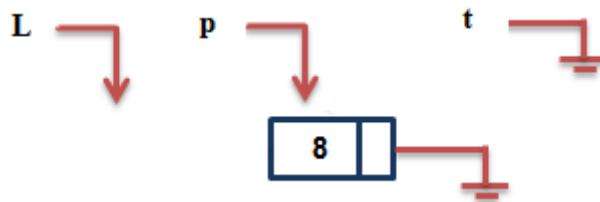
- Ponteiro p aponta para onde o ponteiro t aponta.



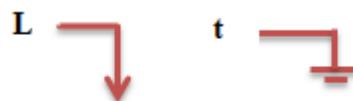
**4º PASSO:** Retorna ao laço while.

- O ponteiro p é diferente de NULL? SIM!! O ponteiro p aponta para o nó de valor 8.

- O ponteiro t aponta para o próximo nó do ponteiro p ( $p->prox$ ). No caso, apontará NULL.



- Desaloca o nó apontado pelo ponteiro p.



- Ponteiro p aponta para onde o ponteiro t aponta.



**5º PASSO:** Retorna ao laço while.

- O ponteiro p é diferente de NULL? NÃO!! O ponteiro p aponta NULL.

**6º PASSO:** Desaloca o ponteiro L.

**7º PASSO:** Terminam as iterações do laço while, como visto, a função liberar\_listा não possui retorno, contudo, liberou toda a lista.

#### 1.1.2.8. Retirar um nó

##### O que fazer? Dicas...

Utilizar um ponteiro auxiliar, o qual deve apontar para onde o ponteiro L aponta. O ponteiro auxiliar deve percorrer a lista comparando o valor de cada nó com o valor a ser removido. O elemento a ser removido poderá não estar na lista, entretanto, caso estiver, este poderá se encontrar:

- Na primeira posição, ou seja, ser o primeiro elemento da lista e, assim sendo, o ponteiro L aponta para este.
- No meio da lista, ou seja, entre dois nós.
- No final da lista, ou seja, ser o último elemento.

O que fazer? Dicas...

Se ocorrer:

- Caso 1, o ponteiro L deve apontar para o segundo elemento, o qual se tornará o primeiro da lista após a retirada do nó.
- Caso 2, o ponteiro L não será alterado, pois não haverá remoção do primeiro nó, contudo, o campo do endereço seguinte do nó anterior ao elemento a ser removido deverá apontar para o nó seguinte do nó a ser retirado.
- Caso 3, o ponteiro prox do nó anterior ao nó a ser removido deverá apontar para NULL.

Após a realização de um desses casos desaloca o nó a ser retirado.

A função denominada **retirar\_elemento**, apresentada na figura 21, possui as seguintes características:

- A função possui como parâmetro o valor a ser removido e o ponteiro L, do tipo ListaSimp, o qual aponta para o primeiro nó da lista.
- O retorno é do tipo ListaSimp, retorna a lista modificada ou inalterada.

```
ListaSimp* retirar_elemento(ListaSimp* L, int valor){
```

```
1     ListaSimp* p = L, *ant = NULL;
2
2     while (p != NULL && p -> prox != valor){
2         ant = p;
2         p = p -> prox;
3     }
4
4     if (p == NULL)
4         return L;
5
5     if (ant == NULL)
5         L = p -> prox;
6     else
6         ant -> prox = p -> prox;
7
7     free(p);
8
8     return L;
}
```

Figura 21. Função para retirar elemento da lista.

*Descrição das funcionalidades* apresentadas na figura 21:

1. O ponteiro L aponta para o primeiro nó da lista, este não deve ser manipulado para não perder a referência, assim, cria-se dois ponteiros auxiliares:
  - utiliza-se um ponteiro auxiliar denominado de p, o qual aponta para onde L aponta, ou seja, para o primeiro nó;
  - utiliza-se um ponteiro denominado de ant (anterior) que aponta para NULL.
2. Se p for diferente de NULL, ainda há elemento(s) na lista, e se o valor da região info (campo de informação) do ponteiro p for igual ao valor a ser retirado, então:
  - O ponteiro ant aponta para onde o ponteiro p aponta.
  - O ponteiro p é incrementado ( $p = p->prox$ ).
3. Baseando-se nos valores obtidos para o ponteiro p, anteriormente, se o ponteiro p for igual à NULL não encontrou o elemento, assim, retorna a lista inalterada.
4. Baseando-se nos valores obtidos para o ponteiro ant no tópico 2, se o ponteiro ant for igual à NULL significa que o ponteiro p não foi incrementado e, assim, o ponteiro ant não foi alterado, ou seja o elemento a ser removido é o primeiro.
5. O elemento a ser removido está na lista, contudo, no meio ou no fim, então, o ponteiro prox do nó anterior ao nó a ser removido, aponta para o próximo nó do elemento a ser retirado.
6. Desaloca o nó apontado pelo ponteiro p.
7. Retorna a lista.

#### Observação

Na função principal (`main()`) a função retirar deve ser chamada como:

`L = retirar_elemento(L, <valor do nó a ser removido>);`

Para melhor entendimento, graficamente, realiza-se quatro exemplos a seguir. Para a realização dos quatro exemplos será utilizada a lista representada na figura 19.

1. Retirar um elemento que não existe na lista.
2. Retirar o primeiro elemento.
3. Retirar o último elemento.
4. Retirar um elementos entre dois outros.

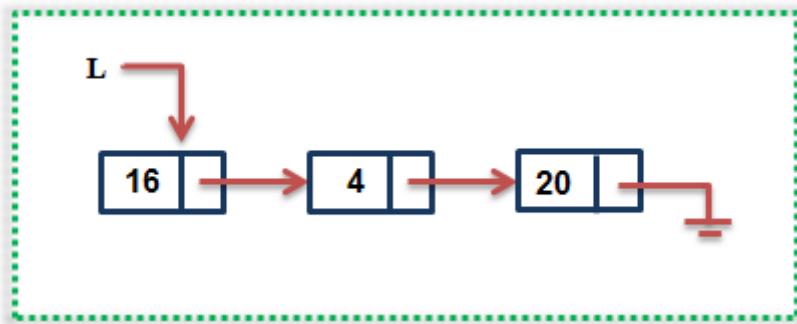


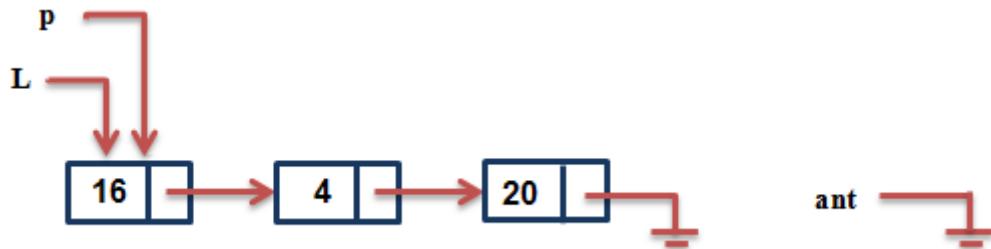
Figura 22. Lista a ser considerada nos exemplos.

### Exemplo 1

Considerando a lista da figura 22 deseja-se retirar o nó que contenha o valor 2.

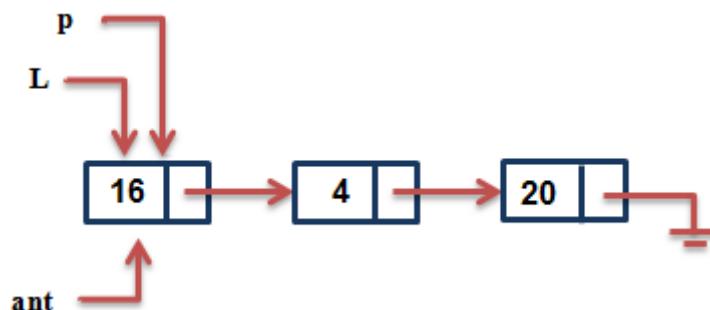
**1º PASSO:** Declaração de ponteiros auxiliares:

- Ponteiro p aponta para onde o ponteiro L aponta.
- Ponteiro ant aponta para NULL.

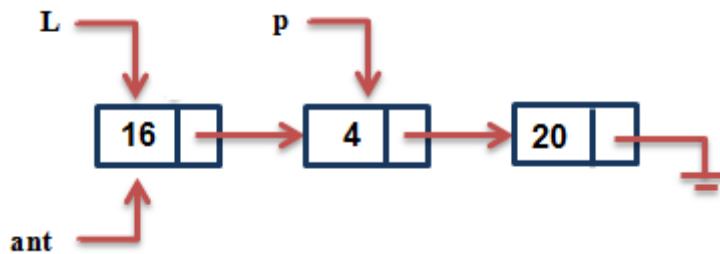


**2º PASSO:** Entra no laço while.

- O ponteiro p é diferente de NULL? SIM!! O ponteiro p aponta para o nó de valor de 16.
- O ponteiro p na região info ( $p->info$ ) possui valor 16 é diferente de 2? SIM!!
- O ponteiro ant aponta para onde o ponteiro p aponta.

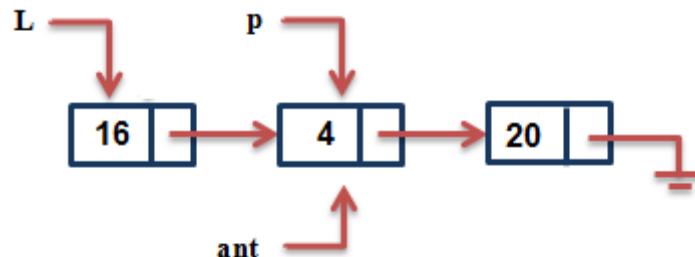


- O ponteiro  $p$  é incrementado.

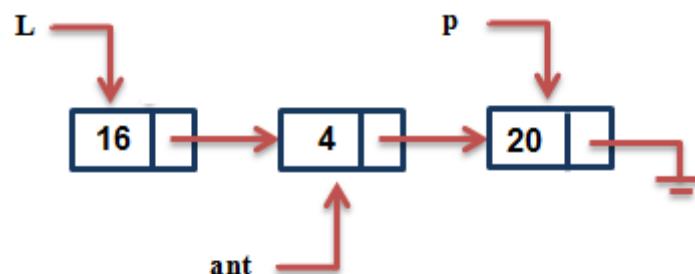


**3º PASSO:** Retorna ao laço while.

- O ponteiro  $p$  é diferente de NULL? SIM!!
- O ponteiro  $p$  na região info ( $p->info$ ) possui valor 4, é diferente de 2? SIM!!
- O ponteiro ant aponta para onde o ponteiro  $p$  aponta.

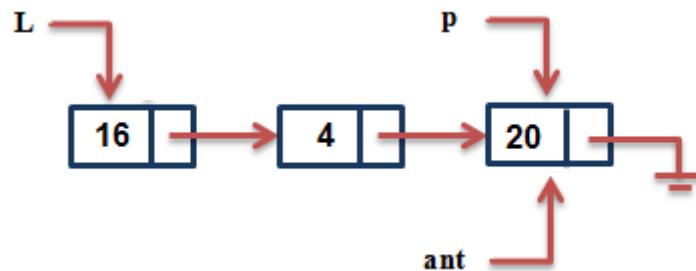


- O ponteiro  $p$  é incrementado.

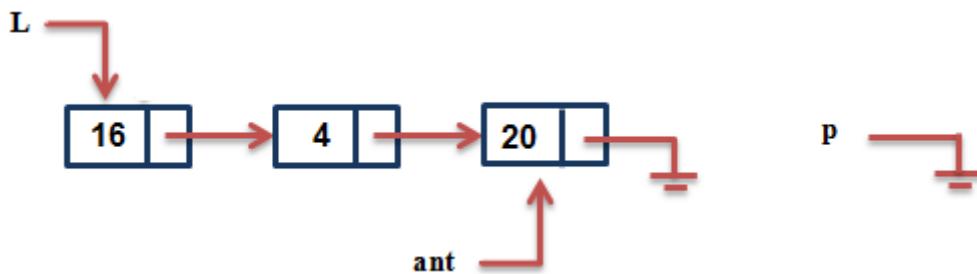


**4º PASSO:** Retorna ao laço while.

- O ponteiro  $p$  é diferente de NULL? SIM!!
- O ponteiro  $p$  na região info ( $p->info$ ) possui valor 20, é diferente de 2? SIM!!
- O ponteiro ant aponta para onde o ponteiro  $p$  aponta.



- O ponteiro  $p$  é incrementado.



**5º PASSO:** Retorna ao laço while.

- O ponteiro  $p$  é diferente de NULL? NÃO!!
- O ponteiro  $p$  na região info ( $p->info$ ) possui valor 20 é diferente de 2? SIM!!

**6º PASSO:** Termina as iterações do laço while.

**7º PASSO:** Entra na condicional if.

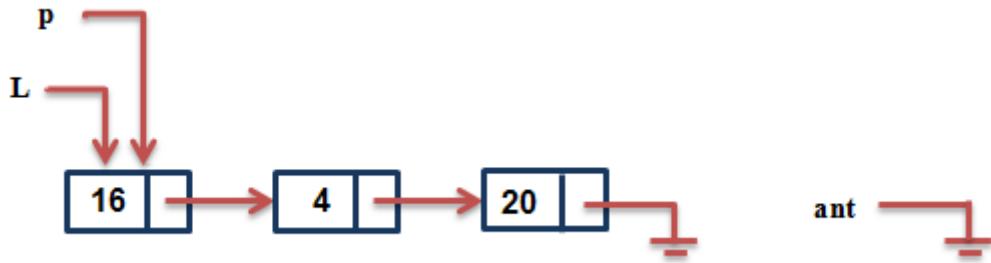
- O ponteiro  $p$  é igual à NULL, ou seja, aponta para NULL? SIM!! Não encontrou o elemento.
- Retorna a lista inalterada.

### Exemplo 2

Considerando a lista da figura 22 deseja-se retirar o nó que contém o valor 16.

**1º PASSO:** Declaração de ponteiros auxiliares:

- Ponteiro  $p$  aponta para onde o ponteiro  $L$  aponta.
- Ponteiro  $ant$  aponta para NULL.



**2º PASSO:** Entra no laço while.

- O ponteiro p é diferente de NULL? SIM!!
- O ponteiro p na região info ( $p->info$ ) possui valor 16 é diferente de 16? NÃO!!

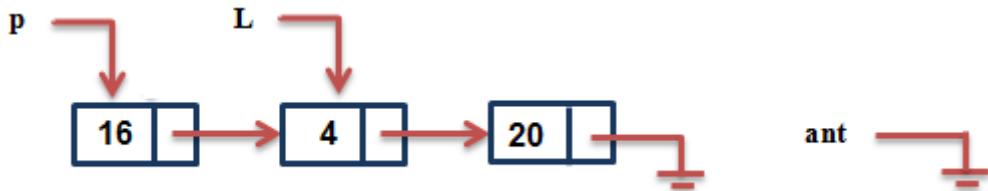
**3º PASSO:** Termina as iterações do laço while.

**4º PASSO:** Entra na condicional if.

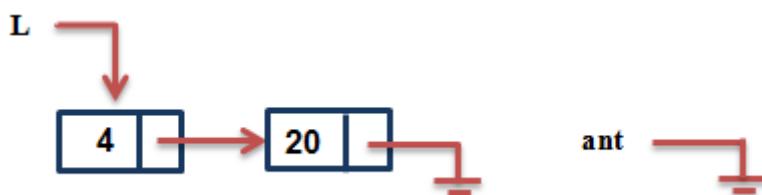
- O ponteiro p é igual à NULL, ou seja, aponta para NULL? NÃO!! O elemento encontra-se na lista.

**5º PASSO:** Entra na segunda condicional if.

- O ponteiro ant é igual à NULL, ou seja, aponta para NULL? SIM!! O elemento a ser retirado é o primeiro.
- O ponteiro L aponta para o próximo nó referente ao ponteiro p.



**6º PASSO:** desaloca o nó apontado pelo ponteiro p.



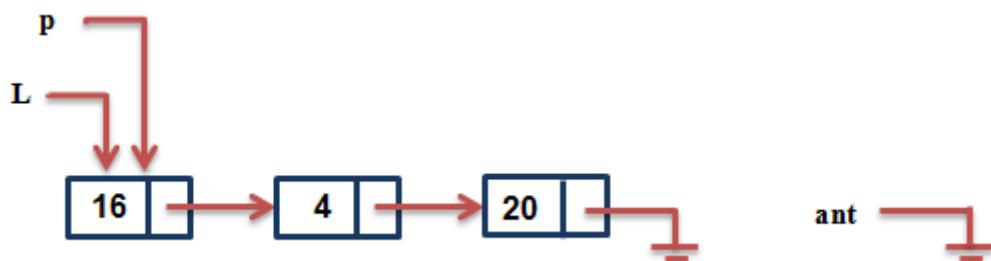
**7º PASSO:** Retorna da lista alterada.

**Exemplo 3**

Considerando a lista da figura 22 deseja-se retirar o nó que contenha o valor 20.

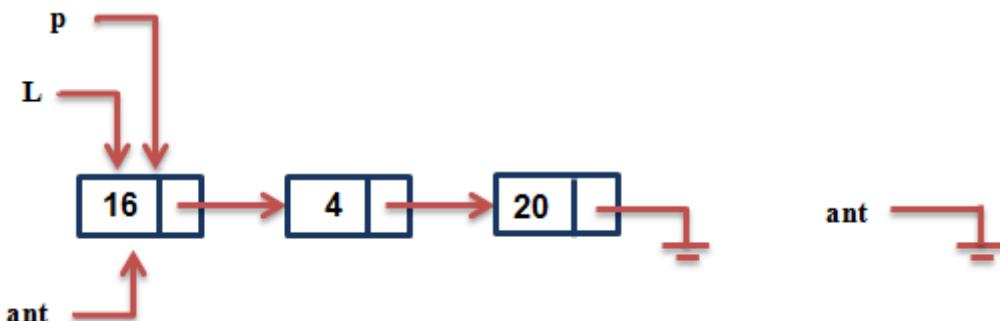
**1º PASSO:** Declaração de ponteiros auxiliares:

- Ponteiro p aponta para onde o ponteiro L aponta.
- Ponteiro ant aponta para NULL.

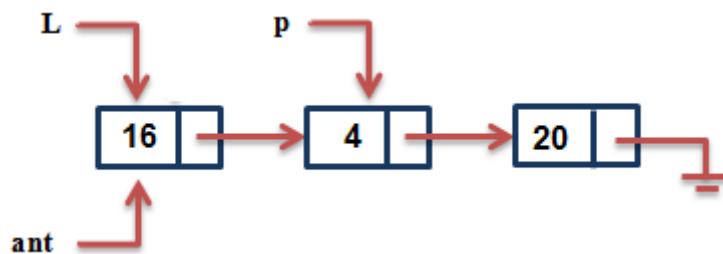


**2º PASSO:** Entra no laço while.

- O ponteiro p é diferente de NULL? SIM!!
- O ponteiro p na região info ( $p->info$ ) possui valor 16 é diferente de 20? SIM!!
- O ponteiro ant aponta para onde o ponteiro p aponta.

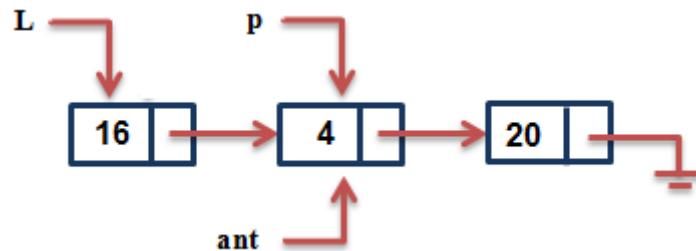


- O ponteiro p é incrementado.

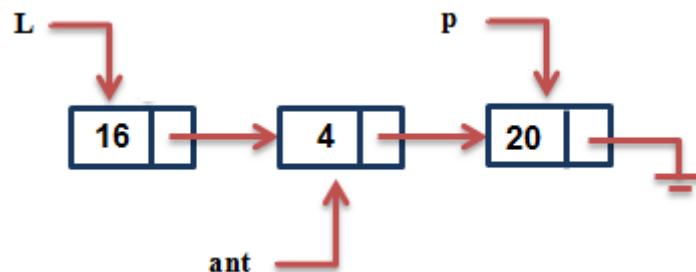


**3º PASSO:** Entra no laço while.

- O ponteiro p é diferente de NULL? SIM!!
- O ponteiro p na região info ( $p->info$ ) possui valor 4 é diferente de 20? SIM!!
- O ponteiro ant aponta para onde o ponteiro p aponta.



- O ponteiro p é incrementado.



**4º PASSO:** Entra no laço while.

- O ponteiro p é diferente de NULL? SIM!!
- O ponteiro p na região info ( $p->info$ ) possui valor 20 é diferente de 20? NÃO!!

**5º PASSO:** Termina as iterações do laço while.

**6º PASSO:** Entra na condicional if.

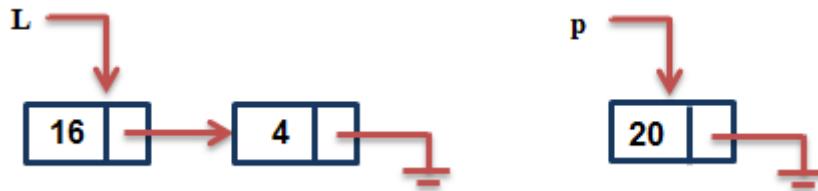
- O ponteiro p é igual à NULL, ou seja, aponta para NULL? NÃO!! O elemento encontra-se na lista.

**7º PASSO:** Entra na segunda condicional if.

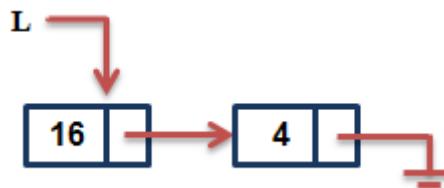
- O ponteiro ant é igual à NULL, ou seja, aponta para NULL? NÃO!!

**8º PASSO:** Entra na condicional else.

- O ponteiro prox contido no elemento apontado pelo ponteiro ant ( $ant->prox$ ) aponta para onde o ponteiro prox do nó apontado pelo ponteiro p aponta, o qual aponta para NULL.



**9º PASSO:** Desaloca o nó apontado pelo ponteiro p.



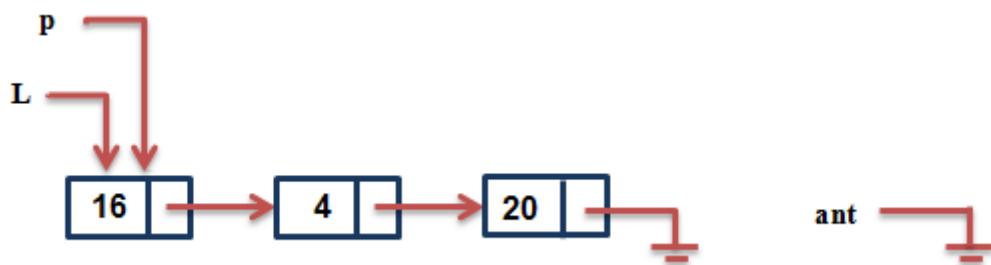
**10º PASSO:** Retorna da lista alterada.

#### Exemplo 4

Considerando a lista da figura 22 deseja-se retirar o nó que contenha o valor 4.

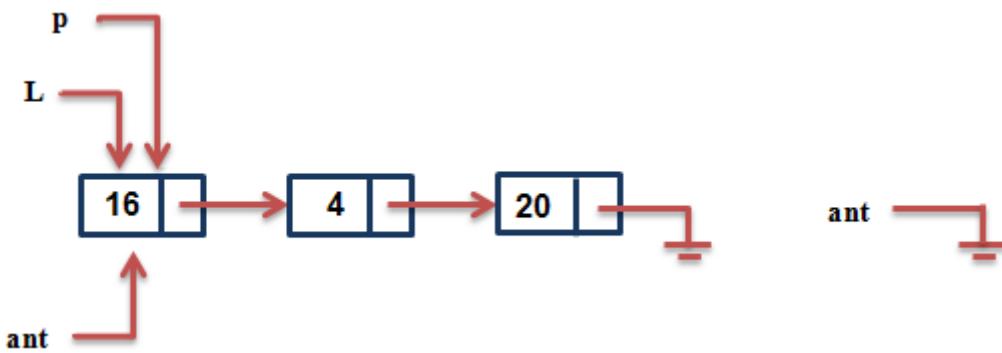
**1º PASSO:** Declaração de ponteiros auxiliares:

- Ponteiro p aponta para onde o ponteiro L aponta.
- Ponteiro ant aponta para NULL.

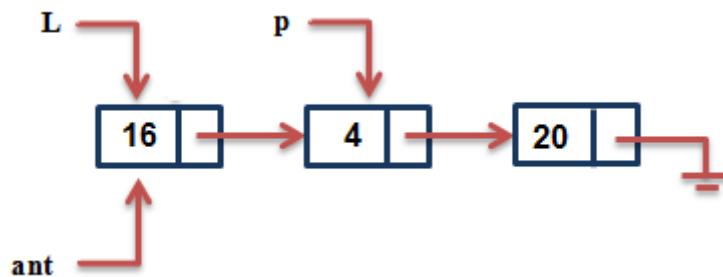


**2º PASSO:** Entra no laço while.

- O ponteiro p é diferente de NULL? SIM!!
- O ponteiro p na região info ( $p->info$ ) possui valor 16, é diferente de 4? SIM!!
- O ponteiro ant aponta para onde o ponteiro p aponta.



- O ponteiro  $p$  é incrementado.



**3º PASSO:** Entra no laço while.

- O ponteiro  $p$  é diferente de NULL? SIM!!
- O ponteiro  $p$  na região info ( $p->info$ ) possui valor 4, é diferente de 4? NÃO!!

**4º PASSO:** Termina as iterações do laço while.

**5º PASSO:** Entra na condicional if.

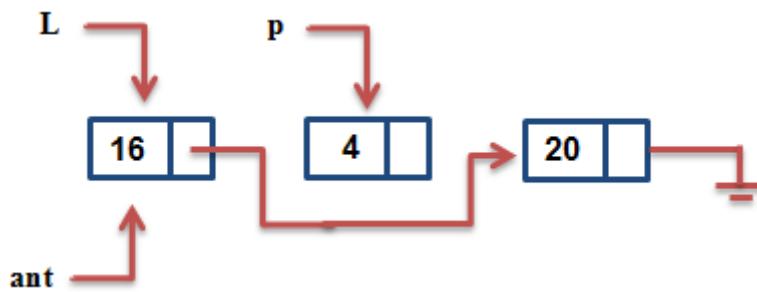
- O ponteiro  $p$  é igual à NULL, ou seja, aponta para NULL? NÃO!! O elemento encontra-se na lista.

**6º PASSO:** Entra na segunda condicional if.

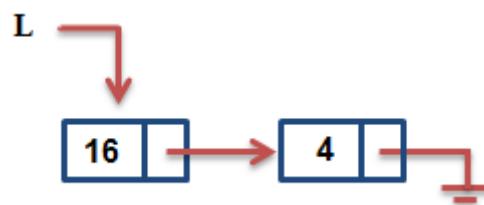
- O ponteiro  $ant$  é igual à NULL, ou seja, aponta para NULL? NÃO!! O elemento a ser retirado não é o primeiro nó da lista e, sim, encontra-se entre dois outros nós.

**7º PASSO:** Entra na condicional else.

- O ponteiro  $prox$  contido no elemento apontado pelo ponteiro  $ant$  ( $ant->prox$ ) aponta para onde o ponteiro  $prox$  do nó apontado pelo ponteiro  $p$  aponta, o qual aponta para nó de valor 20.



**8º PASSO:** desaloca o nó apontado pelo ponteiro p.



**9º PASSO:** Retorna da lista alterada.

## 1.2. Listas Duplamente Encadeadas

### 1.2.1. Conceitos

Uma lista duplamente encadeada é um conjunto de elementos, denominados de *nós* da lista. Cada nó contém três campos, como mostra a figura 23:

- *Campo de informação*: armazena o real elemento da lista.
- *Campo do endereço seguinte*: usado para acessar determinado nó. Este campo do endereço seguinte dentro de um nó pode ser visto como uma ligação ou um ponteiro para o próximo nó.
- *Campo do endereço anterior*: este campo dentro de um nó pode ser visto como uma ligação ou um ponteiro para o nó anterior.

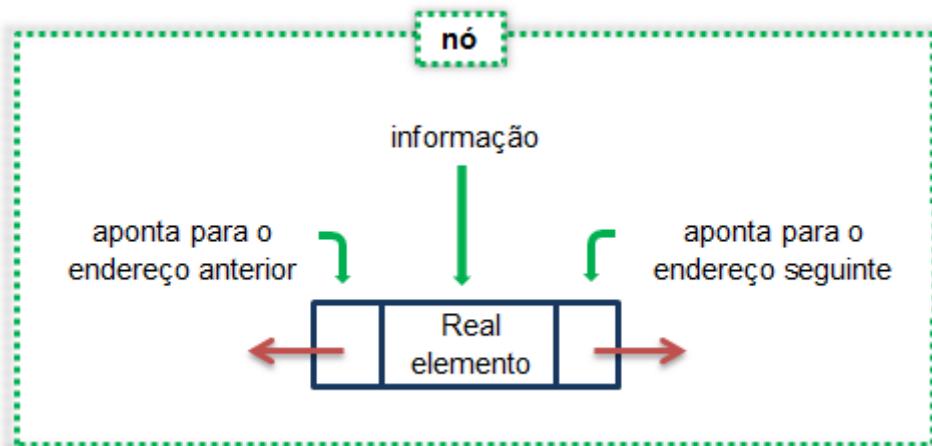


Figura 23. Composição de um nó da lista.

A lista inteira é acessada a partir de um ponteiro externo denominado **L**, este aponta para o primeiro nó da lista.

Tanto o *campo do endereço seguinte* do último nó quanto o *campo do endereço anterior* do primeiro nó da lista contêm um valor especial, conhecido como **NUL**, que não é um endereço válido. Como apresentado na figura 24.

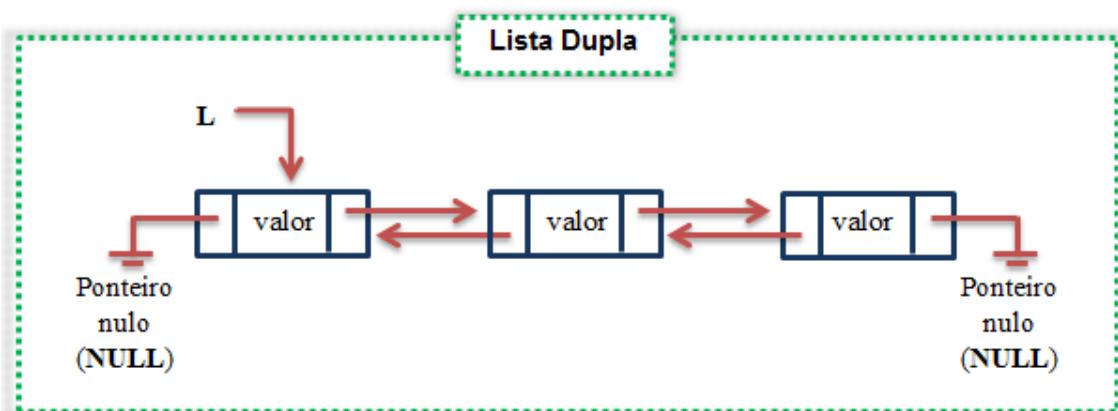


Figura 24. Representação de uma lista duplamente encadeada.

### Observação

Por ponteiro “*externo*”, entende-se aquele que não está incluído dentro de um nó. Em vez disso, seu valor pode ser acessado diretamente, por referência a uma variável.

A lista sem nós é chamada *lista vazia* ou *lista nula* (figura 25). O valor do ponteiro externo, para este tipo de lista, é o ponteiro nulo.

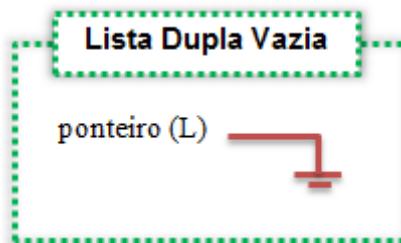


Figura 25. Representação de uma Lista Dupla Vazia.

Em uma lista duplamente encadeada os elementos podem ser manipulados em duas direções, como visto na figura 24, tanto da *esquerda para direita* quanto da *direita para esquerda*. Estas direções são determinadas pelos campos dos endereços seguintes e anteriores, os quais são ponteiros, que partem de um nó para seu sucessor e outro para seu anterior na lista dupla, respectivamente.

#### 1.2.2. Criação e Manipulação

Pode-se programar a criação da lista dupla e estabelecer as devidas funcionalidades a ela, entre estas se tem:

- 1.2.2.1. Criação de um Tipo Abstrato de Dado (TAD).
- 1.2.2.2. Criar a lista.
- 1.2.2.3. Inserir elementos do início na lista.
- 1.2.2.4. Averiguar se a lista está vazia.
- 1.2.2.5. Imprimir os elementos.
- 1.2.2.6. Buscar um determinado elemento na lista.
- 1.2.2.7. Liberar todos os elementos.
- 1.2.2.8. Retirar um nó.

### 1.2.2.1. Criação de um tipo abstrato de dado (TAD)

Criação de um novo tipo de dado, como mostra a figura 26.

```
typedef struct lista{
    1     int info;
    2     struct lista* prox;
    3     struct lista* ant;
}ListaDupla;
```

Figura 26. Estrutura de um novo tipo de dado do tipo ListaDupla.

**Descrição das funcionalidades** apresentadas na figura 26:

1. Declarou-se a informação (info) do *campo de informação*, o qual armazena o real elemento da lista, sendo este elemento do tipo inteiro (figura 27).
2. Declarou-se o ponteiro prox, referente ao *campo do endereço seguinte*, sendo do tipo struct lista (figura 27).
3. Declarou-se o ponteiro ant, referente ao *campo do endereço anterior*, sendo do tipo struct lista (figura 27).



Figura 27. Regiões declaradas como info, ponteiro prox e outra como ponteiro ant.

#### Observação

Para facilitar, com a utilização do `typedef`, em vez de usar a designação `struct lista` troca-se por `ListaDupla`. Entretanto, como visto no marcador 2 e 3 não se pode declarar o ponteiro prox e o ant como sendo do tipo `ListaDupla`, pois este ainda não havia sido referenciado.

### 1.2.2.2. Criar a lista

#### O que fazer? Dica...

Toda lista deve inicializar vazia!!!

A função denominada **criar\_lista**, apresentada na figura 28, possui as seguintes características (graficamente, está função está representada na figura 25, apresentada anteriormente):

- A função não possui parâmetros.
- O retorno é do tipo **ListaDupla**, retornando o valor nulo (NULL).

```
ListaDupla* criar_lista(void){
    return NULL;
}
```

Figura 28. Função referente à atribuição NULL para a Lista Dupla.

#### Observação

Na função principal (**main()**) a função **criar\_lista** deve ser chamada como:

```
L = criar_lista();
```

#### 1.2.2.3. Inserir no início da lista

##### O que fazer? Dicas...

- Deve alocar memória, de maneira dinâmica, para cada nó a ser inserido.
- O ponteiro L, aponta sempre para o primeiro nó da lista.
- A cada inserção o campo do endereço seguinte deste apontará para onde o ponteiro L aponta e, posteriormente, o ponteiro L apontará para o nó a ser inserido.
- O ponteiro ant do nó que era o primeiro da lista aponta para o nó inserido.
- O ponteiro ant do nó inserido aponta para NULL.

A função denominada **inserir\_elementos**, apresentada na figura 29, possui as seguintes características:

- A função possui como parâmetros o ponteiro L, do tipo **ListaDupla**, que aponta para o primeiro nó da lista, e o valor a ser inserido, este do tipo inteiro, no campo de informação.
- O retorno da função é do tipo **ListaDupla**, retornando o novo nó, assim, encadeando o elemento na lista existente.

```

ListaDupla* inserir_elementos(ListaDupla* L, int valor){

    1      ListaDupla* novo = (ListaDupla*)malloc(sizeof(ListaDupla));

    2      novo -> info = valor;
    3      novo -> prox = L;
    4      novo -> ant = NULL;

    5      if(L != NULL)
    5          L -> ant = novo;

    6      return novo;
}

```

Figura 29. Função referente à inserir elementos no início da lista dupla.

*Descrição das funcionalidades* apresentadas figura 29:

1. Utilizando o conceito de alocação dinâmica, aloca-se um nó denominado de *novo* do tipo *ListaDupla*.
2. O *novo -> info* recebe o valor, do tipo inteiro, fornecido por meio do parâmetro da função.
3. O *novo->prox* aponta para onde o ponteiro *L* aponta, se for para *NULL* apontará para *NULL*, senão, apontará para o nó designado como primeiro da lista.
4. O *novo->ant* aponta *NULL*.
5. Verifica se a lista não está vazia, ou seja, se o ponteiro *L* é diferente de *NULL*. Se caso não estiver vazia, o ponteiro *ant*, referenciado pelo ponteiro *L* (*L-> ant*), aponta para o novo nó.
6. Retorna o novo nó criado para o ponteiro *L*, assim, este aponta para o novo nó.

O ponteiro *novo* sendo do tipo *ListaDupla* tem acesso as regiões definidas de um nó. Assim, para referenciar a parte *info* do novo nó escreve-se *novo->info*, para a parte do ponteiro *prox* designa-se como *novo->prox*, assim como, para o ponteiro *ant* designa-se como *novo->ant*. Para melhor entendimento, graficamente, apresenta-se a figura 30.

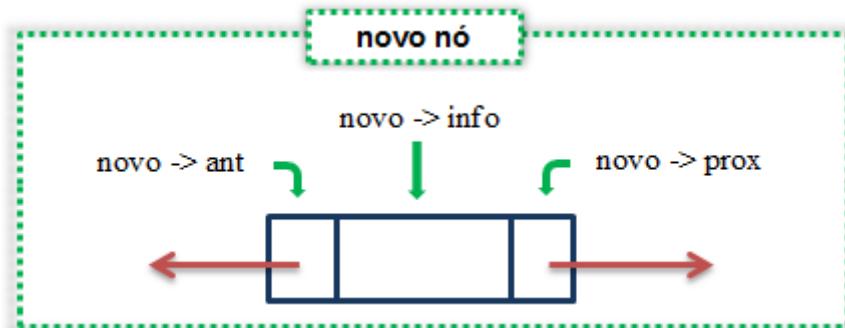


Figura 30. Regiões do nó denominado de novo.

Na função principal (**main()**) deve constar (figura 31):

```
main(void){  
    1     ListaDupla* L;  
    2     L = criar_lista();  
    3     L = inserir_elementos(L, <valor a ser inserido>);  
    }
```

Figura 31. Função principal chama a função criar\_lista e inserir\_elementos na lista dupla.

*Descrição das funcionalidades* apresentadas na figura 31:

1. Cria-se um ponteiro L, do tipo ListaDupla, o qual referenciará o início da lista.
2. Chamar a função criar\_lista para atribuir o primeiro valor a lista, o qual é nulo. O ponteiro L declarado recebe o retorno da função criar\_lista.
3. Chamar a função inserir\_elementos passando como parâmetros o ponteiro L (apontando para o primeiro nó, caso houver, ou para NULL caso a lista esteja vazia) e o valor a ser inserido. O ponteiro L declarado recebe o retorno da função inserir\_elementos, o qual é um novo nó.

Para melhor entendimento, graficamente, realiza-se um exemplo a seguir, utilizando o conceito *teste de mesa*.

### Exemplo

Inserir os seguintes elementos 4, 2, e 7.

**1º PASSO:** Na função principal colocar os seguintes comandos apresentados na figura 32:

```
main(void){  
    ListaDupla* L;  
    L = criar_lista();  
    L = inserir_elementos(L, 4);  
    L = inserir_elementos(L, 2);  
    L = inserir_elementos(L, 7);  
}
```

Figura 32. Função principal: inserção de elementos.

**2º PASSO:** Quando inicializada a função principal o ponteiro L, tipo ListaDupla, é declarado.

**3º PASSO:** O ponteiro L recebe o retorno da função criar\_lista, o qual é NULL.



**4º PASSO:** a função inserir\_elementos(L, 4) realiza os seguintes procedimentos:

- Aloca-se um novo nó.



- Este nó recebe o valor 4 no campo de informação (novo -> info).



- O campo do endereço seguinte do nó criado (novo -> prox) aponta para onde o ponteiro L aponta (Como visto anteriormente, no passo 3, aponta para NULL).



- O campo do endereço anterior do nó criado (novo -> ant) aponta para NULL.



- O ponteiro L é diferente de NULL? NÃO, pois ainda não houve inserção na lista, esta possui valor NULL. Como pode ser visto no passo 3.
- O ponteiro L, na função principal, recebe o novo nó, assim, deixa de apontar para NULL e aponta para o nó criado (figura 33).

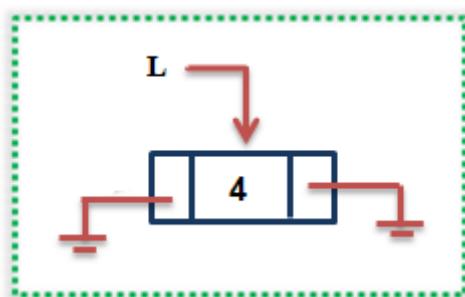


Figura 33. Inserção do elemento 4.

**5º PASSO:** a função inserir\_elementos(L, 2) realiza os seguintes procedimentos:

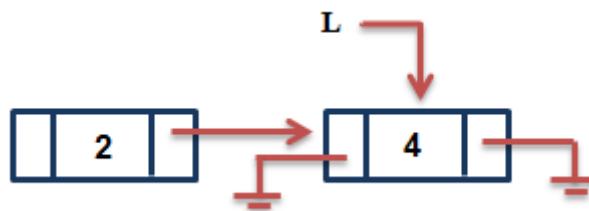
- Aloca-se um novo nó.



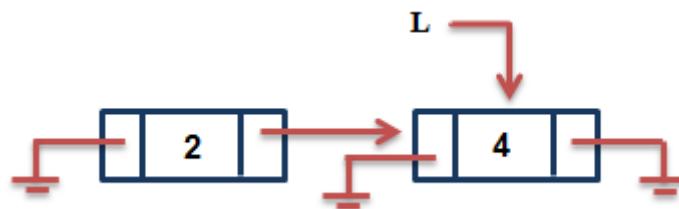
- Este nó recebe o valor 2 no campo de informação (novo -> info).



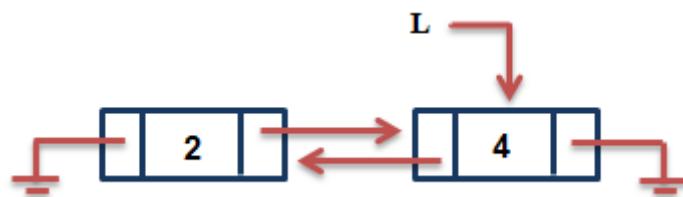
- O campo do endereço seguinte do nó criado (novo -> prox) aponta para onde o ponteiro L aponta (Como visto anteriormente, na figura 33, aponta para nó com valor 4).



- O campo do endereço anterior do nó criado (novo -> ant) aponta para NULL.



- O ponteiro L é diferente de NULL? SIM, pois aponta para o nó de valor 4. Assim, o ponteiro ant, referenciado pelo ponteiro L (L -> ant), aponta para o novo nó.



- O ponteiro L, na função principal, recebe o novo nó, assim, deixa de apontar para o nó com valor 4 e aponta para o novo nó criado com valor 2 (figura 34).

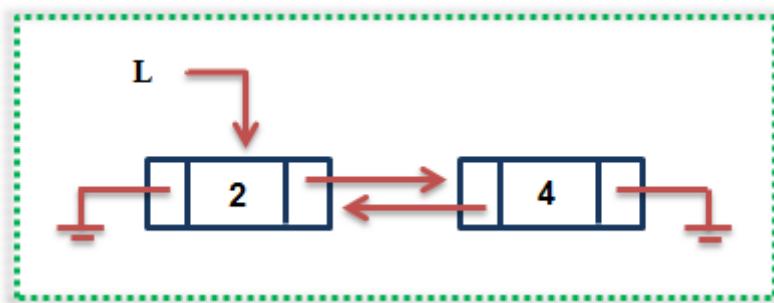


Figura 34. Inserção do elemento 2.

**6º PASSO:** a função inserir\_elementos( $L$ , 7) realiza os seguintes procedimentos:

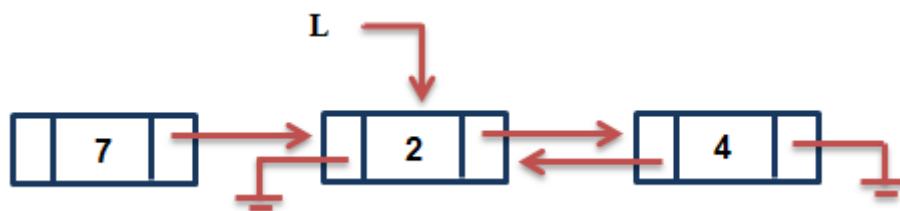
- Aloca-se um novo nó.



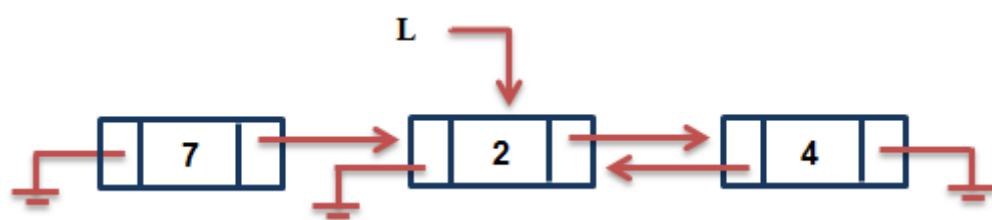
- Este nó recebe o valor 7 no campo de informação (novo  $\rightarrow$  info).



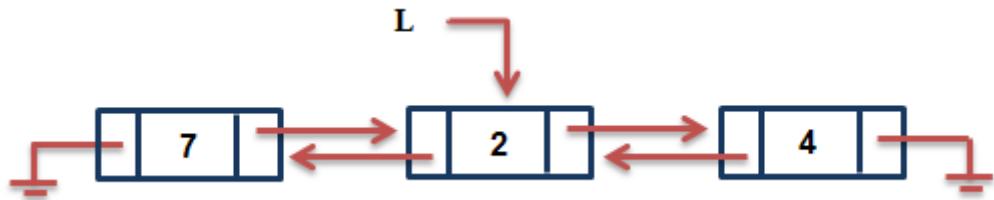
- O campo do endereço seguinte do nó criado (novo  $\rightarrow$  prox) aponta para onde o ponteiro  $L$  aponta (Como visto anteriormente, na figura 34, aponta para nó com valor 2).



- O campo do endereço anterior do nó criado (novo  $\rightarrow$  ant) aponta para NULL.



- O ponteiro  $L$  é diferente de NULL? SIM, pois aponta para o nó de valor 2. Assim, o ponteiro ant, referenciado pelo ponteiro  $L$  ( $L \rightarrow$  ant) aponta para o novo nó.



- O ponteiro L, na função principal, recebe o novo nó, assim, deixa de apontar para o nó com valor 2 e aponta para o novo nó criado com valor 7 (figura 35).

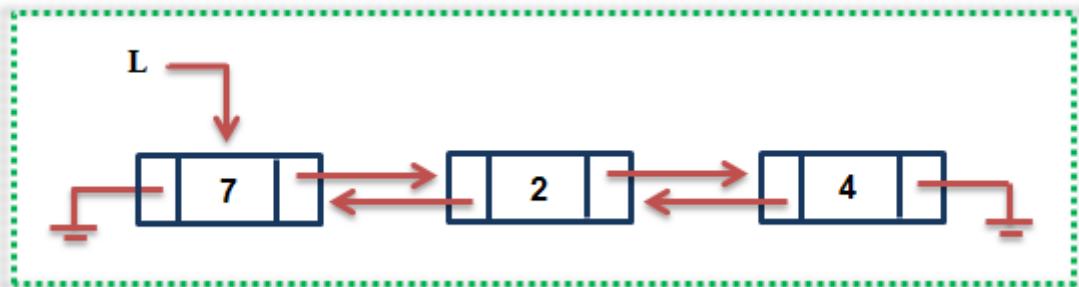


Figura 35. Inserção do elemento 7.

#### Reflexão...

Pode-se inserir no meio ou no fim, contudo, deve realizar modificações do código referente a inserir na lista.

O uso de desenhos auxilia na formação da lógica necessária para a formação do código.

#### 1.2.2.4. Averiguar se a lista está vazia

#### O que fazer? Dicas...

Verificar se a lista aponta para o valor nulo (NULL).

A função denominada **lista\_vazia**, apresentada na figura 36, possui as seguintes características:

- A função possui como parâmetro o ponteiro L o qual aponta para o primeiro nó da lista.
- O retorno é do tipo **booleano**, retornando valor 1 (verdadeiro) quando a lista estiver vazia, caso contrário, retorna valor 0 (falso);

```
int lista_vazia(ListaDupla* L){
    return (L == NULL);
}
```

Figura 36. Função que verifica se a lista está vazia.

#### 1.2.2.5. Imprimir os elementos

##### Observação

- Pode-se percorrer a lista, tanto da esquerda para direita quanto da direita para a esquerda, imprimindo os valores de cada nó.
- Não manipular o ponteiro L para não perder a referência do inicio da lista.

A função denominada **imprimir**, apresentada na figura 37, possui as seguintes características:

- A função possui como parâmetro o ponteiro L.
- O retorno é do tipo void, retornando na tela os valores contidos na lista.

```
void imprimir (ListaDupla* L){

1     ListaDupla* p;

2     if(lista_vazia(L))
2         printf("\nLista vazia!\n");
3     else{
3         for (p = L; p != NULL; p = p -> prox) {
3             printf("%d ", p -> info);
3         }
3         printf("\n\n");
3     }
}
```

Figura 37. Função imprimir.

*Descrição das funcionalidades* apresentadas na figura 37:

1. Declara-se um ponteiro auxiliar denominado de p do tipo ListaDupla.
2. Dentro da condicional chama-se a função lista\_vazia, caso retorne o valor 1 a lista está vazia e, assim, imprimi-se “Lista vazia!”.

3. Senão entra no laço for:

- Utiliza-se um ponteiro auxiliar denominado de p, o qual aponta para onde o ponteiro L aponta, ou seja, para o primeiro nó.
- Enquanto o ponteiro p for diferente de NULL ainda há elemento(s) na lista.
- O incremento do ponteiro p é representado por  $p = p \rightarrow prox$ .
- A impressão é baseada no valor armazenado na região do campo de informação (info).

#### Observação

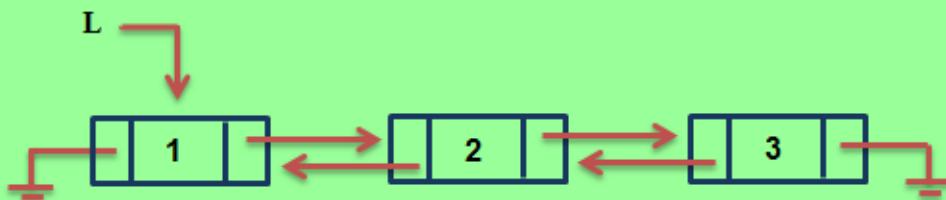
Na função principal (`main()`) a função imprimir deve ser chamada como:

`imprimir(L);`

Para melhor entendimento, graficamente, realiza-se um exemplo a seguir, utilizando o conceito *teste de mesa*.

#### Exemplo

Considerando a seguinte lista:



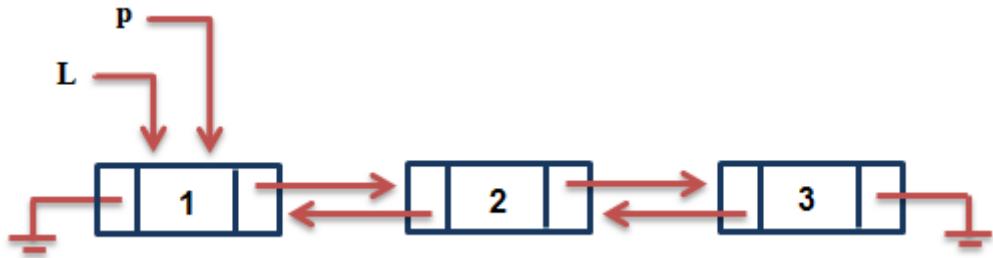
**1º PASSO:** Declara-se um ponteiro auxiliar denominado de p.

**2º PASSO:** Entra na condicional if.

- Chama-se a função `lista_vazia`, a lista está vazia? NÃO, pois o ponteiro L aponta para o nó de valor 1, ou seja, é diferente de NULL.

**3º PASSO:** Entra na condicional else. Percorre-se o laço for:

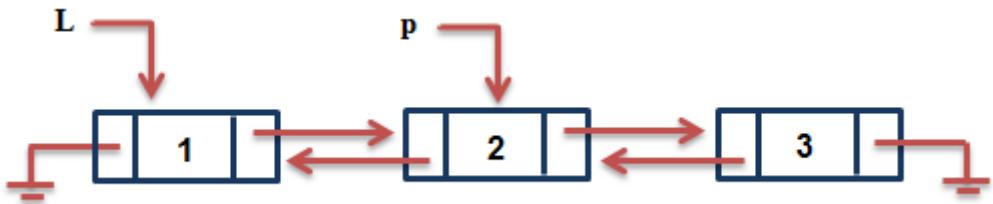
- O ponteiro p aponta para onde o ponteiro L aponta.



- O ponteiro  $p$  é diferente de NULL, ou seja, não aponta para NULL? SIM, assim imprime-se o valor 1 contido no elemento apontado pelo ponteiro  $p$  ( $p \rightarrow \text{info}$ ).

**4º PASSO:** Retorna ao laço for:

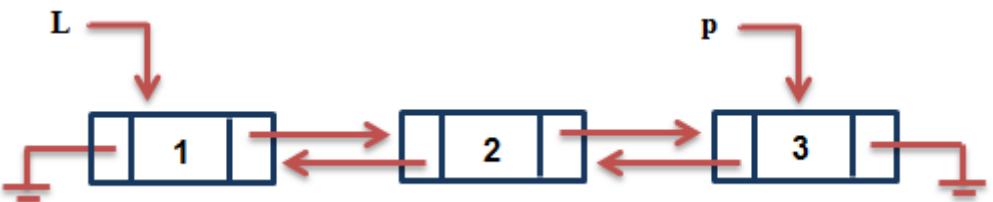
- O incremento do ponteiro  $p$  é dado por  $p = p \rightarrow \text{prox}$ , onde  $p \rightarrow \text{prox}$  é o elemento de valor 2. Assim, o ponteiro  $p$  aponta para este nó.



- O ponteiro  $p$  é diferente de NULL, ou seja, não aponta para NULL? SIM, assim imprime-se o valor 2 contido no elemento apontado pelo ponteiro  $p$  ( $p \rightarrow \text{info}$ ).

**5º PASSO:** Retorna ao laço for:

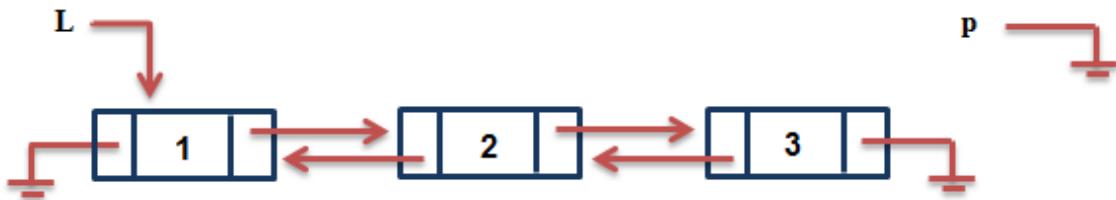
- O incremento do ponteiro  $p$  é dado por  $p = p \rightarrow \text{prox}$ , onde  $p \rightarrow \text{prox}$  é o elemento de valor 3. Assim, o ponteiro  $p$  aponta para este nó.



- O ponteiro  $p$  é diferente de NULL, ou seja, não aponta para NULL? SIM, assim imprime-se o valor 3 contido no elemento apontado pelo ponteiro  $p$  ( $p \rightarrow \text{info}$ ).

**6º PASSO:** Retorna ao laço for:

- O incremento do ponteiro  $p$  é dado por  $p = p \rightarrow \text{prox}$ , onde  $p \rightarrow \text{prox}$  possui valor nulo. Assim, o ponteiro  $p$  aponta para NULL.



- O ponteiro p é diferente de NULL, ou seja, não aponta para NULL? NÃO, ou seja, imprimiram-se todos os elementos contidos na lista.

**7º PASSO:** Termina o laço, assim, encerra a impressão!!

#### 1.2.2.6. Buscar um determinado elemento na lista

##### O que fazer? Dicas...

- Percorrer a lista comparando o valor que o nó possui com o valor a ser buscado.
- Caso encontre, retorna o nó que contém o valor. Caso contrário, retorna NULL.

A função denominada **buscar\_elemento**, apresentada na figura 38, possui as seguintes características:

- A função possui como parâmetro o ponteiro L, o qual aponta para o primeiro nó da lista, e o valor a ser procurado.
- O retorno é do tipo ListaDupla\*. Caso o elemento seja encontrado o ponteiro p aponta para ele, assim, retorna este nó, caso contrário, retorna o valor nulo (NULL).

```
ListaDupla* buscar_elemento (ListaDupla* L, int valor ){
```

```
    ListaDupla* p;
        for(p = L; p != NULL; p = p -> prox) {
                if(p -> info == valor)
                        return p;
        }
        return NULL;
}
```

Figura 38. Função buscar elemento na lista.

*Descrição das funcionalidades* apresentadas na figura 38:

1. O ponteiro L aponta para o primeiro nó da lista, este não deve ser manipulado para não perder a referência, então, utiliza-se um ponteiro auxiliar denominado de p, o qual aponta para onde L aponta, ou seja, para o primeiro nó.
2. Enquanto p for diferente de NULL ainda há elemento(s) na lista.
3. O incremento do p é representado por  $p = p -> prox$ .
4. Se o campo de informação do nó apontado pelo ponteiro p ( $p->info$ ) for igual ao valor procurado retorna-se o ponteiro p, o qual terá acesso ao campo de informação (info), ao campo do endereço seguinte (prox) e ao campo do endereço anterior (ant), já que p é do tipo ListaDupla.
5. Caso não encontrado, retorna o valor nulo (NULL).

#### Observação

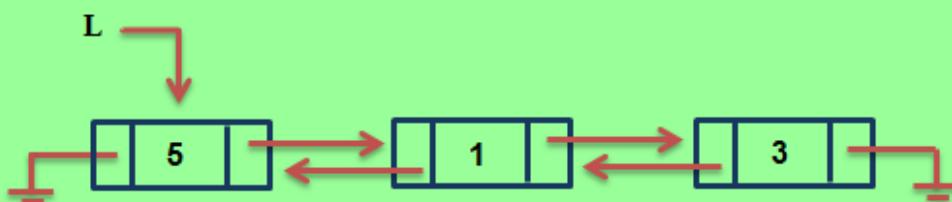
Na função principal (`main()`) a função buscar deve ser chamada como:

`ListaDupla* resultado = buscar_elemento (L, <valor a ser procurado>);`

Para melhor entendimento, graficamente, realiza-se um exemplo a seguir, utilizando o conceito *teste de mesa*.

#### Exemplo

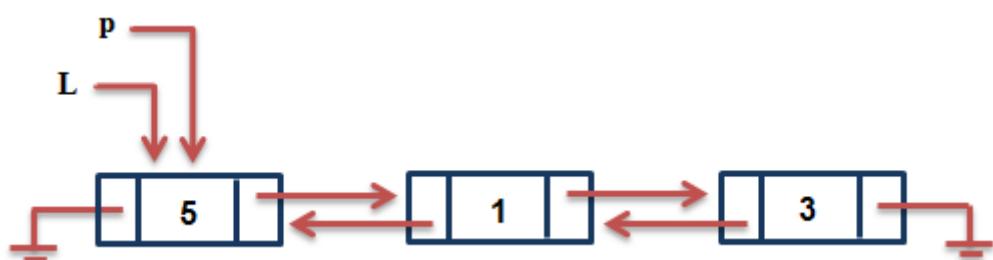
Considerando a seguinte lista e o número 1 a ser procurado:



**1º PASSO:** Declara-se um ponteiro auxiliar denominado de p.

**2º PASSO:** Percorrer o laço for:

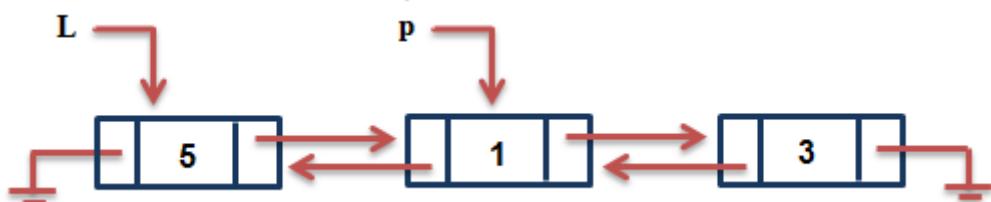
- O ponteiro p aponta para onde o ponteiro L aponta, ou seja, para o primeiro nó.



- O ponteiro  $p$  é diferente de NULL? SIM!! O ponteiro  $p$  aponta para o nó de valor 5.
- Entra na condicional if.
- O ponteiro  $p$  que na região info ( $p->info$ ) vale 5, é igual ao valor procurado que é 1, passado como parâmetro? NÃO!!

**3º PASSO:** Retorna ao laço for:

- O incremento do ponteiro  $p$  é dado por  $p = p -> prox$ , onde  $p -> prox$  é o elemento de valor 1. Assim, o ponteiro  $p$  aponta para este nó.



- O ponteiro  $p$  é diferente de NULL? SIM!! O ponteiro  $p$  aponta para o nó de valor 1.
- Entra na condicional if.
- O ponteiro  $p$  que na região info ( $p->info$ ) vale 1, é igual ao valor procurado que é 1, passado como parâmetro? SIM!!

**4º PASSO:** Retorna o nó apontado pelo ponteiro  $p$ , o qual possui valor 1.

#### 1.2.2.7. Liberar todos os elementos

##### O que fazer? Dicas...

- Percorrer a lista desalocando cada nó.
- Utilizar um ponteiro auxiliar que aponte para onde o ponteiro  $L$  aponta, e ainda, utilizar um outro ponteiro que aponte para o nó seguinte do ponteiro auxiliar criado.

A função denominada **liberar\_lista**, apresentada na figura 39, possui as seguintes características:

- A função possui como parâmetro o ponteiro  $L$ .
- Não há retorno para a função principal, é do tipo void.

```

void liberar(ListaDupla* L){

    1      ListDupla* p = L;

    2      while (p != NULL) {

        3          ListDupla* t = p -> prox;
        4          free(p);
        5          p = t;
        6      }
    }   free(L);

```

Figura 39. Função liberar a lista.

*Descrição das funcionalidades* apresentadas na figura 39:

1. O ponteiro L aponta para o primeiro nó da lista, este não deve ser manipulado para não perder a referência, então, utiliza-se um ponteiro um ponteiro auxiliar denominado de p, o qual aponta para onde o ponteiro L aponta, ou seja, para o primeiro nó.
2. No laço while, enquanto o ponteiro p for diferente de NULL ainda há elemento(s) na lista, então, executa-se os comandos encontrados dentro do laço while.
3. Utiliza-se um ponteiro auxiliar denominado de t que aponta para o nó a frente do ponteiro p ou para NULL, caso o ponteiro p aponte para o último nó.
4. Desaloca o nó apontado pelo ponteiro p.
5. O ponteiro p aponta para onde o ponteiro t aponta.
6. Após desalocar todos os elementos desaloca-se o ponteiro L.

#### Observação

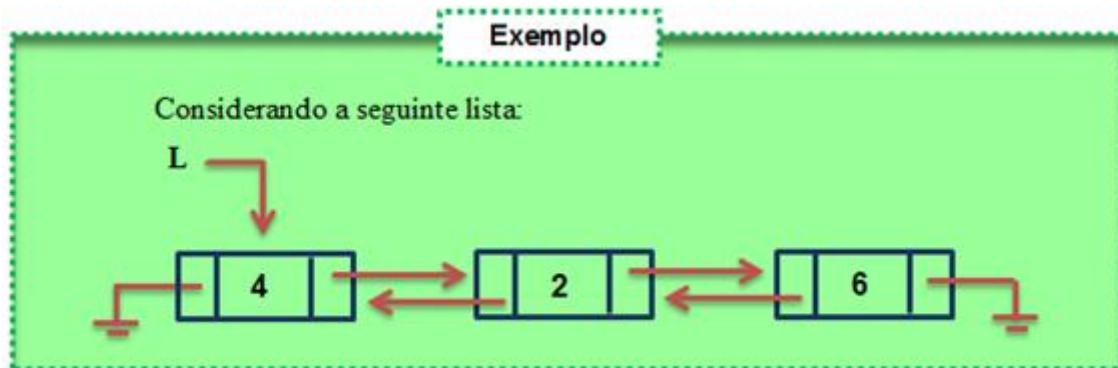
O ponteiro L é desalocado, assim, antes de inserir novos elementos deve-se chamar a função `cria_lista()`, na função principal (`main()`), para inicializar o ponteiro L com o valor nulo (NULL).

#### Observação

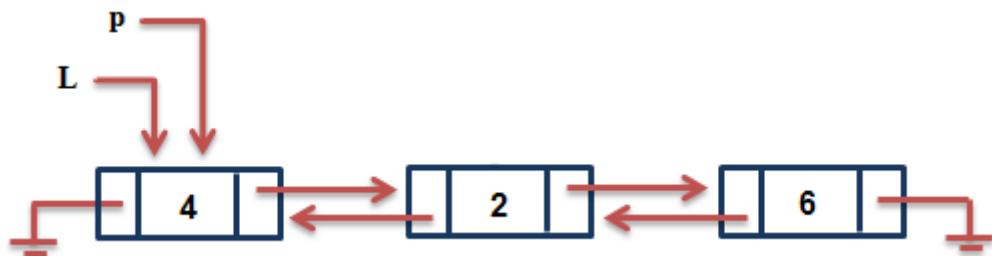
Na função principal (`main()`) a função `liberar` deve ser chamada como:

`liberar_lista(L);`

Para melhor entendimento, graficamente, realiza-se um exemplo a seguir, utilizando o conceito *teste de mesa*.

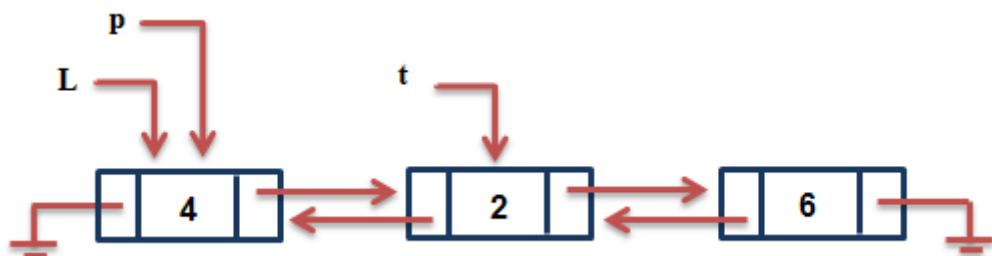


**1º PASSO:** Ponteiro p aponta para onde L aponta.

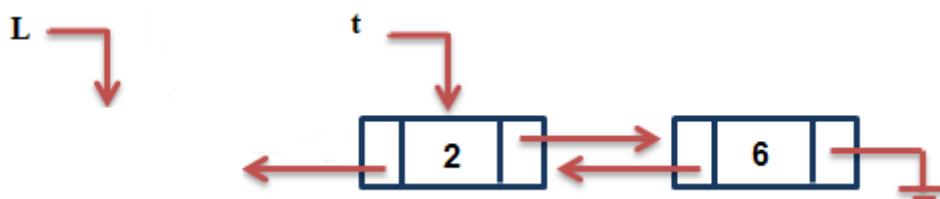


**2º PASSO:** Entra no laço while.

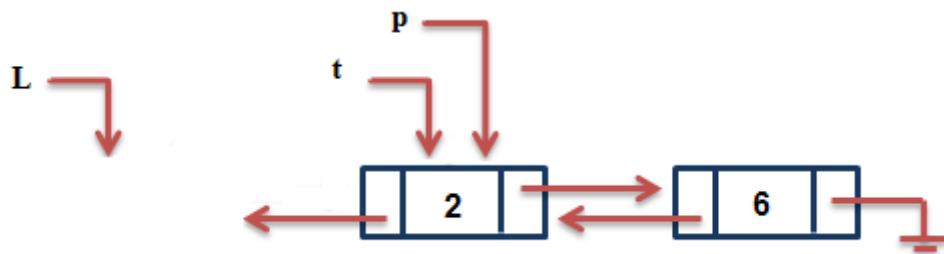
- O ponteiro p é diferente de NULL? SIM!! O ponteiro p aponta para o nó de valor 4.
- O ponteiro t aponta para o próximo nó do ponteiro p ( $p->prox$ ). No caso, apontará para o nó de valor 2.



- Desaloca o nó apontado pelo ponteiro p.

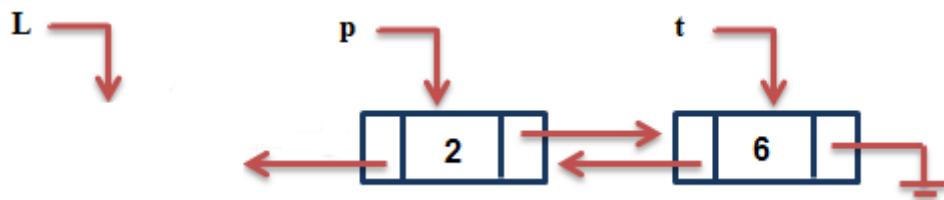


- O ponteiro p aponta para onde o ponteiro t aponta.

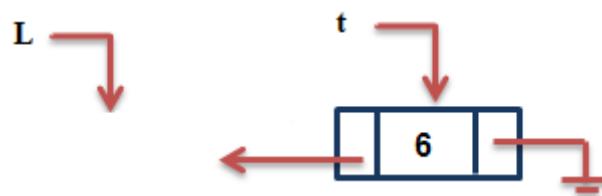


**3º PASSO:** Retorna ao laço while.

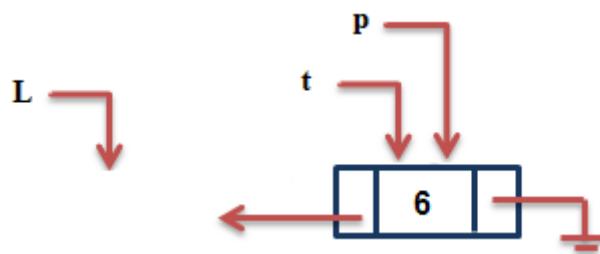
- O ponteiro p é diferente de NULL? SIM!! O ponteiro p aponta para o nó de valor 2.
- O ponteiro t aponta para o próximo nó do ponteiro p ( $p->prox$ ). No caso, apontará para o nó de valor 6.



- Desaloca o nó apontado pelo ponteiro p.



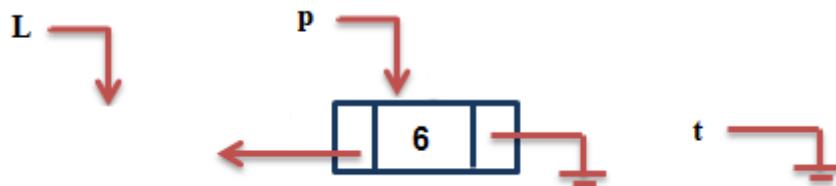
- O ponteiro p aponta para onde o ponteiro t aponta.



**4º PASSO:** Retorna ao laço while.

- O ponteiro p é diferente de NULL? SIM!! O ponteiro p aponta para o nó de valor 6.

- O ponteiro t aponta para o próximo nó do ponteiro p ( $p->prox$ ). No caso, apontará NULL.



- Desaloca o nó apontado pelo ponteiro p.



- O ponteiro p aponta para onde o ponteiro t aponta.



**5º PASSO:** Retorna ao laço while.

- O ponteiro p é diferente de NULL? NÃO!! O ponteiro p aponta NULL.

**6º PASSO:** Desaloca o ponteiro L.

**7º PASSO:** Terminam as iterações do laço while, como visto, a função liberar\_lista não possui retorno, contudo, liberou toda a lista.

#### 1.2.2.8. Retirar um nó

##### O que fazer? Dicas...

Utilizar ponteiro auxiliar, o qual deve apontar para onde o ponteiro L aponta. O ponteiro auxiliar deve percorrer a lista comparando o valor de cada nó com o valor a ser removido. O elemento a ser removido poderá não se estar na lista, entretanto, caso estiver, este poderá se encontrar:

- Na primeira posição, ou seja, ser o primeiro elemento da lista, e assim sendo, o ponteiro L aponta para este.
- No meio da lista, entre dois nós.
- No final da lista, ou seja, ser o último elemento.

O que fazer? Dicas...

Se ocorrer:

- Caso 1, o ponteiro L deve apontar para o segundo elemento, o qual se tornará o primeiro da lista após a retirada do nó. E o campo do endereço anterior do sucessor ao nó a ser removido deverá apontar para NULL.
- Caso 2, o ponteiro L não será alterado, pois não haverá remoção do primeiro nó, contudo, o campo do endereço seguinte do nó anterior ao elemento a ser removido deverá apontar para o nó seguinte do nó a ser retirado. E o campo do endereço anterior do nó sucessor ao nó a ser removido deverá apontar para o nó anterior do nó a ser retirado.
- Caso 3, o nó anterior ao nó a ser removido deverá apontar para NULL.

Após a realização de um desses casos desaloca o nó a ser retirado.

A função denominada **retirar\_elemento**, apresentada na figura 40, possui as seguintes características:

- A função possui como parâmetro o ponteiro L, do tipo ListaDupla, o qual aponta para o primeiro nó da lista e o valor a ser removido.
- O retorno é do tipo ListaDupla, retorna a lista modificada ou inalterada.

```
ListaDupla* retirar_elemento(ListaDupla* L, int valor){
```

```

1      ListaDupla* p = buscar_elemento(L, <valor a ser procurado>);

2      if (p == NULL)
2          return L;

3      if (L == p)
3          L = p -> prox;
4      else
4          p -> ant -> prox = p -> prox;

5      if (p -> prox != NULL)
5          p -> prox -> ant = p -> ant;

6      free(p);
7      return L;
}
```

Figura 40. Função para retirar elemento da lista.

**Descrição das funcionalidades** apresentadas na figura 40:

1. O ponteiro  $p$  recebe o retorno da função `buscar_elemento`. Caso o elemento estiver na lista o ponteiro  $p$  aponta para este nó, caso contrário, não encontrou o elemento e, assim, retorna `NULL`.
2. Se o ponteiro  $p$  for igual à `NULL`, percorreu a lista e não foi encontrado o elemento, assim, retorna a própria lista inalterada.
3. Se o ponteiro  $L$  aponta para o mesmo nó que o ponteiro  $p$  aponta, então, o nó a ser retirado é o primeiro. Assim, o ponteiro  $L$  aponta para o nó sucessor do ponteiro  $p$  ( $L = p->prox$ ).
4. Se o ponteiro  $L$  não aponta para o mesmo nó que o ponteiro  $p$  aponta, então, o nó a ser retirado pode ser o último da lista ou estar entre dois outros nós, contudo, está última condição será considerada se a próxima condicional for verdadeira. Assim, o ponteiro `prox` do nó anterior ao nó a ser removido ( $p->ant->prox$ ) aponta para o nó sucessor ao nó a ser retirado ( $p->prox$ ).
5. Se o ponteiro `prox` do nó apontado pelo ponteiro  $p$  ( $p->prox$ ) for diferente de `NULL`, então, o nó a ser removido encontra-se entre dois outros nós da lista. Assim, o ponteiro `ant` do nó sucessor ao nó a ser removido ( $p->ant->prox$ ) aponta para o nó anterior ao nó a ser retirado ( $p->ant$ ).
6. Desaloca o nó apontado pelo ponteiro  $p$ .
7. Retorna a lista alterada ou inalterada, caso o elemento a ser retirado não esteja na lista.

**Observação**

Na função principal (`main()`) a função `retirar` deve ser chamada como:

`L = retirar_elemento(L, <valor do nó a ser removido>);`

Para melhor entendimento, graficamente, realiza-se quatro exemplos a seguir:

1. **Retirar um elemento que não existe na lista.**
2. **Retirar o primeiro elemento.**
3. **Retirar o último elemento.**
4. **Retirar um elemento entre dois outros.**

Para melhor entendimento, graficamente, realiza-se quatro exemplos a seguir. Para a realização dos quatro exemplos será utilizada a lista representada na figura 41.

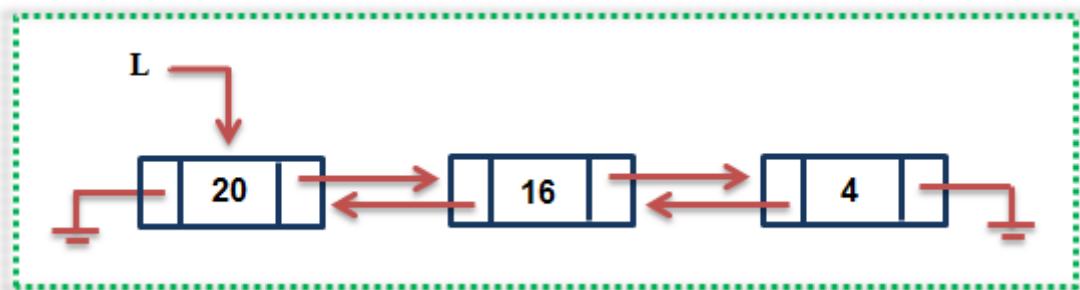


Figura 41. Lista a ser considerada nos exemplos.

### Exemplo 1

Considerando a lista da figura 41 deseja-se retirar o nó que contenha o valor 8.

**1º PASSO:** O ponteiro p recebe o retorno da função buscar\_elemento. A demonstração da funcionalidade desta função encontra-se no marcador 1.2.2.6 apresentada anteriormente. O ponteiro p apontará para NULL, já que o número 8 não se encontra na lista.



**2º PASSO:** Entra na primeira condicional if.

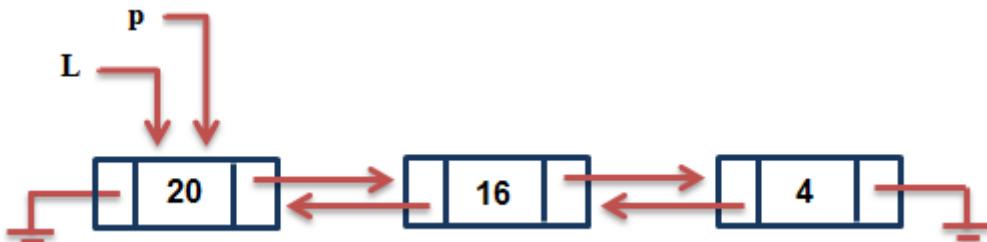
- O ponteiro p é igual à NULL? SIM!! O elemento não foi encontrado.

**3º PASSO:** Retorna a lista inalterada.

### Exemplo 2

Considerando a lista da figura 41 deseja-se retirar o nó que contenha o valor 20.

**1º PASSO:** O ponteiro p recebe o retorno da função buscar\_elemento. A demonstração da funcionalidade desta função encontra-se no marcador 1.2.2.6 apresentada anteriormente. O ponteiro p apontará para o primeiro nó com valor 20, no caso o primeiro da lista.

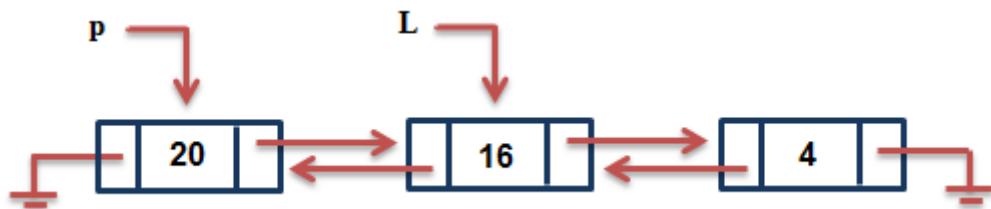


**2º PASSO:** Entra na primeira condicional if.

- O ponteiro p é igual à NULL? NÃO!! O nó a ser removido encontra-se na lista e este é apontado pelo ponteiro p.

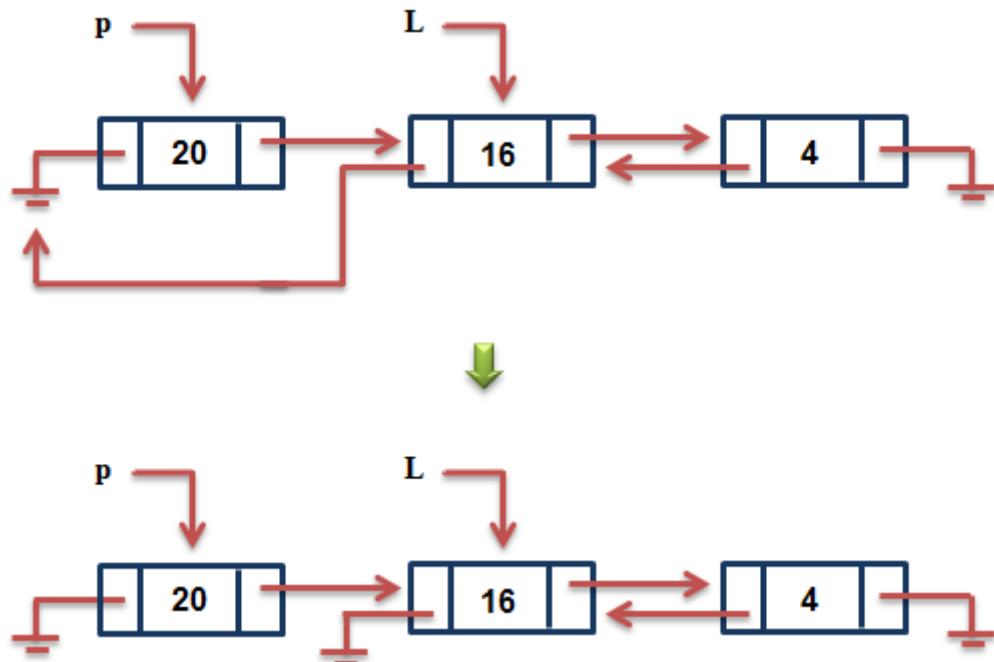
**3º PASSO:** Entra na segunda condicional if.

- O ponteiro L e o ponteiro p apontam para o mesmo nó? SIM!! O elemento a ser retirado é o primeiro elemento da lista.
- O ponteiro L aponta para o nó sucessor do ponteiro p.

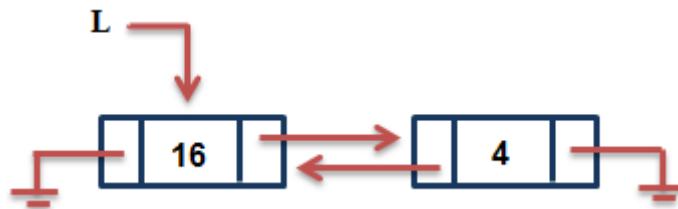


**4º PASSO:** Entra na terceira condicional if.

- O campo do endereço seguinte (ponteiro prox) do nó apontado pelo ponteiro p é diferente de NULL? SIM!!
- O ponteiro ant do nó sucessor ao nó a ser removido ( $p->prox->ant$ ) aponta para o nó anterior ao nó a ser retirado ( $p->ant$ ).



**5º PASSO:** Desaloca o nó apontado pelo ponteiro p.

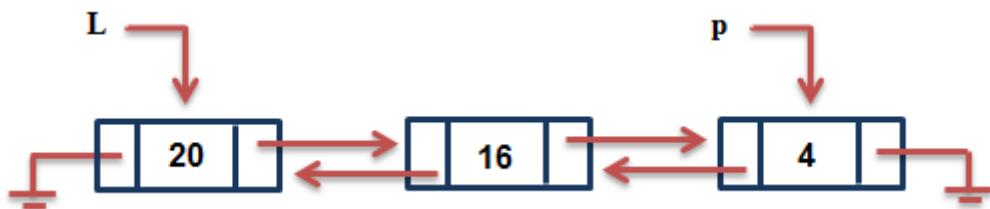


**6º PASSO:** Retorna a lista alterada.

### Exemplo 3

Considerando a lista da figura 41 deseja-se retirar o nó que contenha o valor 4.

**1º PASSO:** O ponteiro  $p$  recebe o retorno da função `buscar_elemento`. A demonstração da funcionalidade desta função encontra-se no marcador 1.2.2.6 apresentada anteriormente. O ponteiro  $p$  apontará para nó que possui o valor 4, no caso, o último da lista.



**2º PASSO:** Entra na condicional if.

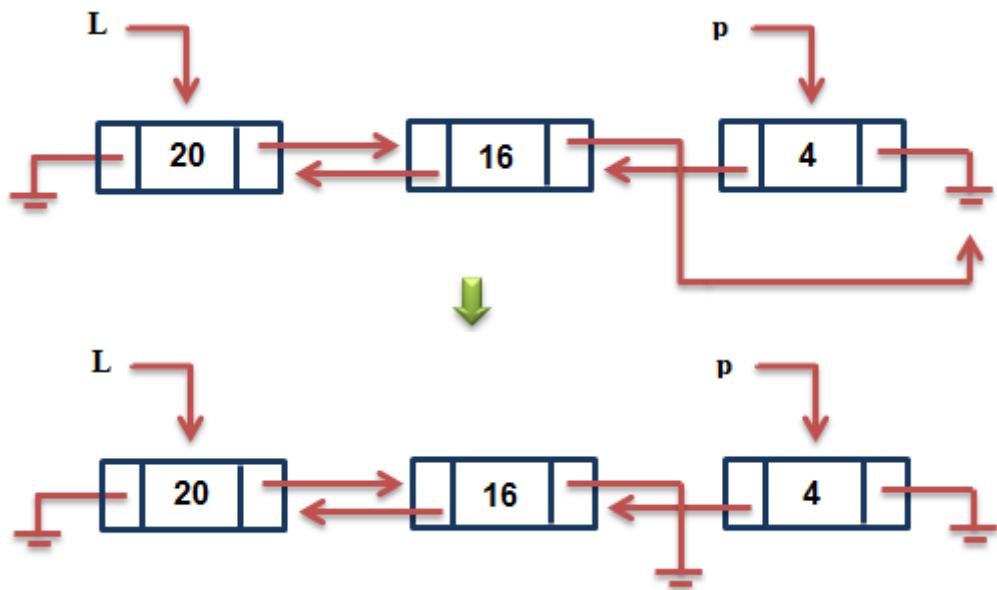
- O ponteiro  $p$  é igual à NULL? NÃO!! O nó a ser removido encontra-se na lista e este é apontado pelo ponteiro  $p$ .

**3º PASSO:** Entra na segunda condicional if.

- O ponteiro  $L$  e o ponteiro  $p$  apontam para o mesmo nó? NÃO!! O elemento a ser retirado não é o primeiro.

**4º PASSO:** Entra na condicional else.

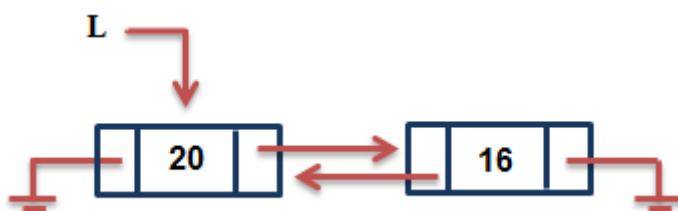
- O ponteiro  $prox$  do nó anterior ao nó a ser removido ( $p->ant->prox$ ) aponta para o nó sucessor ao nó a ser retirado ( $p->prox$ ), no caso, aponta para o valor nulo.



**5º PASSO:** Entra na terceira condicional if.

- O campo do endereço seguinte (ponteiro prox) do nó apontado pelo ponteiro p é diferente de NULL? NÃO!! O nó a ser removido é o último da lista.

**6º PASSO:** Desaloca o nó apontado pelo ponteiro p.

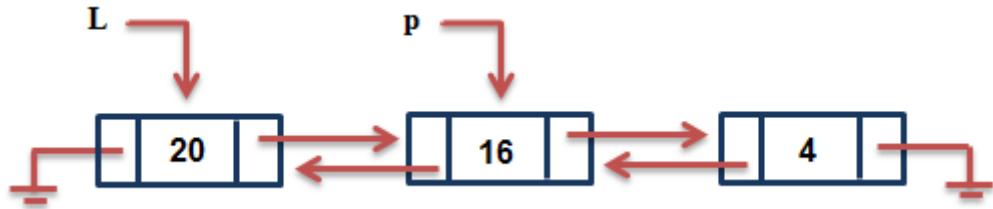


**7º PASSO:** Retorna a lista alterada.

#### Exemplo 4

Considerando a lista da figura 41 deseja-se retirar o nó que contém o valor 16.

**1º PASSO:** O ponteiro p recebe o retorno da função buscar\_elemento. A demonstração da funcionalidade desta função encontra-se no marcador 1.2.2.6 apresentada anteriormente. O ponteiro p apontará para o nó que possui o valor 16, no caso, o segundo da lista.



**2º PASSO:** Entra na primeira condicional if.

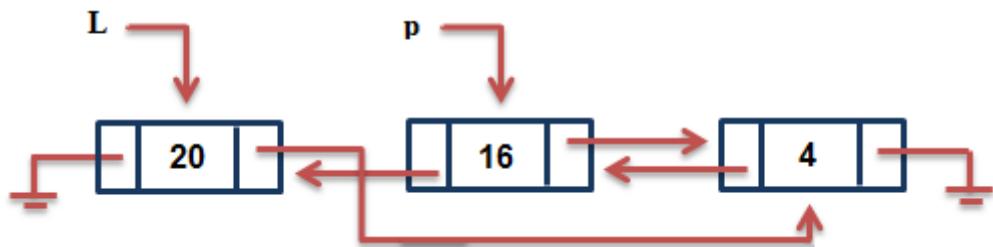
- O ponteiro p é igual à NULL? NÃO!! O nó a ser removido encontra-se na lista e este é apontado pelo ponteiro p.

**3º PASSO:** Entra na segunda condicional if.

- O ponteiro L e o ponteiro p apontam para o mesmo nó? NÃO!! O elemento a ser retirado não é o primeiro.

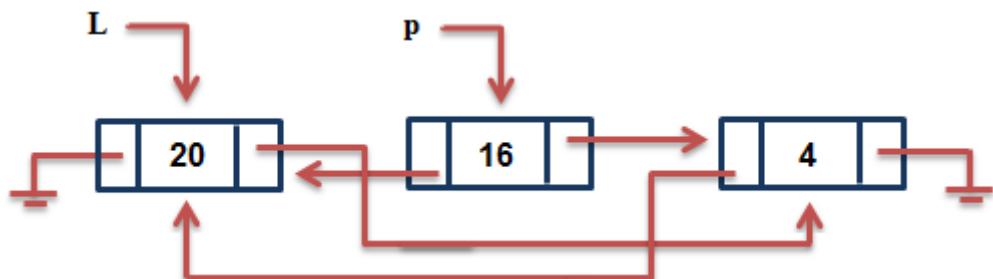
**4º PASSO:** Entra na condicional else.

- O ponteiro prox do nó anterior ao nó a ser removido ( $p->ant->prox$ ) aponta para o nó sucessor ao nó a ser retirado ( $p->prox$ ).

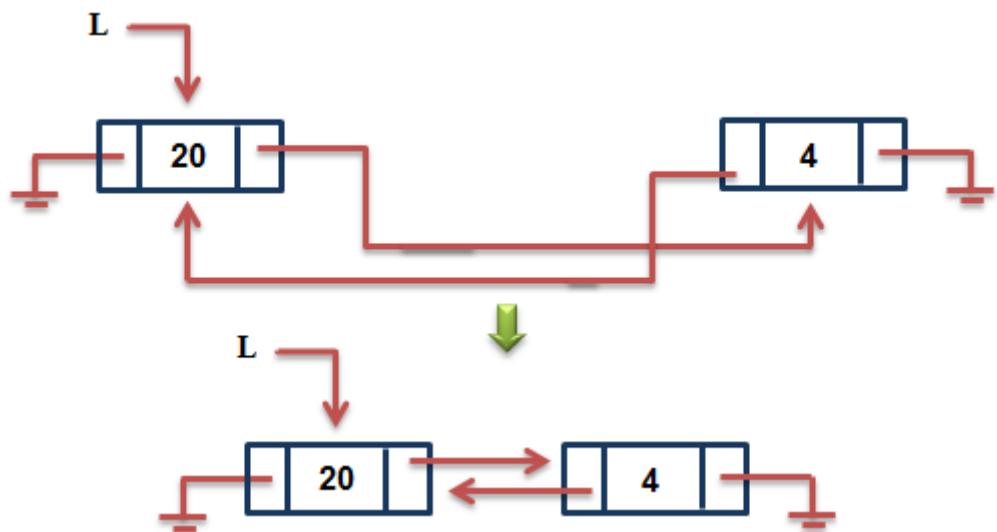


**5º PASSO:** Entra na terceira condicional if.

- O campo do endereço seguinte (ponteiro prox) do nó apontado pelo ponteiro p é diferente de NULL? SIM!! O nó a ser removido encontra-se entre doi outros nós.
- O ponteiro ant do nó sucessor ao nó a ser removido ( $p->prox->ant$ ) aponta para o nó anterior ao nó a ser retirado ( $p->ant$ ).



**6º PASSO:** desaloca o nó apontado pelo ponteiro p.



**7º PASSO:** Retorna a lista alterada.

### 1.3. Lista Circular

#### 1.3.1. Conceitos

Uma lista circular é um conjunto de elementos, denominados de nós da lista. Cada nó contém dois campos:

- *campo de informação*: armazena o real elemento da lista.
- *campo do endereço seguinte*: usado para acessar o próximo nó. Este campo dentro de um nó pode ser visto como uma ligação ou um ponteiro para o próximo nó.

O último elemento tem como próximo o primeiro nó da lista, formando um ciclo, como mostra a figura 42. A lista pode ser representada por um ponteiro para um elemento inicial qualquer da lista.

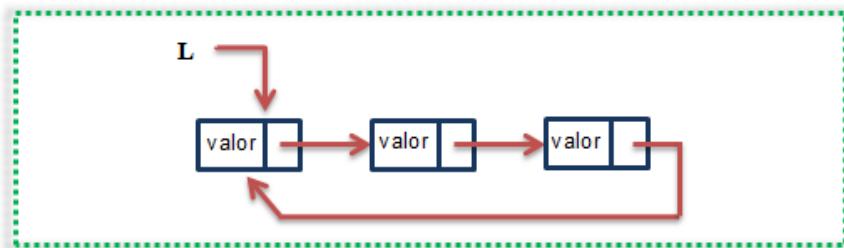


Figura 42. Representação de uma lista circular.

##### 1.3.1.1. Vantagens e desvantagens

As vantagens e desvantagens da utilização de uma lista circular são descritas no quadro 1.

Quadro 1. Vantagens e desvantagens da lista circular.

Vantagens	Desvantagens
Otimização dos recursos de memória.	Acesso indireto aos nós.
Não necessita deslocar os nós nas operações de inserção e remoção, apenas modifica os ponteiros.	Necessidade de percorrer a lista para ter acesso a um elemento.
É possível, sem reiniciar do ponto inicial, percorrer a lista diversas vezes.	A lista não possui um fim definido, ou seja, o valor nulo (NULL) não está presente na lista.
Inclusão e remoção de nós definidos como primeiro ou último não são considerados.	

### 1.3.2. Manipulação

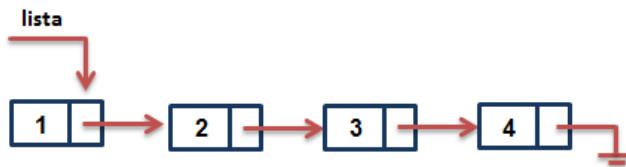
Pode-se programar a criação da lista e estabelecer funcionalidades a ela. O código a seguir, apresentado na figura 43, se refere a imprimir os valores contido em uma lista circular.

```
void imprimir(ListaCirc* L){  
  
    ListaCirc* p = L;          /*faz p apontar para o nó inicial*/  
  
    /*testa se a lista não é vazia e então percorre com do-while*/  
    if(p) do{  
        /*imprime a informação do nó*/  
        printf("%d\n", p->info);  
        p = p -> prox;           /*avança para o próximo nó*/  
    }while(p != L);  
}
```

Figura 43. Função imprimir.

## 1.4. Exercícios

1) Dada a lista:



a) Qual será a configuração final da lista após a execução dos comandos abaixo?

```

for(r = 1; r <= 3; r++){
    if(lista != NULL)
        exc = lista;
    lista = lista -> prox;
    free(exc);
}
  
```

2) Criar um TAD para **lista duplamente encadeada** que contenha as seguintes operações:

a) Inserir elemento x na última posição da lista. Protótipo:

```
void Insere(TipoItem x, TipoLista *lista);
```

b) Inserir elemento x após célula E. Protótipo:

```
void InsereAposElemento(TipoItem x, Apontador E);
```

c) Remover a célula C e retornar o apontador para a próxima célula da lista. Protótipo:

```
Apontador *RemoveElemento(Apontador E);
```

d) Retornar o apontador para a célula após E. Protótipo:

```
Apontador RetornaProxElemento(Apontador E);
```

e) Imprimir os elementos da lista. Protótipo:

```
void ImprimeLista(TipoLista *lista);
```

f) Contar o número de elementos na lista. Protótipo:

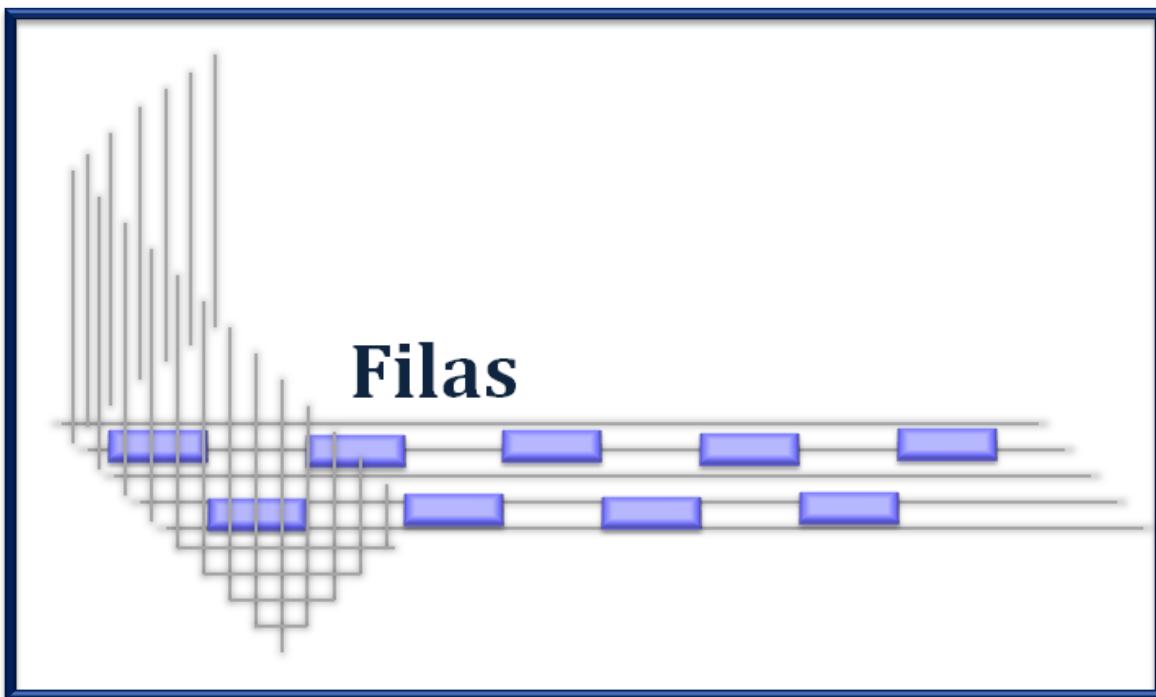
```
int ContaElementos(TipoLista *lista);
```

3) Considerando a seguinte estrutura referente ao cadastro de alunos em uma Universidade:

```
typedef struct aluno{  
    int RA;  
    char nome[30];  
    float P1, P2, T, APS;  
}Aluno;
```

Considerando uma lista duplamente encadeada, implemente as seguintes funções:

- a) Incluir um aluno no final da lista.
  - b) Inserir um aluno depois do enésimo elemento da lista. A posição passada como parâmetro.
  - c) Eliminar o enésimo nó da lista. A posição passada como parâmetro.
- 4) Implemente uma função que realize a ordenação de uma lista duplamente encadeada.
- 5) Faça uma função que receba como parâmetros uma lista dupla ordenada, de forma crescente, e um número a ser inserido. Manter a ordenação da lista.
- 6) Implemente uma função que retire os elementos com valores primos de uma lista simplesmente encadeada L1 e armazena-os em uma L2.



## 2. FILAS

Toda fila é formada a partir de uma lista, esta sendo constituída por um conjunto de elementos (nós), os quais são organizados de maneira que haja o **início** e o **fim** da fila, porém, a conexão entre os elementos se difere conforme a classificação da fila, a qual possui três configurações:

- **Fila Simples.** Formada a partir de uma lista simples, assim, dado o ponteiro de um nó não é possível ter acesso aos elementos adjacentes, pois mantém seus elementos em uma única direção, da *esquerda para direita*. Esta direção é determinada pelos campos do endereço seguinte de cada nó, os quais são ponteiros, que partem de um nó para seu sucessor na lista, como mostra a figura 44.

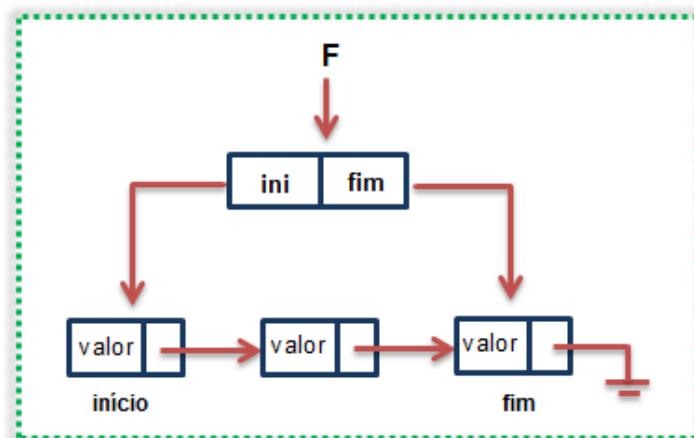


Figura 44. Representação de uma fila simples.

- **Fila Dupla.** Formada a partir de uma lista Dupla, assim, existe a possibilidade de acesso aos nós adjacentes, pois mantém seus elementos em duas direções, tanto da *esquerda para direita* quanto da *direita para esquerda*. Estas direções são determinadas pelos campos do endereço seguinte e pelos campos do endereço anterior de cada nó, os quais são ponteiros, que partem de um nó para seu sucessor e para seu anterior, respectivamente, como mostra a figura 45.

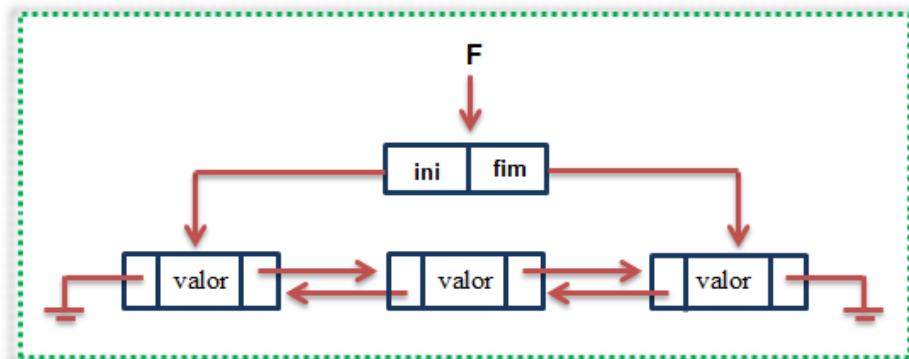


Figura 45. Representação de uma fila dupla.

- **Fila circular.** O último elemento tem como próximo o primeiro nó da fila, formando um ciclo, como mostra a figura 46. A fila pode ser representada por um ponteiro para um elemento inicial e final quaisquer.

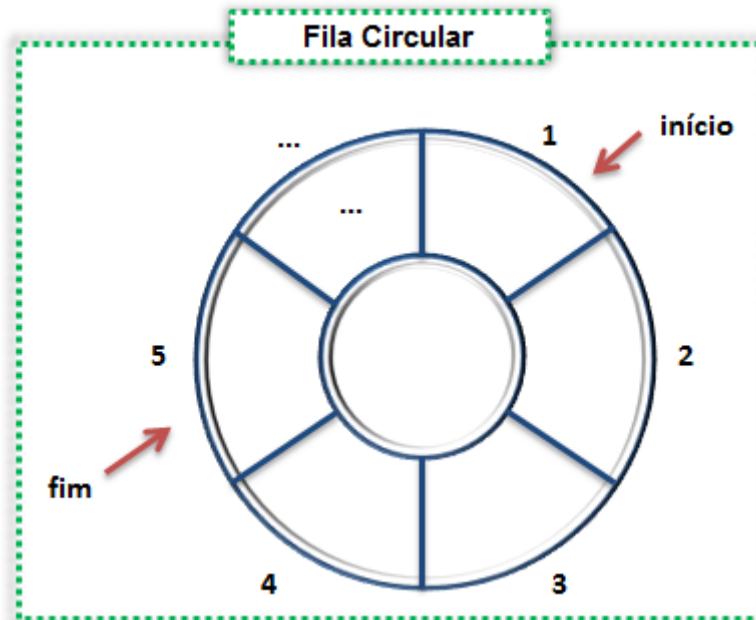


Figura 46. Representação de uma fila circular.

## 2.1. Fila com Lista Simplesmente Encadeada

### 2.1.1. Conceitos

Uma Fila Simples, formada a partir de uma Lista Simplesmente Encadeada (seção 1.1), é um conjunto de elementos (nós), os quais são organizados de maneira que haja o início e o fim da fila (Figura 47), assim, dois ponteiros, ambos apontando inicialmente para NULL, são criados para realizar esta função (figura 48):

- Ponteiro que referencia o *início* da fila denominado de **ini**.
- Ponteiro que referencia o *fim* da fila denominado de **fim**.

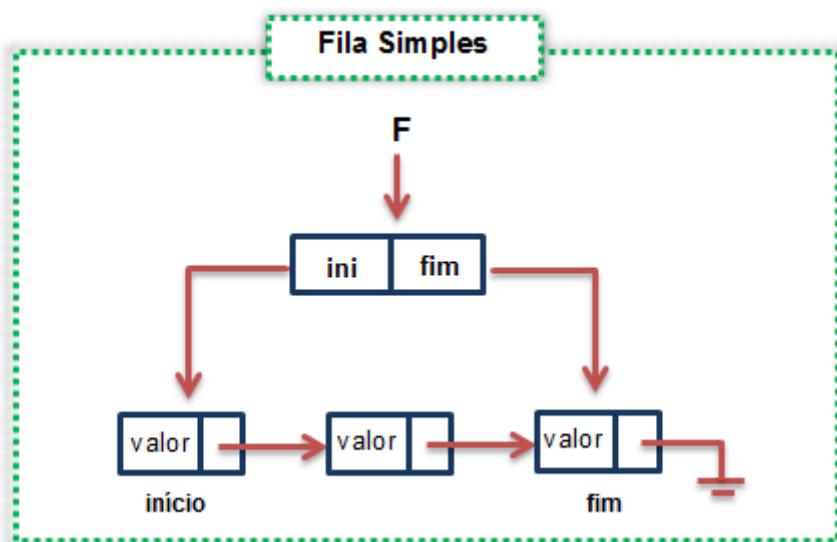


Figura 47. Representação de uma fila simples.

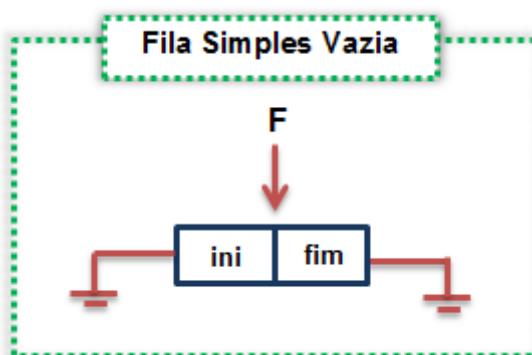


Figura 48. Representação de uma fila simples vazia.

Uma Fila Simples, como visto anteriormente, é formada a partir de uma lista Simples, assim, dado o ponteiro de um nó não é possível ter acesso aos elementos adjacentes, pois mantém seus elementos em uma única direção, da *esquerda para direita*. Esta direção é determinada pelos campos do endereço seguinte de cada nó, os quais são ponteiros, que partem de um nó para seu sucessor na lista.

**Observação**

Utiliza-se os conceitos de Lista Simplesmente Encadeada para a criação de uma Fila Simples. Contudo, como a estrutura a ser desenvolvida é referente a uma Fila não utiliza-se o ponteiro **L** que possui a função de apontar para o inicio da lista e, sim, dois ponteiros com a função de apontar para o inicio e fim da fila.

### **2.1.2. Criação e Manipulação**

Pode-se programar a criação da fila simples e estabelecer as devidas funcionalidades a ela, entre estas se tem:

- 2.1.2.1. Criação de um Tipo Abstrato de Dado (TAD).
- 2.1.2.2. Criar a fila.
- 2.1.2.3. Inserir elementos na fila.
- 2.1.2.4. Averiguar se a fila está vazia.
- 2.1.2.5. Imprimir os elementos.
- 2.1.2.6. Buscar um determinado elemento na fila.
- 2.1.2.7. Liberar todos os elementos.
- 2.1.2.8. Retirar um nó.

#### 2.1.2.1. Criação de um tipo abstrato de dado (TAD)

Criação de um novo tipo de dado, como mostra a figura 49.

```

typedef struct lista{
    int info;
    struct lista* prox;
}ListaSimp;

typedef struct fila{
    ListaSimp* ini;
    ListaSimp* fim;
}FilaSimp;

```

Figura 49. Estrutura de um novo tipo de dado do tipo FilaSimp.

**Descrição das funcionalidades** apresentadas na figura 49:

Os marcadores 1 e 2 se referem ao tipo abstrato de dado **Lista Simples**, possuindo as seguintes funcionalidades (figura 50):

1. Declarou-se a informação (info) do *campo de informação*, do tipo inteiro, o qual armazena o valor contido no nó.
2. Declarou-se o ponteiro do *campo do endereço seguinte* (prox), sendo do tipo struct lista.



Figura 50. Região declarada como info e outra como prox.

Os marcadores 3 e 4 se referem ao tipo abstrato de dado **Fila Simples**, como visto anteriormente, uma fila é composta por uma lista, assim, seus atributos serão dois ponteiros do tipo ListaSimp. Possuindo as seguintes funcionalidades (figura 51):

3. Declarou-se o ponteiro **ini**, do tipo ListaSimp, o qual representa o início da Fila.
4. Declarou-se o ponteiro **fim**, do tipo ListaSimp, o qual representa o fim da Fila.



Figura 51. Região declarada como ini e outra como fim.

#### 2.1.2.2. Criar a Fila

##### O que fazer? Dica...

Toda fila deve inicializar vazia!!!

A função denominada **criar\_fila**, apresentada na figura 52, possui as seguintes características:

- A função não possui parâmetros.
- O retorno é um ponteiro denominado de **F** alocado, do tipo **FilaSimp**.

```

FilaSimp* criar_fila(void){

    1     FilaSimp* F = (FilaSimp*)malloc(sizeof(FilaSimp));

    2     F->ini = NULL;
    3     F->fim = NULL;

    4     return F;
}

```

Figura 52. Função referente à atribuição NULL para a fila simples.

*Descrição das funcionalidades* apresentadas figura 52:

1. Utilizando o conceito de alocação dinâmica, aloca-se o ponteiro denominado de **F**, do tipo FilaSimp.
2. O **F -> ini** recebe o valor nulo, ou seja, aponta para NULL.
3. O **F -> fim** recebe o valor nulo.
4. Retorna o ponteiro criado denominado de **F**, este tendo acesso aos ponteiros **ini** e **fim**.

#### Observação

Na função principal (**main()**) a função para criar a fila deve ser chamada como:

**F = criar\_fila();**

#### 2.1.2.3. Inserir elementos na fila

##### O que fazer? Dicas...

- Deve alocar memória, de maneira dinâmica, para cada nó a ser inserido.
- O ponteiro **ini** e **fim**, do tipo ListaSimp, composição do ponteiro **F**, do tipo FilaSimp, apontam para o início e fim da fila, respectivamente.

\*\*\* Insere-se elementos sempre e somente no final da fila simples!

O que fazer? Dicas...

- Quando a fila está vazia o ponteiro **ini** ( $F->ini$ ) e o ponteiro **fim** ( $F->fim$ ), ambos referenciados pelo ponteiro **F**, apontam para o primeiro nó.
- Após a realização da primeira inserção o ponteiro **fim**, referenciado pelo ponteiro **F** ( $F->fim$ ), apontará para o nó a ser inserido. O ponteiro **ini** ( $F->ini$ ) não será manipulado.

A função denominada **inserir\_elementos**, apresentada na figura 53 possui as seguintes características:

- A função possui como parâmetros o ponteiro **F**, do tipo **FilaSimp**, que referencia o ponteiro **ini** e **fim**, ambos do tipo **ListaSimp**, e o valor a ser inserido, este do tipo inteiro, no campo de informação.
- A função é do tipo **void**, assim, nada retorna. As atribuições são realizadas dentro da função.

```

void inserir_elementos(FilaSimp* F, int valor){

    1     ListaSimp* novo = (ListaSimp*)malloc(sizeof(ListaSimp));

    2     novo -> info = valor;
    3     novo -> prox = NULL;

    4     if (F -> fim != NULL)
    5         F -> fim -> prox = novo;
    6     else
    5         F -> ini = novo;
    6     F -> fim = novo;
    }
```

Figura 53. Função referente à inserir elementos na fila simples.

*Descrição das funcionalidades* apresentadas figura 53:

1. Utilizando o conceito de alocação dinâmica, aloca-se o ponteiro denominado de **novo**, do tipo **ListaSimp**.
2. O *novo -> info* recebe um valor, do tipo inteiro, fornecido por meio do parâmetro da função, como pode ser observado na figura 54.
3. O *novo->prox* aponta para o valor nulo (NULL).

4. Se o ponteiro fim, referenciado pelo ponteiro F, for diferente de NULL (fila com elemento(s)) o ponteiro fim, referenciado pelo ponteiro F, aponta para o último elemento, deste o ponteiro prox aponta para o novo nó.
5. Senão significa que a fila está vazia, assim, o ponteiro ini, referenciado pelo ponteiro F, aponta para o novo nó, este sendo o primeiro a ser inserido. A condicional *else* será realizada somente uma única vez, como visto, para a inserção do primeiro elemento.
6. Independente da condicional, if ou else, o ponteiro fim, referenciado pelo ponteiro F, aponta para o novo elemento.

Na função principal (**main()**) deve constar (figura 54):

```
main(void){
    1     FilaSimp* F;
    2     F = criar_fila();
    3     inserir_elementos(F, <valor a ser inserido>);
}
```

Figura 54. Função principal chamando a função criar\_fila e inserir\_elementos na fila simples.

*Descrição das funcionalidades* apresentadas na figura 54:

1. Cria-se um ponteiro F, do tipo FilaSimp, o qual referenciará o início (F -> ini) e o fim da fila (F -> fim).
2. Chamar a função criar\_fila para atribuir o primeiro valor a fila, o qual é nulo. O ponteiro F declarado recebe o retorno desta função.
3. Chamar a função inserir\_elementos passando como parâmetros o ponteiro F e o valor a ser inserido. Como visto não há retorno desta função.

Para melhor entendimento, graficamente, realiza-se um exemplo a seguir, utilizando o conceito *teste de mesa*.

### Exemplo

Inserir os seguintes elementos 6, 1 e 5.

**1º PASSO:** Na função principal colocar os seguintes comandos apresentados na figura 55:

```

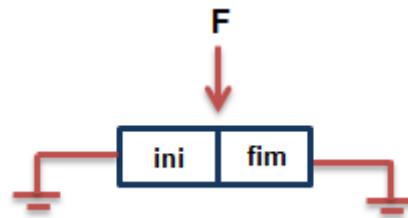
main(void){
    FilaSimp* F;
    F = criar_fila();
    inserir_elementos(F, 6);
    inserir_elementos(F, 1);
    inserir_elementos(F, 5);
}

```

Figura 55. Comandos que devem ser colocados na função principal.

**2º PASSO:** Quando inicializada a função principal o ponteiro F, tipo FilaSimp, é declarado.

**3º PASSO:** O ponteiro F recebe o retorno da função criar\_fila, o qual será NULL para os ponteiros fim e ini.



**4º PASSO:** a função inserir\_elementos(F, 6) realiza os seguintes procedimentos:

- Aloca-se um novo nó.



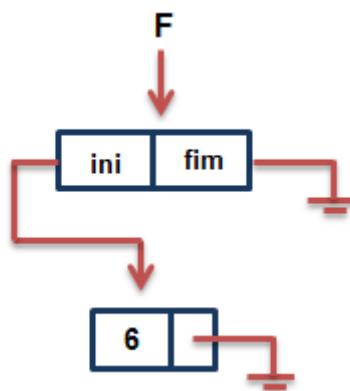
- Este nó recebe o valor 6 no campo de informação (novo -> info).



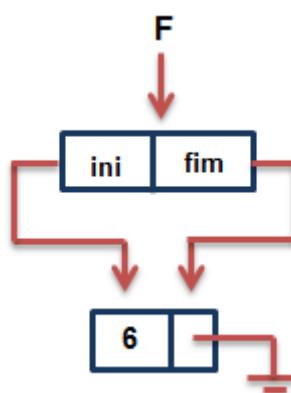
- O campo do endereço seguinte do nó criado (novo -> prox) aponta para NULL.



- O ponteiro fim, referenciado pelo ponteiro F, é diferente de NULL? NÃO, então, ainda não há elemento na fila. Entra na condicional else e, assim, o ponteiro ini, referenciado pelo ponteiro F (F -> ini) aponta para o novo nó.



- O ponteiro  $fim$ , referenciado pelo ponteiro  $F$ , aponta para o novo nó.



**5º PASSO:** a função inserir\_elementos( $F$ , 1) realiza os seguintes procedimentos:

- Aloca-se um novo nó.



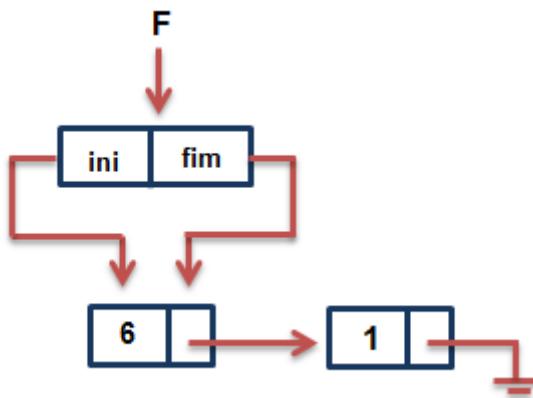
- Este nó recebe o valor 1 no campo de informação ( $novo \rightarrow info$ ).



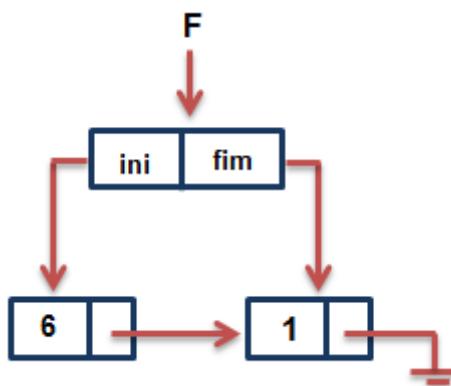
- O campo do endereço seguinte do nó criado ( $novo \rightarrow prox$ ) aponta para NULL.



- O ponteiro  $fim$ , referenciado pelo ponteiro  $F$ , é diferente de NULL? SIM, então, há elemento na fila. Assim, o ponteiro  $prox$  do elemento apontado pelo ponteiro  $fim$ , este referenciado pelo ponteiro  $F$  ( $F \rightarrow fim \rightarrow prox$ ), aponta para o novo nó.



- O ponteiro fim, referenciado pelo ponteiro F, aponta para o novo nó.



**6º PASSO:** a função inserir\_elementos(F, 5) realiza os seguintes procedimentos:

- Aloca-se um novo nó.



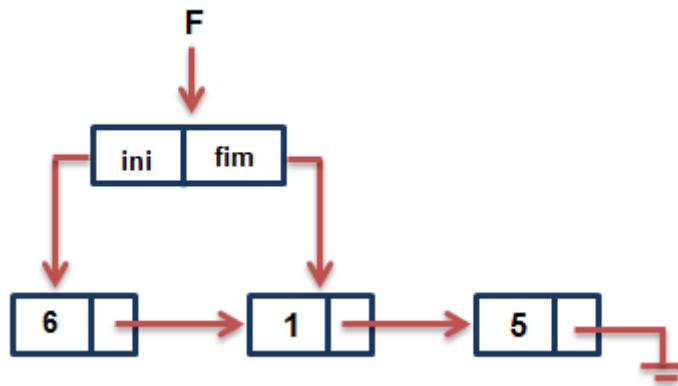
- Este nó recebe o valor 5 no campo de informação (*novo -> info*).



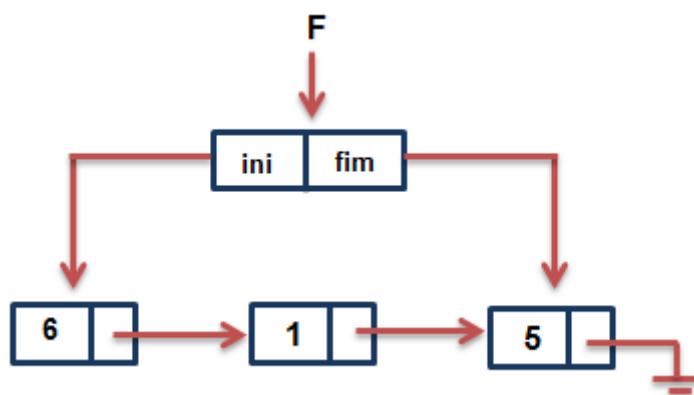
- O campo do endereço seguinte do nó criado (*novo -> prox*) aponta para NULL.



- O ponteiro fim, referenciado pelo ponteiro F, é diferente de NULL? SIM, então, há elemento na fila. Assim, o ponteiro prox do elemento apontado pelo ponteiro fim, este referenciado pelo ponteiro F (F -> fim -> prox), aponta para o novo nó.



- O ponteiro fim, referenciado pelo ponteiro F, aponta para o novo nó.



#### 2.1.2.4. Averiguar se a lista está vazia

**O que fazer? Dicas...**

Verificar se o inicio da fila aponta para NULL.

A função denominada **fila\_vazia**, apresentada na figura 56, possui as seguintes características:

- A função possui como parâmetro o ponteiro F.
- O retorno é do tipo **booleano**, retornando valor 1 (verdadeiro) quando a fila estiver vazia, caso contrário, retorna valor 0 (falso).

```
int fila_vazia(FilaSimp* F){
    return (F -> ini == NULL);
}
```

Figura 56. Função que verifica se a fila simples está vazia.

### 2.1.2.5. Imprimir os elementos

#### O que fazer? Dicas...

Percorrer a fila imprimindo os valores de cada nó.

Não manipular tanto o ponteiro ini quanto o ponteiro fim para não perder a referência do inicio e final da fila.

A função denominada **imprimir**, apresentada na figura 57, possui as seguintes características:

- A função possui como parâmetro o ponteiro F, o qual referencia os ponteiros ini e fim.
- O retorno é do tipo void, retornando na tela os valores contidos na fila.

```
void imprimir (FilaSimp* F){

    1     ListaSimp* q;

    2     if(fila_vazia(F))
    2         printf("\nFila vazia!\n");
    3     else{
    3         for (q = F -> ini; q != NULL; q = q -> prox) {
    3             printf("%d ", q -> info);
    }
    printf("\n\n");
}
```

Figura 57. Função imprimir.

*Descrição das funcionalidades* apresentadas na figura 57:

1. Declara-se um ponteiro auxiliar denominado de q, do tipo ListaSimp.
2. Dentro da condicional chama-se a função fila\_vazia, caso retorne o valor 1 a fila está vazia e, assim, imprime-se “Fila vazia!”.
3. Senão entra no laço for:
  - Utiliza-se um ponteiro auxiliar denominado de q, o qual aponta para onde o ponteiro ini, referenciado pelo ponteiro F, aponta, ou seja, para o primeiro nó.
  - Enquanto o ponteiro q for diferente de NULL ainda há elemento(s) na fila.

- O incremento do ponteiro q é representado por  $q = q \rightarrow prox.$
- A impressão é baseada no valor armazenado na região do campo de informação (info).

### Observação

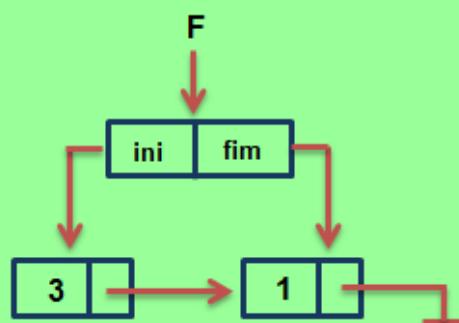
Na função principal (`main()`) a função para imprimir a fila deve ser chamada como:

`imprimir(F);`

Para melhor entendimento, graficamente, realiza-se um exemplo a seguir, utilizando o conceito *teste de mesa*.

### Exemplo

Considerando a seguinte fila:



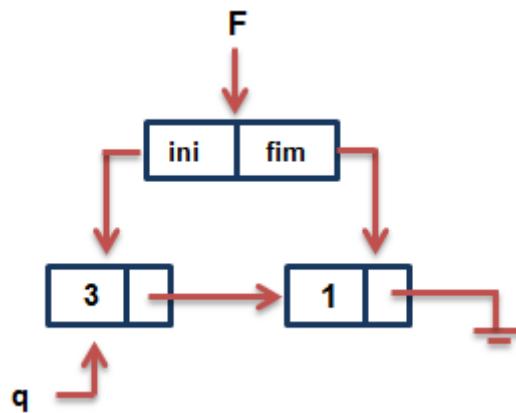
**1º PASSO:** Declara-se um ponteiro auxiliar denominado de q.

**2º PASSO:** Entra na condicional if.

- Chama-se a função `fila_vazia`, a fila está vazia? NÃO!! Pois o ponteiro ini, referenciado pelo ponteiro F, aponta para o nó de valor 3, ou seja, é diferente de NULL.

**3º PASSO:** Entra na condicional else. Percorre-se o laço for.

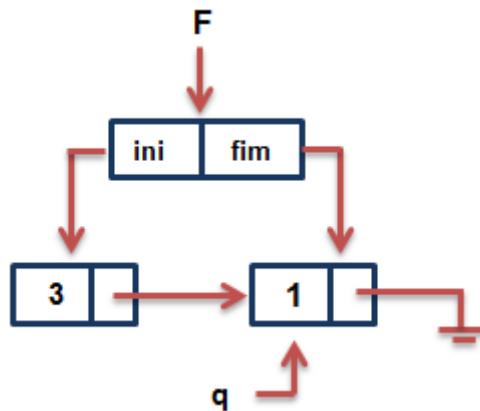
- O ponteiro q aponta para onde o ponteiro ini, referenciado pelo ponteiro F, aponta, ou seja, para o primeiro elemento da fila.



- O ponteiro  $q$  é diferente de NULL, ou seja, não aponta para NULL? SIM, assim imprime-se o valor 3 contido no elemento apontado pelo ponteiro  $q$  ( $q \rightarrow \text{info}$ ).

**4º PASSO:** Retorna ao laço for:

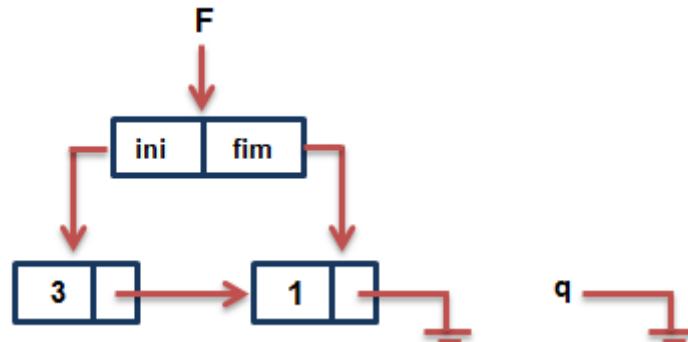
- O incremento do ponteiro  $q$  é dado por  $q = q \rightarrow \text{prox}$ , onde  $q \rightarrow \text{prox}$  é o elemento de valor 1. Assim, o ponteiro  $q$  aponta para este nó.



- O ponteiro  $q$  é diferente de NULL, ou seja, não aponta para NULL? SIM, assim imprime-se o valor 1 contido no elemento apontado pelo ponteiro  $q$  ( $q \rightarrow \text{info}$ ).

**5º PASSO:** Retorna ao laço for:

- O incremento do ponteiro  $q$  é dado por  $q = q \rightarrow \text{prox}$ , onde  $q \rightarrow \text{prox}$  possui valor nulo. Assim, o ponteiro  $q$  aponta para NULL.



- O ponteiro q é diferente de NULL, ou seja, não aponta para NULL? NÃO, ou seja, imprimiram-se todos os elementos contidos na fila.

**6º PASSO:** Termina o laço, assim, encerra-se a impressão!!

#### 2.1.2.6. Buscar um determinado elemento na fila

##### O que fazer? Dicas...

- Percorrer a fila comparando o valor que o nó possui com o valor a ser buscado.
- Caso encontre, retorna o nó que contém o valor. Caso contrário, retorna NULL.

A função denominada **buscar\_elemento**, apresentada na figura 58, possui as seguintes características:

- A função possui como parâmetro o valor a ser procurado e o ponteiro F, o qual possui acesso aos ponteiros ini e fim.
- O retorno é do tipo FilaSimp. Caso o elemento seja encontrado o ponteiro q aponta para ele, assim, retorna este nó, caso contrário, retorna o valor nulo (NULL).

```
FilaSimp* buscar_elemento (FilaSimp* F, int valor){
```

```
    ListaSimp* q;
```

```
    for(q = F -> ini; q != NULL; q = q -> prox) {
```

```
        if(q -> info == valor)
```

```
            return q;
```

```
}
```

```
    return NULL;
```

```
}
```

Figura 58. Função buscar elemento na fila simples.

*Descrição das funcionalidades* apresentadas na figura 58:

1. O ponteiro ini, referenciado pelo ponteiro F, aponta para o primeiro nó da fila, este não deve ser manipulado para não perder a referência, então, utiliza-se um

ponteiro auxiliar denominado de  $q$ , o qual aponta para onde  $F \rightarrow ini$  aponta, ou seja, para o primeiro nó;

2. Enquanto o ponteiro  $q$  for diferente de NULL ainda há elemento(s) na fila.
3. O incremento do ponteiro  $q$  é representado por  $q = q \rightarrow prox$ .
4. Se o campo de informação do nó apontado pelo ponteiro  $q$  ( $q \rightarrow info$ ) for igual ao valor procurado retorna-se o ponteiro  $q$ , o qual terá acesso ao campo de informação (info) e ao campo do endereço seguinte (prox), já que  $q$  é do tipo ListaSimp.
5. Caso não encontrado, retorna-se o valor nulo (NULL).

#### Observação

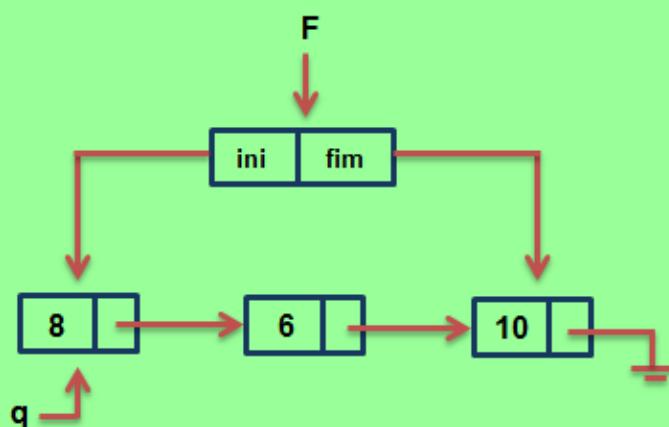
Na função principal (**main()**) a função buscar deve ser chamada como:

```
ListaSimp* resultado = buscar_elemento (F, <valor a ser procurado>);
```

Para melhor entendimento, graficamente, realiza-se um exemplo a seguir, utilizando o conceito *teste de mesa*.

#### Exemplo

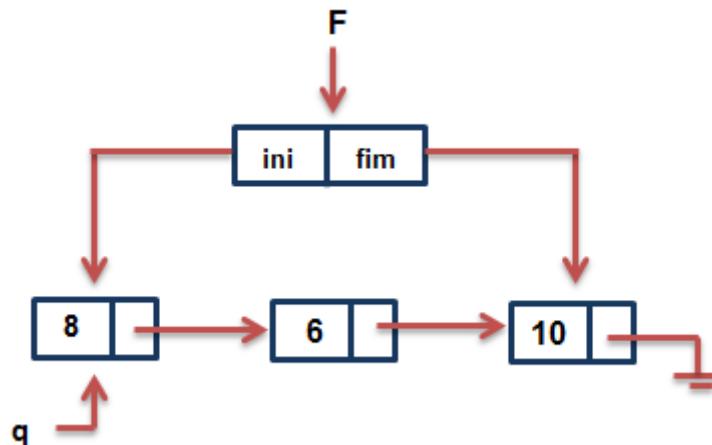
Considerando a seguinte fila e o número 8 a ser procurado:



**1º PASSO:** Declara-se um ponteiro auxiliar denominado de  $q$ .

**2º PASSO:** Percorrer o laço for:

- O ponteiro  $q$  aponta para onde o ponteiro  $ini$ , referenciado pelo ponteiro  $F$  ( $F \rightarrow ini$ ) aponta, ou seja, para o primeiro nó.



- O ponteiro *q* é diferente de NULL? SIM!! O ponteiro *q* aponta para o nó de valor 8.
- Entra na condicional if.
- O ponteiro *q* que na região *info* (*q->info*) vale 8, é igual ao valor procurado que é 8, passado como parâmetro? SIM!!

**3º PASSO:** Retorna o nó apontado pelo ponteiro *q*, o qual possui valor 8.

#### 2.1.2.7. Liberar todos os elementos

##### O que fazer? Dicas...

- Percorrer a fila desalocando cada elemento.
- Utilizar um ponteiro auxiliar que aponte para o inicio da fila, e ainda, utilizar um outro ponteiro que aponte para o nó seguinte ao ponteiro auxiliar criado.

A função denominada **liberar\_fila**, apresentada na figura 59, possui as seguintes características:

- A função possui como parâmetro o ponteiro *F*.
- Não há retorno para a função principal, é do tipo void.

```

void liberar_fila (FilaSimp* F){

    1      ListaSimp* q = F -> ini;

    2      while (q != NULL) {

        3          ListaSimp* t = q -> prox;
        4          free(q);
        5          q = t;
        6      }
    }      free(F);
}

```

Figura 59. Função liberar a fila.

*Descrição das funcionalidades* apresentadas na figura 59:

1. Declara-se um ponteiro auxiliar denominado de q, do tipo ListaSimp, o qual aponta para onde o ponteiro ini, referenciado pelo F, aponta, ou seja, para o primeiro elemento da fila.
2. No laço while, enquanto o ponteiro q for diferente de NULL ainda há elemento(s) na fila, então, executa-se os comandos encontrados dentro do laço.
3. Utiliza-se um ponteiro auxiliar denominado de t que aponta para o nó a frente do ponteiro q ou para NULL, caso o ponteiro q aponte para o último nó.
4. Desaloca o nó apontado pelo ponteiro q.
5. O ponteiro q aponta para onde o ponteiro t aponta.
6. Após desalocar todos os elementos desaloca-se o ponteiro F.

#### Observação

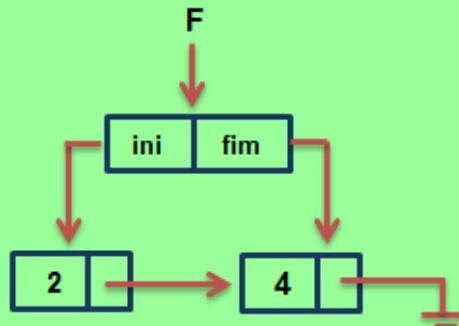
Na função principal (**main()**) a função liberar deve ser chamada como:

**liberar\_fila(F);**

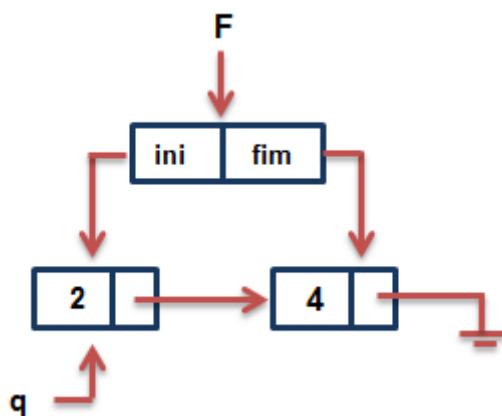
Para melhor entendimento, graficamente, realiza-se um exemplo a seguir, utilizando o conceito *teste de mesa*.

Exemplo

Considerando a seguinte fila:

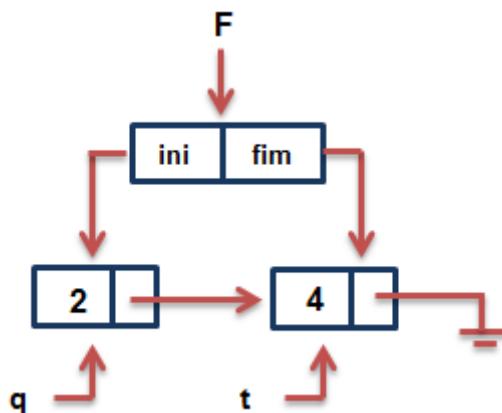


**1º PASSO:** O ponteiro q aponta para onde o ponteiro ini, referenciado pelo ponteiro F, aponta, ou seja, para o início da fila.

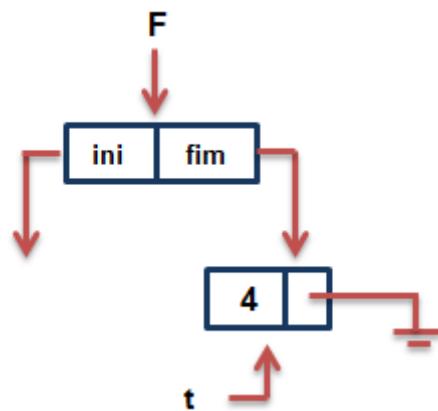


**2º PASSO:** Entra no laço while.

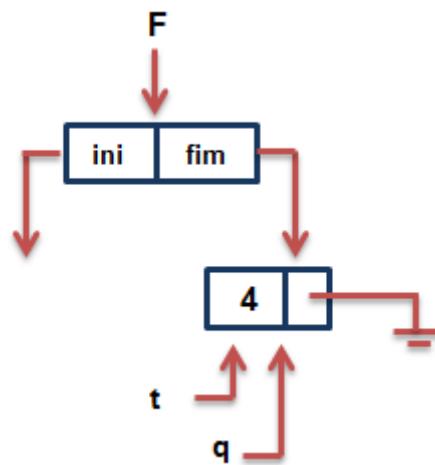
- O ponteiro q é diferente de NULL? SIM!! O ponteiro q aponta para o nó de valor 2.
- O ponteiro t aponta para o próximo nó do ponteiro q ( $q->prox$ ). No caso, apontará para o nó de valor 4.



- Desaloca o nó apontado pelo ponteiro q.

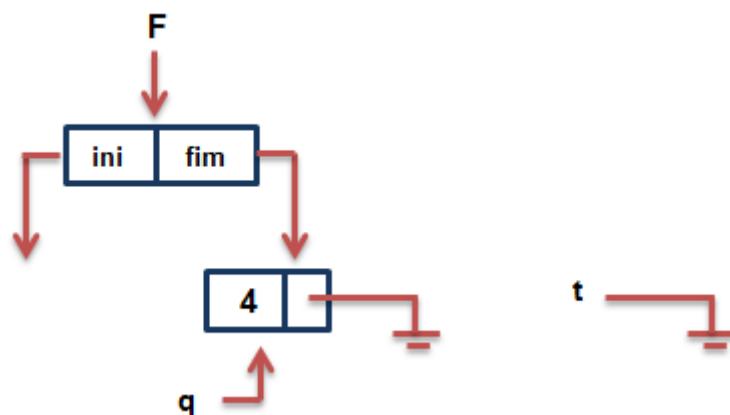


- O ponteiro q aponta para onde o ponteiro t aponta.

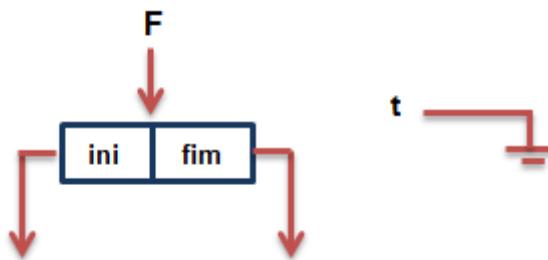


**3º PASSO:** Retorna ao laço while.

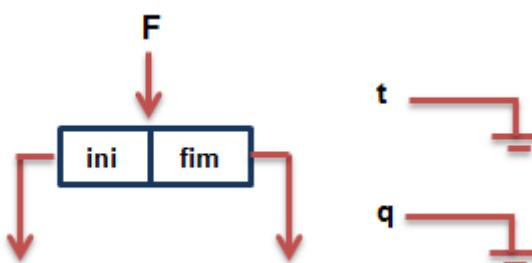
- O ponteiro q é diferente de NULL? SIM!! O ponteiro q aponta para o nó de valor 4.
- O ponteiro t aponta para o próximo nó do ponteiro q ( $q->prox$ ). No caso, apontará para NULL.



- Desaloca o nó apontado pelo ponteiro q.



- O ponteiro q aponta para onde o ponteiro t aponta.



**4º PASSO:** Retorna ao laço while.

- O ponteiro q é diferente de NULL? NÃO!! O ponteiro q aponta para NULL.
- Terminam as iterações do laço while.

**5º PASSO:** Desaloca o ponteiro F. Assim, liberou toda a fila.

#### 2.1.2.8. Retirar um nó

##### O que fazer? Dicas...

- Utilizar um ponteiro auxiliar, o qual deve apontar para o primeiro elemento da fila.
- O ponteiro que referencia o inicio da fila deve apontar para o segundo elemento, o qual se tornará o primeiro após a retirada do nó.

\*\*\* Em uma fila simples retira-se sempre e somente do inicio!!!

A função denominada **retirar\_elemento**, apresentada na figura 60, possui as seguintes características:

- A função possui como parâmetro o ponteiro **F**, do tipo **FilaSimp**.

- O retorno é do tipo inteiro, retornando o valor contido no campo de informação (*info*) do nó a ser removido.

```

int retirar_elemento (FilaSimp* F){

    1      ListaSimp* t;
    2      int v;

    3      if(fila_vazia(F)){
    3          printf("\nFila vazia!\n");
    3          exit(1);
    }

    4      t = F -> ini;
    5      v = t -> info;
    6      F -> ini = t -> prox;

    7      if(F -> ini == NULL)
    7          F -> fim = NULL;

    8      free(t);
    9      return v;
}

```

Figura 60. Função para retirar o primeiro elemento da fila.

*Descrição das funcionalidades* apresentadas na figura 60:

1. O ponteiro F possui acesso aos dois ponteiros ini e fim. Somente o ponteiro ini será relevante (retira-se sempre e somente o elemento que se encontra no início da fila), assim, declara-se um ponteiro auxiliar denominado de t para que não ocorra a perda da referência do ponteiro ini, referenciado pelo ponteiro F.
2. Declara-se uma variável, do tipo inteiro, denominada de v. Esta armazenará o valor do primeiro nó que será retirado.
3. Chama-se a função fila\_vazia, se estiver vazia apresentará uma mensagem na tela e finalizará a chamada da função retirar\_elemento.
4. O ponteiro t aponta para o início da fila ( $F \rightarrow ini$ ), ou seja, para o primeiro elemento.
5. A variável v armazena o valor do campo de informação do nó apontado pelo ponteiro t.
6. O ponteiro ini, referenciado pelo ponteiro F, aponta para o nó seguinte em relação ao ponteiro t, ou seja, para o segundo elemento, ou para NULL, caso a fila tenha somente um nó antes da chamada da função retirar\_elemento.

7. Se o ponteiro ini, referenciado pelo ponteiro F, alterado no comando 6, for igual a NULL significa que há um único elemento na fila, assim, o ponteiro fim, referenciado pelo ponteiro F, deverá deixar de apontar para o nó e apontar para NULL.
8. Desaloca o nó apontado pelo ponteiro t.
9. Retorna o valor contido no campo de informação do elemento removido.

**Observação**

Na função principal (`main()`) a função retirar deve ser chamada como:

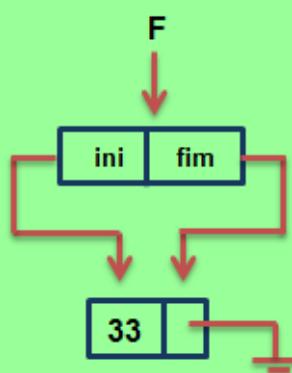
```
int valor_retirado = retirar_elemento(F);
```

Para melhor entendimento, graficamente, realiza-se dois exemplos a seguir.

1. A fila possui um único elemento.
2. A fila possui quatro elementos.

**Exemplo 1**

Considerando a seguinte fila:

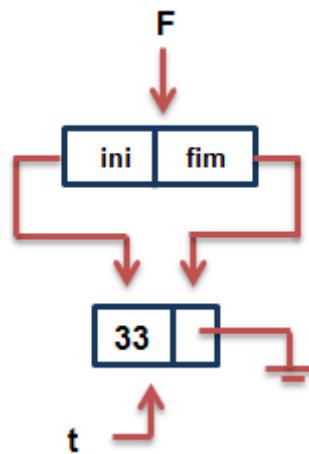


**1º PASSO:** Declaração do ponteiro t e da variável v.

**2º PASSO:** Entra na condicional if.

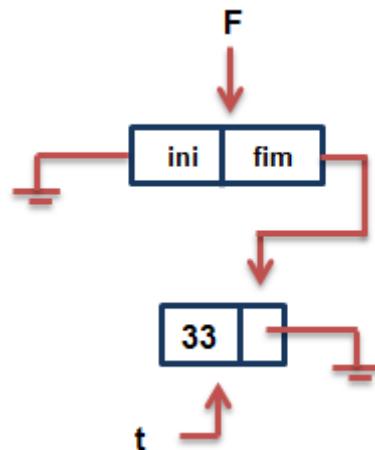
- Chama-se a função fila\_vazia, contudo retorna 0, ou seja, a fila não está vazia, possui um elemento com valor 33.

**3º PASSO:** O ponteiro t aponta para o primeiro elemento, este apontado pelo ponteiro ini, referenciado pelo ponteiro F ( $F \rightarrow ini$ ).



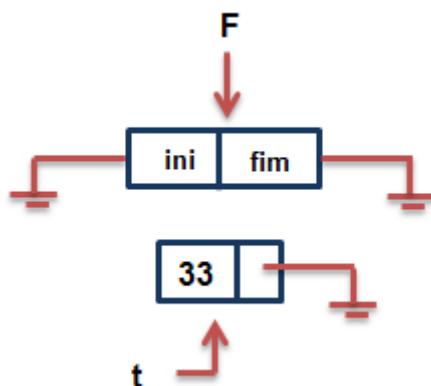
**4º PASSO:** A variável v armazena o valor 33 contido no campo de informação do nó apontado pelo ponteiro t ( $t \rightarrow \text{info}$ ).

**5º PASSO:** O ponteiro ini, referenciado pelo ponteiro F, aponta para o nó seguinte em relação ao ponteiro t ( $t \rightarrow \text{prox}$ ), ou seja, para NULL.

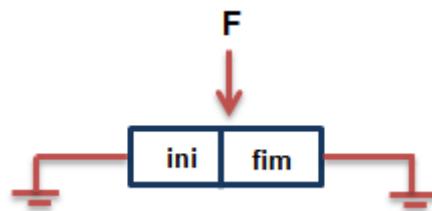


**6º PASSO:** Entra na condicional if.

- O ponteiro ini, referenciado pelo ponteiro F, é igual a NULL? SIM!! Assim, o ponteiro fim, referenciado pelo ponteiro F ( $F \rightarrow \text{fim}$ ), aponta para NULL.



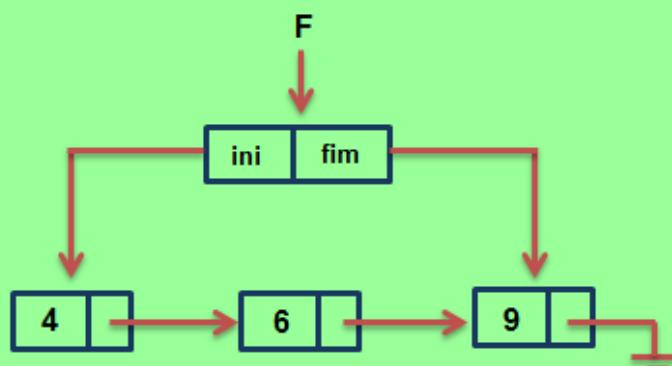
. **7º PASSO:** Desaloca o nó apontado pelo ponteiro t.



**8º PASSO:** Retorna o valor 33 contido na variável v.

### Exemplo 2

Considerando a seguinte fila:

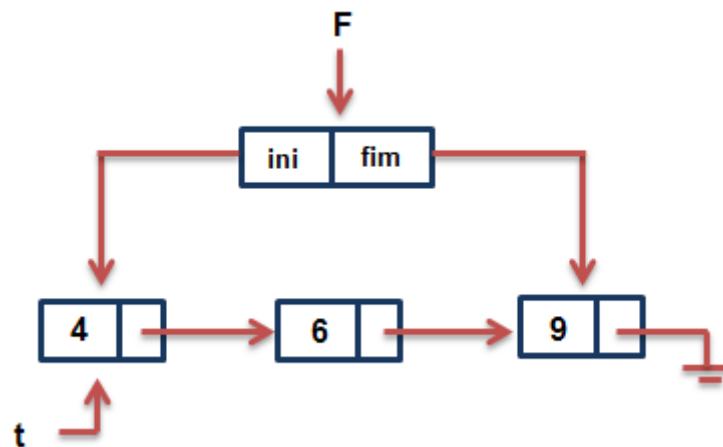


**1º PASSO:** Declaração do ponteiro t e da variável v.

**2º PASSO:** Entra na condicional if.

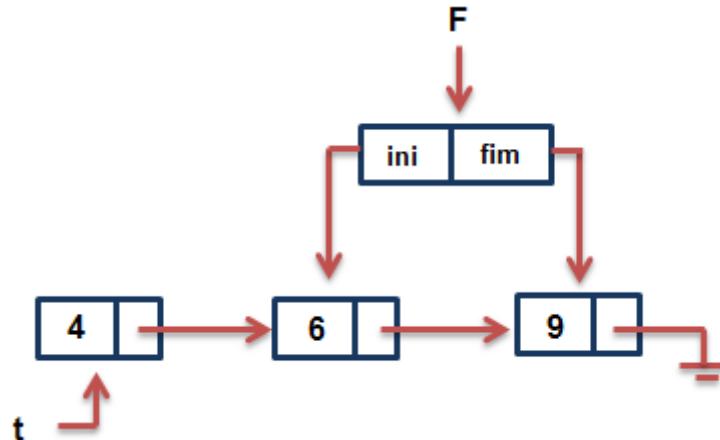
- Chama-se a função fila\_vazia, contudo retorna 0, ou seja, a fila não está vazia, possui um três elementos.

**3º PASSO:** O ponteiro t aponta para o primeiro elemento, este apontado pelo ponteiro ini, referenciado pelo ponteiro F ( $F \rightarrow ini$ ).



**4º PASSO:** A variável v armazena o valor 4 contido no campo de informação do nó apontado pelo ponteiro t ( $t \rightarrow \text{info}$ ).

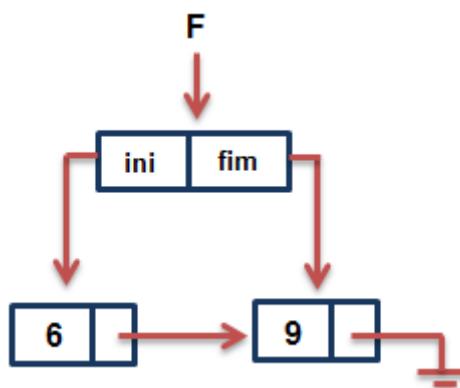
**5º PASSO:** O ponteiro ini, referenciado pelo ponteiro F, aponta para o nó seguinte em relação ao ponteiro t, ou seja, para o nó de valor 6.



**6º PASSO:** Entra na condicional if.

- O ponteiro ini, referenciado pelo ponteiro F, é igual a NULL? NÃO!!

**7º PASSO:** Desaloca o nó apontado pelo ponteiro t.



**8º PASSO:** Retorna o valor 4 contido na variável v.

## 2.2. Fila com Lista Duplamente Encadeada

### 2.2.1. Conceitos

Uma Fila Dupla, formada a partir de uma Lista Duplamente Encadeada (seção 1.2), é um conjunto de elementos (nós), os quais são organizados de maneira que haja o início e o fim da fila (Figura 61). Dois ponteiros, ambos apontando inicialmente para NULL, são criados para realizar esta função (figura 62):

- Ponteiro que referencia o *início* da fila denominado de **ini**.
- Ponteiro que referencia o *fim* da fila denominado de **fim**.

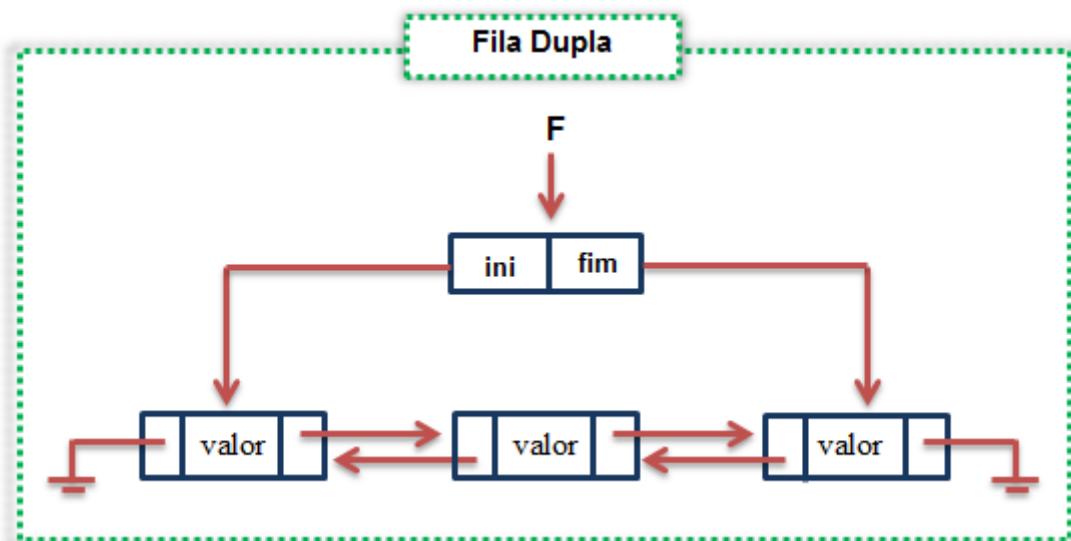


Figura 61. Representação de uma fila dupla.

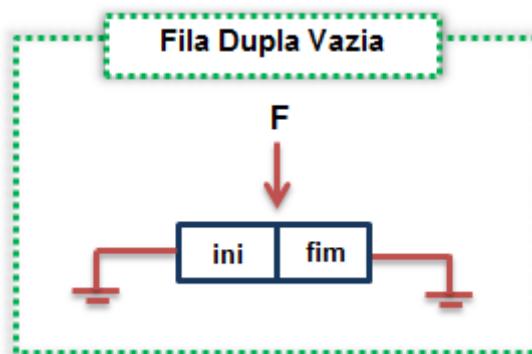


Figura 62. Representação de uma fila dupla vazia.

Uma Fila Dupla, como visto anteriormente, é formada a partir de uma lista Dupla, assim, existe a possibilidade de acesso aos nós adjacentes, pois mantém seus elementos em duas direções, tanto da *esquerda para direita* quanto da *direita para esquerda*.

Estas direções são determinadas pelos campos do endereço seguinte e pelos campos do endereço anterior de cada nó, os quais são ponteiros, que partem de um nó para seu sucessor e para seu anterior, respectivamente.

**Observação**

Utiliza-se os conceitos de Lista Duplamente Encadeada para a criação de uma Fila Dupla. Contudo, como a estrutura a ser desenvolvida é referente a uma Fila não utiliza-se o ponteiro **L** que possui a função de apontar para o inicio da lista e, sim, dois ponteiros com a função de apontar para o inicio e fim da fila.

### **2.2.2. Criação e Manipulação**

Pode-se programar a criação da fila dupla e estabelecer as devidas funcionalidades a ela, entre estas se tem:

- 2.2.2.1. Criação de um Tipo Abstrato de Dado (TAD).
- 2.2.2.2. Criar a fila.
- 2.2.2.3. Inserir elementos.
- 2.2.2.4. Imprimir os elementos.
- 2.2.2.5. Averiguar se a fila está vazia.
- 2.2.2.6. Buscar um determinado elemento na fila.
- 2.2.2.7. Liberar todos os elementos.
- 2.2.2.8. Retirar um nó.

#### **2.2.2.1. Criação de um tipo abstrato de dado (TAD)**

Criação de um novo tipo de dado, como mostra a figura 63.

```

typedef struct lista{
    int info;
    struct lista* prox;
    struct lista* ant;
}ListaDupla;

typedef struct fila{
    ListaDupla* ini;
    ListaDupla* fim;
}FilaDupla;

```

Figura 63. Estrutura de um novo tipo de dado do tipo FilaDupla.

**Descrição das funcionalidades** apresentadas na figura 63:

Os marcadores 1,2 e 3 se referem ao tipo abstrato de dado **Lista Dupla**, possuindo as seguintes funcionalidades (figura 64):

1. Declarou-se a informação (info) do *campo de informação*, do tipo inteiro, o qual armazena o valor contido no nó.
2. Declarou-se o ponteiro do *campo do endereço seguinte* (prox), sendo do tipo struct lista.
3. Declarou-se o ponteiro do *campo do endereço anterior* (ant), sendo do tipo struct lista.



Figura 64. Regiões declaradas como info, prox e ant.

Nos marcadores 4 e 5 se referem ao tipo abstrato de dado **Fila Dupla**, como visto anteriormente, uma fila é composta por uma lista, assim, seus atributos serão dois ponteiros do tipo ListaDupla, como mostra a figura 65. Possuindo as seguintes funcionalidades:

4. Declarou-se o ponteiro **ini**, do tipo ListaDupla, o qual representa o início da Fila.
5. Declarou-se o ponteiro **fim**, do tipo ListaDupla, o qual representa o fim da Fila.



Figura 65. Região declarada como ini e outra como fim.

#### 2.2.2.2. Criar a Fila

O que fazer? Dica...

Toda fila deve inicializar vazia!!!

A função denominada **criar\_fila**, apresentada na figura 66, possui as seguintes características:

- A função não possui parâmetros.
- O retorno é o ponteiro denominado de **F** alocado, do tipo **FilaDupla**.

```

FilaDupla* criar_fila(void){

    1     FilaDupla* F = (FilaDupla*)malloc(sizeof(FilaDupla));

    2     F->ini = NULL;
    3     F->fim = NULL;

    4     return F;
}

```

Figura 66. Função referente à atribuição NULL para a fila dupla.

*Descrição das funcionalidades* apresentadas figura 66:

1. Utilizando o conceito de alocação dinâmica, aloca-se o ponteiro denominado de **F**, do tipo FilaDupla.
2. O **F -> ini** recebe o valor nulo, ou seja, aponta para NULL.
3. O **F -> fim** recebe o valor nulo
4. Retorna o ponteiro criado denominado de **F**, este tendo acesso aos ponteiros **ini** e **fim**.

#### Observação

Na função principal (**main()**) a função para criar a fila deve ser chamada como:

**F = criar\_fila();**

#### 2.2.2.3. Inserir elementos

##### O que fazer? Dicas...

- Deve alocar memória, de maneira dinâmica, para cada nó a ser inserido.
- O ponteiro **ini** e **fim**, do tipo ListaDupla, composição do ponteiro **F**, do tipo FilaDupla, apontam para o inicio e fim da fila, respectivamente.

\*\*\* Insere-se elementos somente no inicio ou no fim!

Tanto a função **inserir\_Inicio** quanto a função **inserir\_Fim**, apresentadas nas figura 67 e 70, possuem as seguintes características:

- A função possui como parâmetros o ponteiro F, do tipo FilaDupla, que referencia o ponteiro ini e fim, ambos do tipo ListaDupla, e o valor a ser inserido, este do tipo inteiro, no campo de informação.
- A função é do tipo void, assim, nada retorna. As atribuições são realizadas dentro da função.

#### 2.2.2.3.1. Inserir no fim da Fila Dupla

O código referente a função denominada **inserir\_Fim** é apresentado na figura 67.

```
void inserir_Fim(FilaDupla* F, int valor){
    1     ListaDupla* novo = (ListaDupla*)malloc(sizeof(ListaDupla));
    2     novo->info = valor;
    3     novo->prox = NULL;

    4     if (F->fim != NULL){
    4         F->fim->prox = novo;
    4         novo->ant = F->fim;
    5     }else{
    5         F->ini = novo;
    5         novo->ant = NULL;
    6     }
    6     F->fim = novo;
}
```

Figura 67. Função referente à inserir elementos no final da fila dupla.

*Descrição das funcionalidades* apresentadas figura 67:

1. Utilizando o conceito de alocação dinâmica, aloca-se o ponteiro denominado de **novo** do tipo ListaDupla.
2. O *novo -> info* recebe um valor, do tipo inteiro, fornecido por meio do parâmetro da função, como pode ser observado na figura 68.
3. O *novo->prox* aponta para o valor nulo (NULL).

4. Se o ponteiro fim, referenciado pelo ponteiro F, for diferente de NULL (fila com elemento(s)) o ponteiro prox do nó apontado pelo ponteiro fim, referenciado pelo ponteiro F ( $F \rightarrow \text{fim} \rightarrow \text{prox}$ ) aponta para o novo nó, assim como, o ponteiro ant do nó criado aponta para onde o ponteiro fim, referenciado pelo ponteiro F, aponta, ou seja, para o último nó.
5. Senão significa que a fila está vazia, assim, o ponteiro ini, referenciado pelo ponteiro F, apontará para o novo nó, este sendo o primeiro a ser inserido e o ponteiro ant do nó criado aponta para NULL. A condicional *else* será realizada somente uma única vez, ou seja, para a inserção do primeiro elemento.
6. Independente da condicional, if ou else, o ponteiro fim, referenciado pelo ponteiro F, apontará para o novo elemento.

Na função principal (**main()**) deve constar (figura 68):

```
main(void){
    1     FilaDupla* F;
    2     F = criar_fila();
    3     inserir_Fim(F, <valor a ser inserido>);
}
```

Figura 8. Função principal chamando a função *criar\_fila* e *inserir\_Fim*.

*Descrição das funcionalidades* apresentadas na figura 68:

1. Cria-se um ponteiro F, do tipo *FilaDupla*, o qual referenciará o início ( $F \rightarrow \text{ini}$ ) e o fim da fila ( $F \rightarrow \text{fim}$ ).
2. Chamar a função *criar\_fila* para atribuir o primeiro valor a fila, o qual é nulo. O ponteiro F declarado recebe o retorno desta função.
3. Chamar a função *inserir\_Fim* passando como parâmetros o ponteiro F e o valor a ser inserido. Como visto não há retorno da função.

Para melhor entendimento, graficamente, realiza-se um exemplo a seguir, utilizando o conceito *teste de mesa*.

### Exemplo

Inserir os seguintes elementos 2, 8 e 5.

**1º PASSO:** Na função principal colocar os seguintes comandos apresentados na figura 69:

```

main(void){  

    FilaDupla* F;  

    F = criar_fila();  

    inserir_Fim(F, 2);  

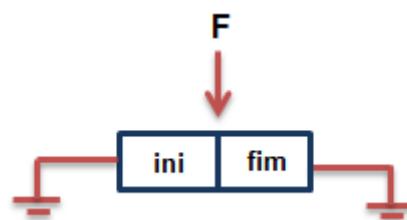
    inserir_Fim(F, 8);  

    inserir_Fim(F, 5);
}
```

Figura 69. Comandos que devem ser colocados na função principal.

**2º PASSO:** Quando inicializada a função principal o ponteiro F, tipo FilaDupla, é declarado.

**3º PASSO:** O ponteiro F recebe o retorno da função criar\_fila, o qual será NULL para ambos os ponteiros fim e ini.

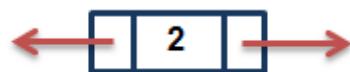


**4º PASSO:** A função inserir\_Fim(F, 6) realiza os seguintes procedimentos:

- Aloca-se um novo nó.



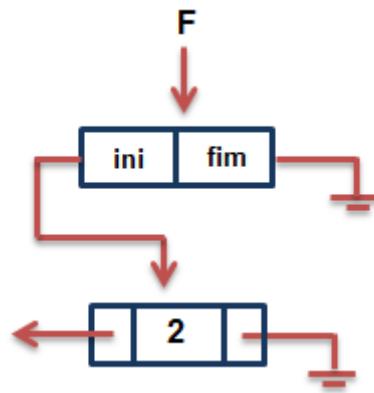
- Este nó recebe o valor 2 no campo de informação (novo -> info).



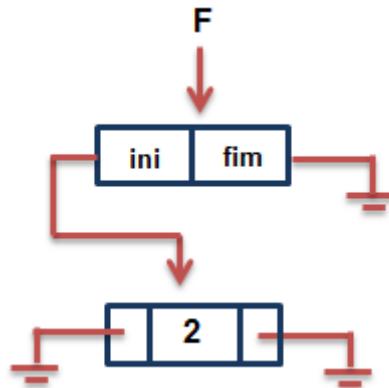
- O campo do endereço seguinte do nó criado (novo -> prox) aponta para NULL.



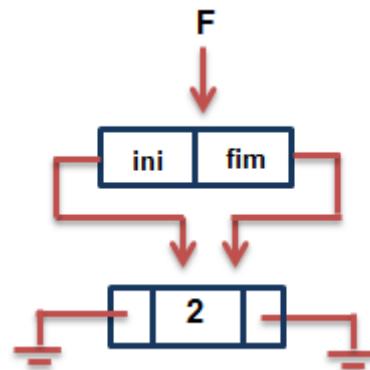
- O ponteiro fim, referenciado pelo ponteiro F, é diferente de NULL? NÃO, então, ainda não há elemento na fila. Entra na condicional else e, assim, o ponteiro ini, referenciado pelo ponteiro F (F -> ini), aponta para o novo nó.



- O campo do endereço anterior do nó criado (novo  $\rightarrow$  ant) aponta para NULL.



- O ponteiro fim, referenciado pelo ponteiro  $F$ , aponta para o novo nó.



**5º PASSO:** a função inserir\_Fim( $F$ , 8) realiza os seguintes procedimentos:

- Aloca-se um novo nó.



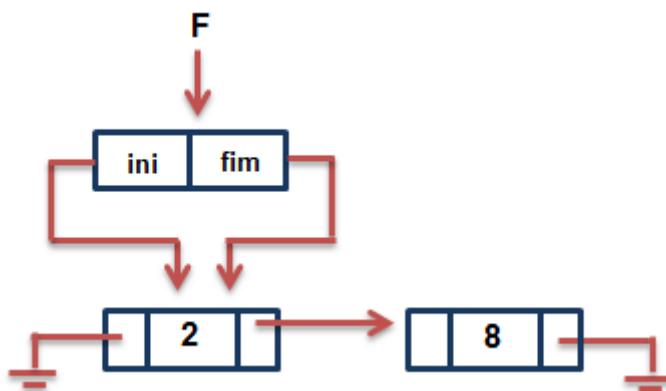
- Este nó recebe o valor 8 no campo de informação (novo  $\rightarrow$  info).



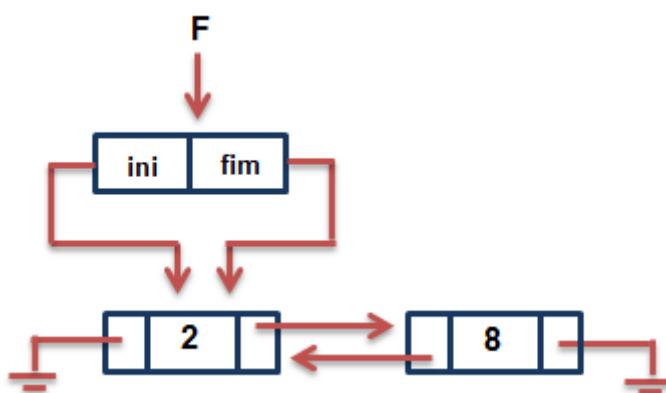
- O campo de endereço seguinte do nó criado (novo -> prox) aponta para NULL.



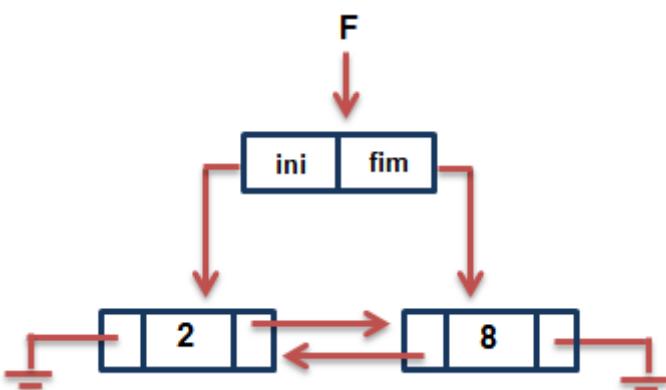
- O ponteiro fim, referenciado pelo ponteiro F, é diferente de NULL? SIM, então, há elemento na fila. Assim, o ponteiro prox do elemento, de valor 2, apontado pelo ponteiro fim, referenciado pelo ponteiro F (F -> fim -> prox), aponta para o novo nó.



- O campo do endereço anterior do nó criado (novo -> ant) aponta para onde o ponteiro fim, referenciado pelo ponteiro F, aponta, ou seja, para o último nó.



- O ponteiro fim, referenciado pelo ponteiro F, aponta para o novo nó.



**6º PASSO:** a função inserir\_Fim(F, 5) realiza os seguintes procedimentos:

- Aloca-se um novo nó.



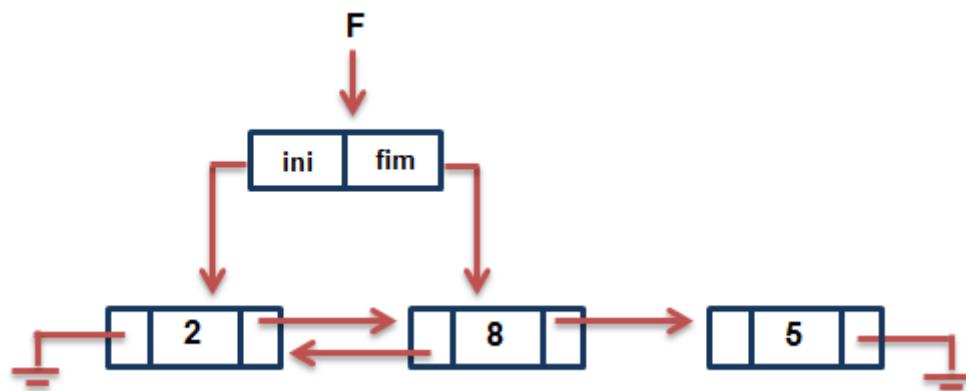
- Este nó recebe o valor 5 no campo de informação (novo -> info).



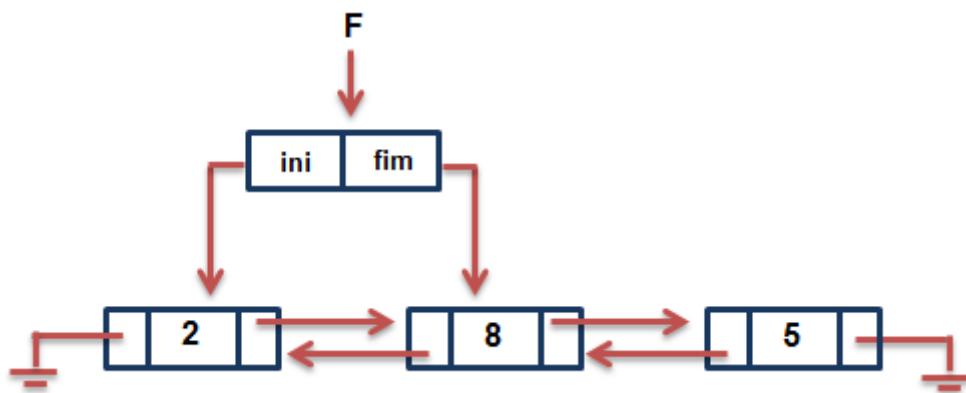
- O campo de endereço seguinte do nó criado (novo -> prox) aponta para NULL.



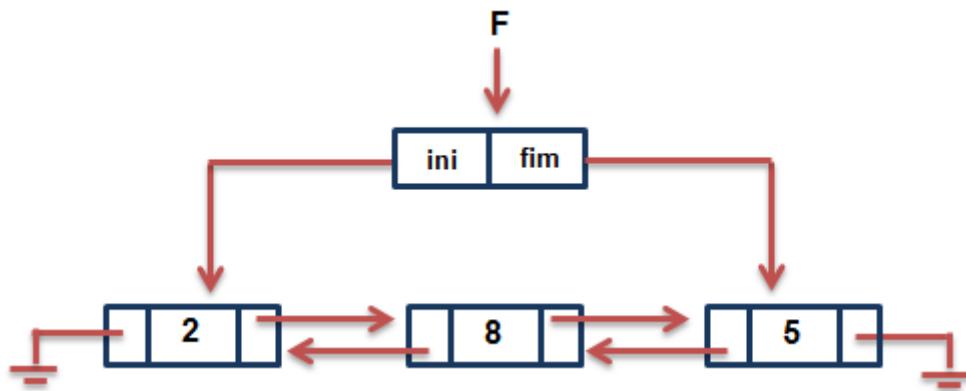
- O ponteiro fim, referenciado pelo ponteiro F, é diferente de NULL? SIM, então, há elemento na fila. Assim, o ponteiro prox do elemento, de valor 8, apontado pelo ponteiro fim, este referenciado pelo ponteiro F (F -> fim -> prox), aponta para o novo nó.



- O campo de endereço anterior do nó criado (novo -> ant) aponta para onde o ponteiro fim, referenciado pelo ponteiro F, aponta ou seja, para o último nó.



- O ponteiro fim, referenciado pelo ponteiro F, aponta para o novo nó.



#### 2.2.2.3.2. Inserir no início da Fila Dupla

O código referente a função denominada **inserir\_Inicio** é apresentado na figura 70.

```
void inserir_Inicio(FilaDupla* F, int valor){
    1     ListaDupla* novo = (ListaDupla*)malloc(sizeof(ListaDupla));
    2     novo -> info = valor;
    3     novo -> ant = NULL;
    4     if (F -> fim != NULL){
    4         F -> ini -> ant = novo;
    4         novo -> prox = F -> ini;
    5     }else{
    5         F -> fim = novo;
    5         novo -> prox = NULL;
    }
    6     F -> ini= novo;
}
```

Figura 70. Função referente a inserir elementos no início da fila dupla.

*Descrição das funcionalidades* apresentadas figura 70:

- Utilizando o conceito de alocação dinâmica, aloca-se o ponteiro denominado de **novo** do tipo **ListaDupla**.
- O **novo -> info** recebe um valor, do tipo inteiro, fornecido por meio do parâmetro da função, como pode ser observado na figura 71.

3. O *novo->ant* aponta para o valor nulo (NULL).
4. Se o ponteiro fim, referenciado pelo ponteiro F, for diferente de NULL (fila com elemento(s)) o ponteiro ant do nó apontado pelo ponteiro ini, referenciado pelo ponteiro F, aponta para o novo nó, assim como, o ponteiro prox do nó criado aponta para onde o ponteiro ini, referenciado pelo ponteiro F, aponta, ou seja, para o primeiro nó.
5. Senão significa que a fila está vazia, assim, o ponteiro fim, referenciado pelo ponteiro F, apontará para o novo nó, este sendo o primeiro a ser inserido e ponteiro prox do nó criado aponta para NULL. A condicional *else* será realizada somente uma única vez, ou seja, para a inserção do primeiro elemento.
6. Independente da condicional, if ou else, executada o ponteiro ini, referenciado pelo ponteiro F, apontará para o novo elemento.

Na função principal (**main()**) deve constar (figura 71):

```
main(void){
    1     FilaDupla* F;
    2     F = criar_fila();
    3     inserir_Inicio(F, <valor a ser inserido>);
}
```

Figura 71. Função principal chamando a função criar\_fila e inserir\_Inicio.

*Descrição das funcionalidades* apresentadas na figura 71:

1. Cria-se um ponteiro F, do tipo FilaDupla, o qual referenciará o início (*F -> ini*) e o fim da fila (*F -> fim*).
2. Chamar a função *criar\_fila* para atribuir o primeiro valor a fila, o qual é nulo. O ponteiro F declarado recebe o retorno desta função.
3. Chamar a função *inserir\_Inicio* passando como parâmetros o ponteiro F e o valor a ser inserido. Como visto não há retorno da função.

Para melhor entendimento, graficamente, realiza-se um exemplo a seguir, utilizando o conceito *teste de mesa*.

### Exemplo

Inserir os seguintes elementos 4 e 2.

**1º PASSO:** Na função principal colocar os seguintes comandos apresentados na figura 72:

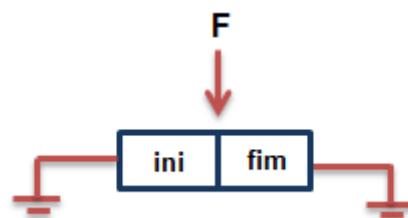
```

main(void){
    FilaDupla* F;
    F = criar_fila();
    inserir_Inicio(F, 4);
    inserir_Inicio(F, 2);
}
```

Figura 72. Comandos que devem ser colocados na função principal.

**2º PASSO:** Quando inicializada a função principal o ponteiro F, tipo FilaDupla, é declarado.

**3º PASSO:** O ponteiro F recebe o retorno da função criar\_fila, o qual será NULL para ambos os ponteiros fim e ini.



**4º PASSO:** a função inserir\_Inicio(F, 4) realiza os seguintes procedimentos:

- Aloca-se um novo nó.



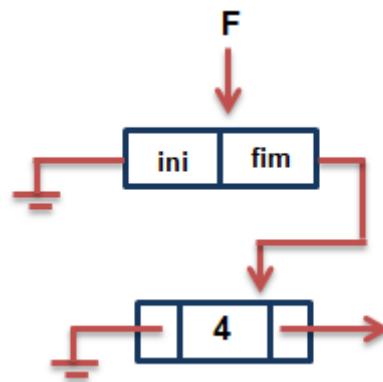
- Este nó recebe o valor 4 no campo de informação (*novo -> info*).



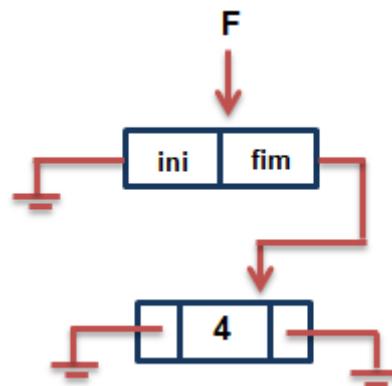
- O campo de endereço anterior do nó criado (*novo -> ant*) aponta NULL.



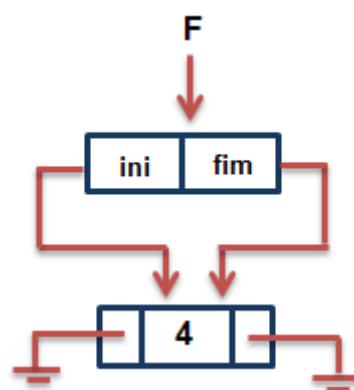
- O ponteiro fim, referenciado pelo ponteiro F, é diferente de NULL? NÃO, então, ainda não há elemento na fila. Entra na condicional else e, assim, o ponteiro fim, referenciado pelo ponteiro F (F -> fim), aponta para o novo nó.



- O campo de endereço seguinte do nó criado (novo -> prox) aponta NULL.



- O ponteiro ini, referenciado pelo ponteiro F, aponta para o novo nó.



**5º PASSO:** a função inserir\_Fim(F, 2) realiza os seguintes procedimentos:

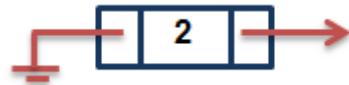
- Aloca-se um novo nó.



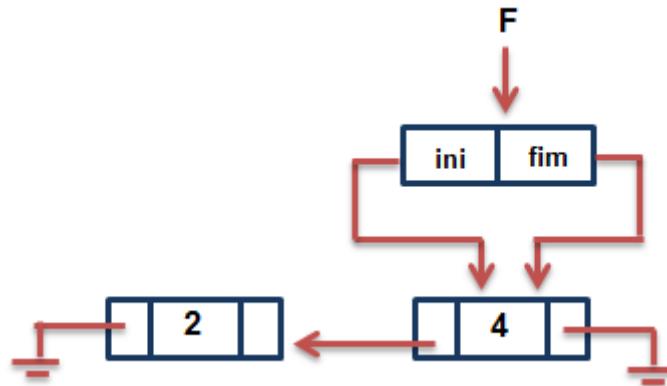
- Este nó recebe o valor 2 no campo de informação (novo -> info).



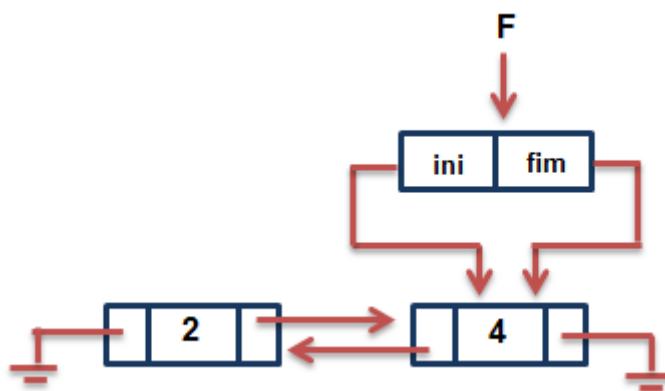
- O campo de endereço anterior do nó criado (novo -> ant) aponta para NULL.



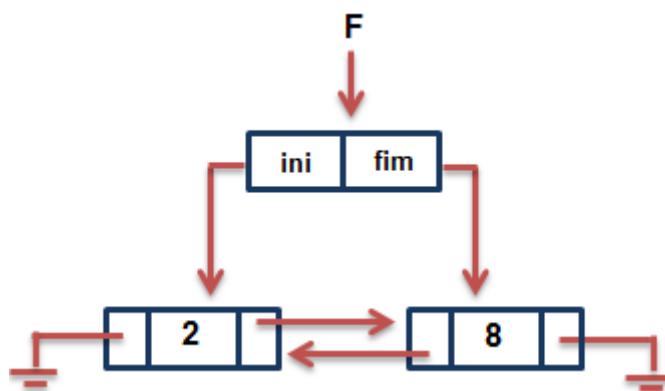
- O ponteiro fim, referenciado pelo ponteiro F, é diferente de NULL? SIM, então, há elemento na fila. Assim, o ponteiro ant do elemento, de valor 4, apontado pelo ponteiro ini, este referenciado pelo ponteiro F (F -> ini -> ant) aponta para o novo nó.



- O campo do endereço seguinte do nó criado (novo -> prox) aponta para onde o ponteiro ini, referenciado pelo ponteiro F, aponta, ou seja, para o primeiro nó.



- O ponteiro ini, referenciado pelo ponteiro F, aponta para o novo nó.



#### 2.2.2.4. Averiguar se a lista está vazia

##### O que fazer? Dicas...

Verificar se o inicio da fila aponta para NULL.

A função denominada **fila\_vazia**, apresentada na figura 73, possui as seguintes características:

- A função possui como parâmetro o ponteiro F.
- O retorno é do tipo **booleano**, retornando valor 1 (verdadeiro) quando a fila estiver vazia, caso contrário, retorna valor 0 (falso).

```
int fila_vazia(FilaDupla* F){
    return (F -> ini == NULL);
}
```

Figura 73. Função que verifica se a fila dupla está vazia.

#### 2.2.2.5. Imprimir os elementos

##### O que fazer? Dicas...

Percorrer a fila imprimindo os valores de cada nó.

Não manipular tanto o ponteiro ini quanto o ponteiro fim para não perder a referência do inicio e final da fila.

A função denominada **imprimir**, apresentada na figura 74, possui as seguintes características:

- A função possui como parâmetro o ponteiro F, o qual referencia os ponteiros ini e fim.
- O retorno é do tipo void, retornando na tela os valores contidos na fila.

```

void imprimir (FilaDupla* F){

    1      ListaDupla* q;

    2      if(fila_vazia(F))
    2          printf("\nFila vazia!\n")
    3      else{
    3          for (q = F -> ini; q != NULL; q = q -> prox)
    3              printf("%d", q -> info);
    3          }
    3          printf("\n\n");
}

```

Figura 74. Função imprimir.

*Descrição das funcionalidades* apresentadas na figura 74:

1. Declara-se um ponteiro auxiliar denominado de q, do tipo ListaDupla.
2. Dentro da condicional chama-se a função fila\_vazia, caso retorne o valor 1 a fila está vazia e, assim, imprimi-se “Fila vazia!”.
3. Senão entra no laço for:
  - Utiliza-se um ponteiro auxiliar denominado de q, o qual aponta para onde o ponteiro ini, referenciado pelo ponteiro F, aponta, ou seja, para o primeiro nó.
  - Enquanto o ponteiro q for diferente de NULL ainda há elemento(s) na fila.
  - O incremento do ponteiro q é representado por  $q = q \rightarrow prox$ .
  - A impressão é baseada no valor armazenado na região do campo de informação (info).

#### Observação

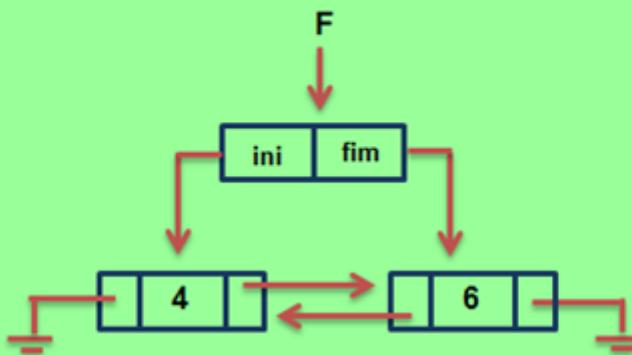
Na função principal (**main()**) a função para imprimir a fila deve ser chamada como:

**imprimir(F);**

Para melhor entendimento, graficamente, realiza-se um exemplo a seguir, utilizando o conceito *teste de mesa*.

Exemplo

Considerando a seguinte fila:



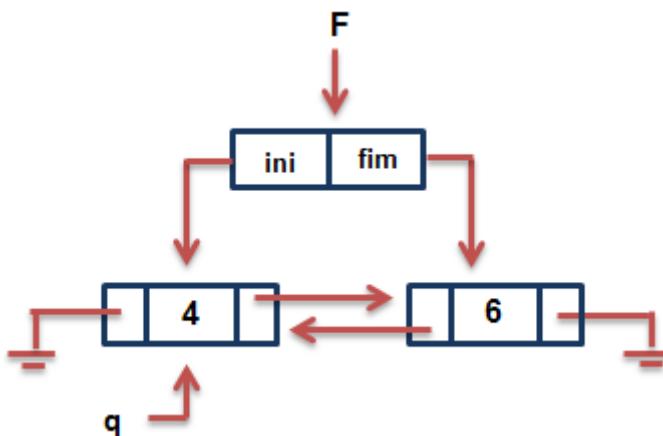
**1º PASSO:** Declara-se um ponteiro auxiliar denominado q.

**2º PASSO:** Entra na condicional if.

- Chama-se a função fila\_vazia, a fila está vazia? NÃO!! Pois o ponteiro ini, referenciado pelo ponteiro F, aponta para o nó de valor 4, ou seja, é diferente de NULL.

**3º PASSO:** Entra na condicional else. Percorre-se o laço for.

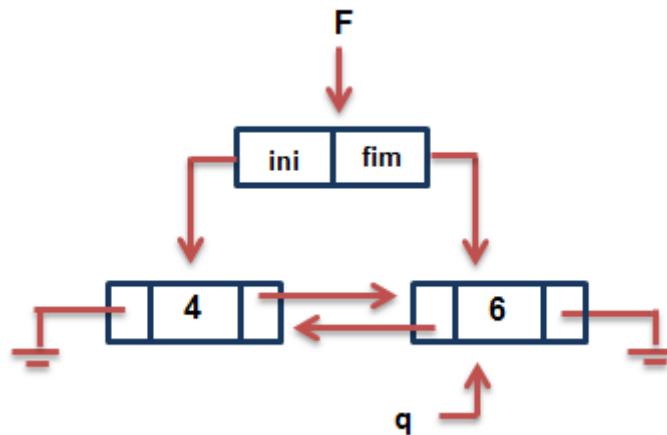
- O ponteiro q aponta para onde o ponteiro ini, referenciado pelo ponteiro F, aponta, ou seja, para o primeiro elemento da fila.



- O ponteiro q é diferente de NULL, ou seja, não aponta para NULL? SIM, assim imprime-se o valor 4 contido no elemento apontado pelo ponteiro q ( $q \rightarrow \text{info}$ ).

**4º PASSO:** Retorna ao laço for:

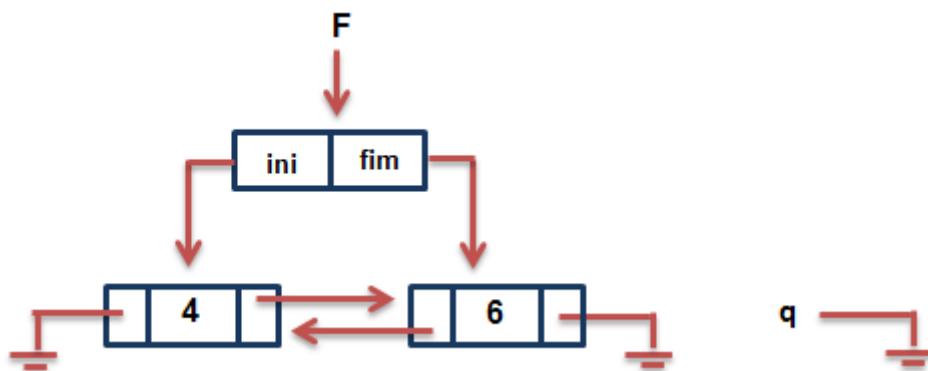
- O incremento do ponteiro q é dado por  $q = q \rightarrow \text{prox}$ , onde  $q \rightarrow \text{prox}$  é o elemento de valor 6. Assim, o ponteiro q aponta para este nó.



- O ponteiro  $q$  é diferente de NULL, ou seja, não aponta para NULL? SIM, assim imprime-se o valor 6 contido no elemento apontado pelo ponteiro  $q$  ( $q \rightarrow \text{info}$ ).

**5º PASSO:** Retorna ao laço for:

- O incremento do ponteiro  $q$  é dado por  $q = q \rightarrow \text{prox}$ , onde  $q \rightarrow \text{prox}$  possui valor nulo. Assim, o ponteiro  $q$  aponta para NULL.



- O ponteiro  $q$  é diferente de NULL, ou seja, não aponta para NULL? NÃO, ou seja, imprimiram-se todos os elementos contidos na fila.

**6º PASSO:** Termina o laço, assim, encerra a impressão!!

#### 2.2.2.6. Buscar um determinado elemento na fila

##### O que fazer? Dicas...

- Percorrer a fila comparando o valor que o nó possui com o valor a ser buscado.
- Caso encontre, retorna o nó que contém o valor. Caso contrário, retorna NULL.

A função denominada **buscar\_elemento**, apresentada na figura 75, possui as seguintes características:

- A função possui como parâmetro o valor a ser procurado e o ponteiro F, o qual possui acesso aos ponteiros ini e fim.
- O retorno é do tipo FilaDupla. Caso o elemento seja encontrado o ponteiro q aponta para ele, assim, retorna este nó, caso contrário, retorna o valor nulo (NULL).

```
FilaDupla* buscar_elemento (FilaDupla* F, int valor ){
```

```
    ListaDupla* q;
    1           2           3
    for(q = F -> ini; q != NULL; q = q -> prox) {
        4           if(q -> info == valor)
        4           return q;
        }
    5           return NULL;
    }
```

Figura 75. Função buscar elemento na fila dupla.

*Descrição das funcionalidades* apresentadas na figura 75:

1. O ponteiro ini, referenciado pelo ponteiro F, aponta para o primeiro nó da fila, este não deve ser manipulado para não perder a referência, então, utiliza-se um ponteiro auxiliar denominado de q, o qual aponta para onde F -> ini aponta, ou seja, para o primeiro nó;
2. Enquanto o ponteiro q for diferente de NULL ainda há elemento(s) na fila.
3. O incremento do ponteiro q é representado por  $q = q -> prox$ .
4. Se o campo de informação do nó apontado pelo ponteiro q ( $q->info$ ) for igual ao valor procurado retorna-se o ponteiro q, o qual terá acesso ao campo de informação (info) e ao campo do endereço seguinte (prox), já que q é do tipo ListaDupla.
5. Caso não encontrado, retorna-se o valor nulo (NULL).

#### Observação

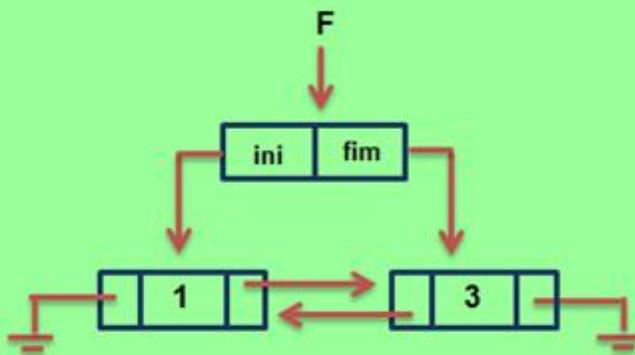
Na função principal (**main()**) a função buscar deve ser chamada como:

```
ListaDupla* resultado = buscar_elemento(F, <valor a ser procurado>);
```

Para melhor entendimento, graficamente, realiza-se um exemplo a seguir, utilizando o conceito *teste de mesa*.

### Exemplo

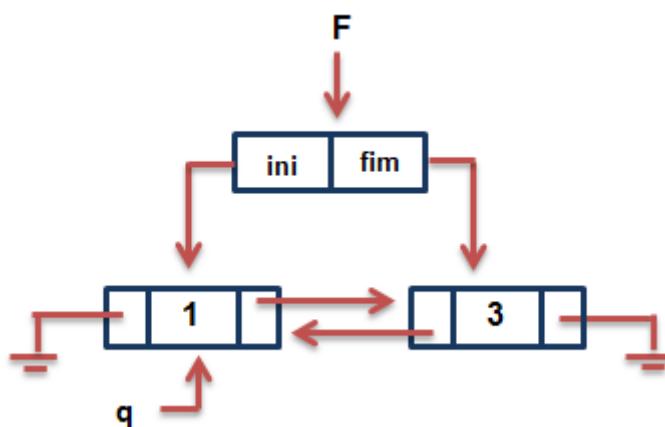
Considerando a seguinte fila e o número 2 a ser procurado:



**1º PASSO:** Declara-se um ponteiro auxiliar denominado q.

**2º PASSO:** Percorrer o laço for:

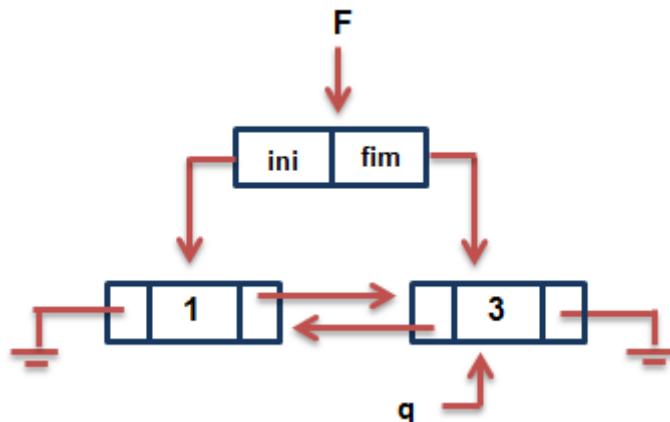
- O ponteiro q aponta para onde o ponteiro ini, referenciado pelo ponteiro F ( $F \rightarrow ini$ ) aponta, ou seja, para o primeiro nó.



- O ponteiro q é diferente de NULL? SIM!! O ponteiro q aponta para o nó de valor 1.
- Entra na condicional if.
- O ponteiro q que na região info ( $q \rightarrow info$ ) vale 1, é igual ao valor procurado que é 2, passado como parâmetro? NÃO!!

**3º PASSO:** Retorna ao laço for.

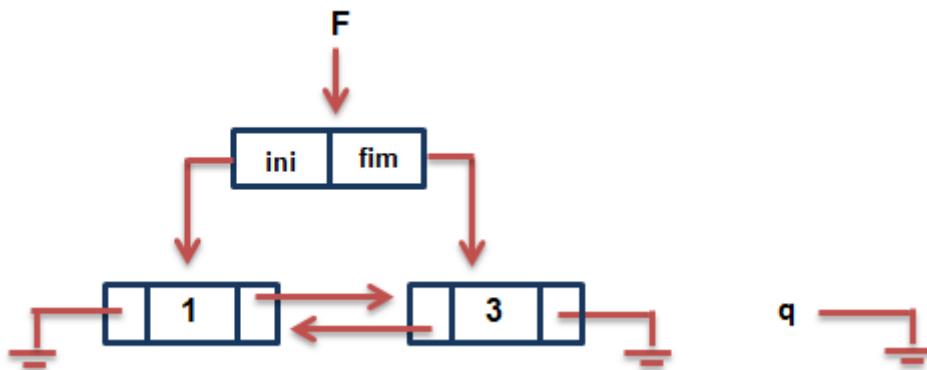
- O incremento do ponteiro  $q$  é dado por  $q = q \rightarrow \text{prox}$ , onde  $q \rightarrow \text{prox}$  é o elemento de valor 3. Assim, o ponteiro  $q$  aponta para este nó.



- O ponteiro  $q$  é diferente de **NULL**? **SIM!!** O ponteiro  $q$  aponta para o nó de valor 3.
- Entra na condicional if.
- O ponteiro  $q$  que na região **info** ( $q \rightarrow \text{info}$ ) vale 3, é igual ao valor procurado que é 2, passado como parâmetro? **NÃO!!**

**4º PASSO:** Retorna ao laço for.

- O incremento do ponteiro  $q$  é dado por  $q = q \rightarrow \text{prox}$ , onde  $q \rightarrow \text{prox}$  possui valor nulo. Assim, o ponteiro  $q$  aponta para **NULL**.



- O ponteiro  $q$  é diferente de **NULL**? **NÃO!!** O ponteiro  $q$  aponta para **NULL**.
- Termina o laço for.

**5º PASSO:** Retorna **NULL**, pois não há um elemento com valor 2.

### 2.2.2.7. Liberar todos os elementos

#### O que fazer? Dicas...

- Percorrer a fila desalocando cada elemento.
- Utilizar um ponteiro auxiliar que aponte para o inicio da fila, e ainda, utilizar um outro ponteiro que aponte para o nó seguinte ao ponteiro auxiliar criado.

A função denominada **liberar\_fila**, apresentada na figura 76, possui as seguintes características:

- A função possui como parâmetro o ponteiro F.
- Não há retorno para a função principal, é do tipo void.

```
void liberar_fila (FilaDupla* F){
    1     ListaDupla* q = F -> ini;
    2     while (q != NULL) {
    3         ListaDupla* t = q -> prox;
    4         free(q);
    5         q = t;
    6     }
}
```

Figura 76. Função liberar fila.

*Descrição das funcionalidades* apresentadas na figura 76:

1. Declara-se um ponteiro auxiliar denominado de q, do tipo ListaDupla, o qual aponta para onde o ponteiro ini, referenciado pelo F, aponta, ou seja, para o primeiro elemento da fila.
2. No laço while, enquanto o ponteiro q for diferente de NULL ainda há elemento(s) na lista, então, executa-se os comandos encontrados dentro do laço.
3. Utiliza-se um ponteiro auxiliar denominado de t que aponta para o nó a frente do ponteiro q ou para NULL, caso o ponteiro q aponte para o último nó.
4. Desaloca o nó apontado pelo ponteiro q.
5. O ponteiro q aponta para onde o ponteiro t aponta.

6. Após desalocar todos os elementos desaloca-se o ponteiro F.

### Observação

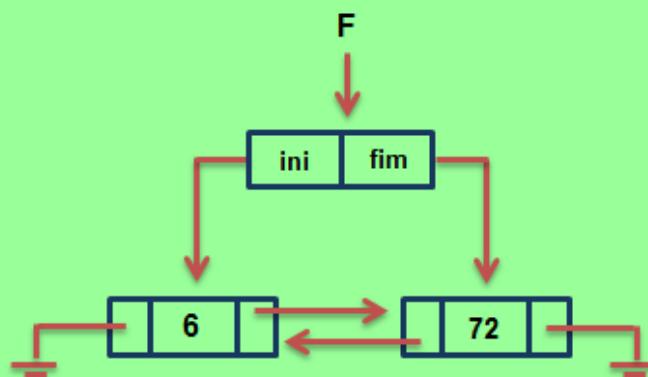
Na função principal (`main()`) a função liberar deve ser chamada como:

`liberar_fila(F);`

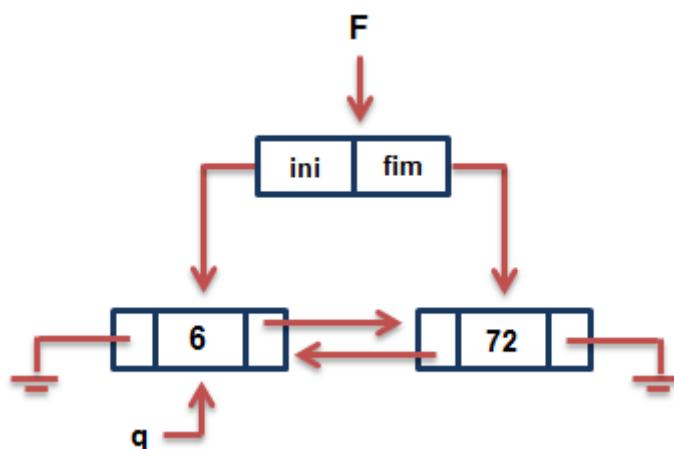
Para melhor entendimento, graficamente, realiza-se um exemplo a seguir, utilizando o conceito *teste de mesa*.

### Exemplo

Considerando a seguinte fila:



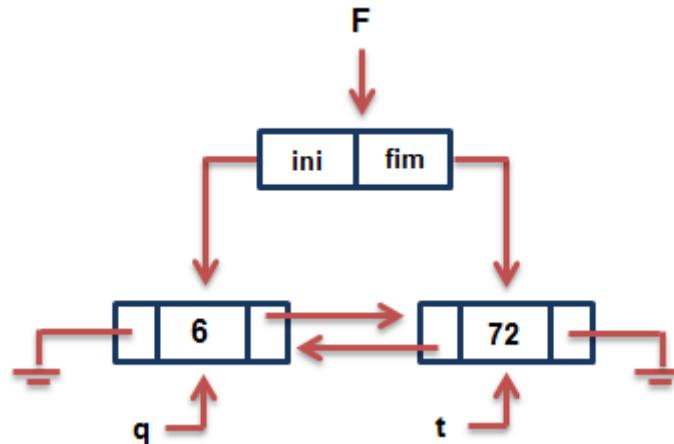
**1º PASSO:** Ponteiro q aponta para onde o ponteiro ini, referenciado pelo ponteiro F, aponta, ou seja, para o início da fila.



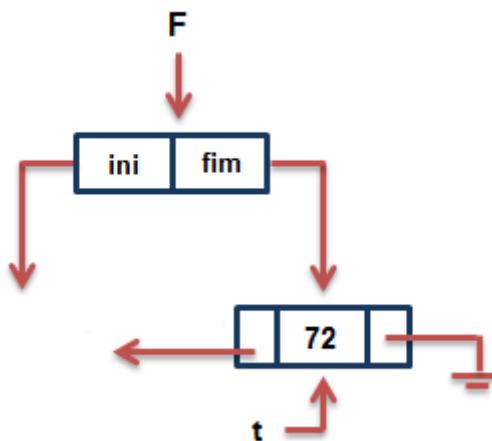
**2º PASSO:** Entra no laço while.

- O ponteiro q é diferente de NULL? SIM!! O ponteiro q aponta para o nó de valor 6.

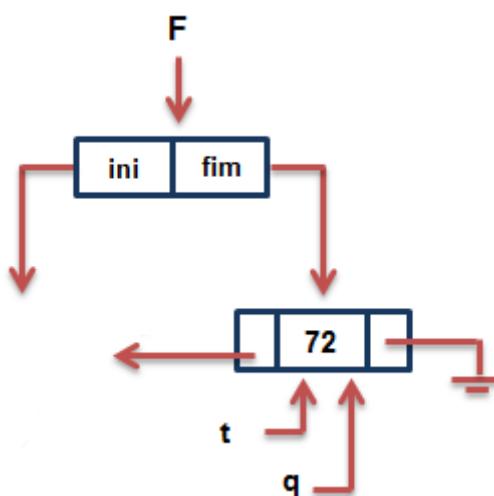
- O ponteiro t aponta para o próximo nó do ponteiro q ( $q->prox$ ). No caso, apontará para o nó de valor 72.



- Desaloca o nó apontado pelo ponteiro q.

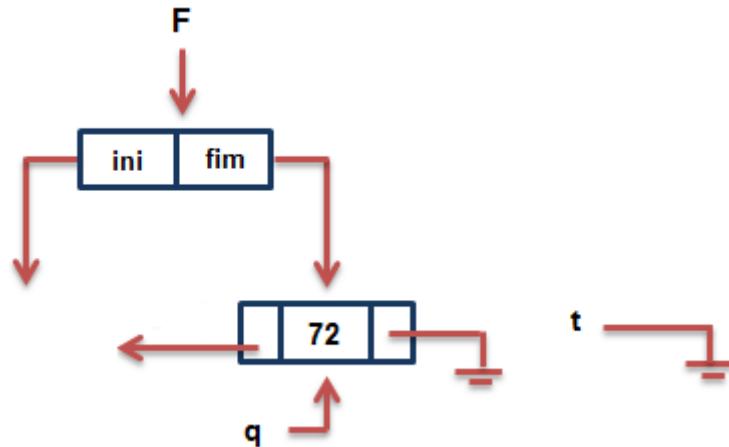


- O Ponteiro q aponta para onde o ponteiro t aponta.

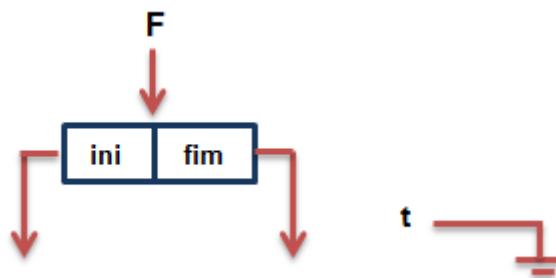


**3º PASSO:** Retorna ao laço while.

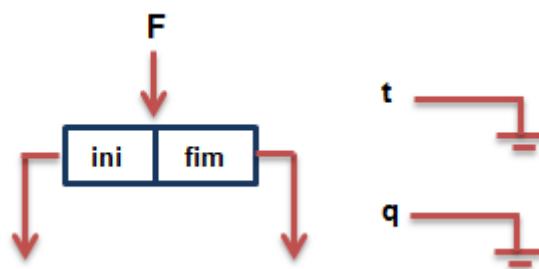
- O ponteiro q é diferente de NULL? SIM!! O ponteiro q aponta para o nó de valor 72.
- O ponteiro t aponta para o próximo nó do ponteiro q ( $q->prox$ ). No caso, apontará para NULL.



- Desaloca o nó apontado pelo ponteiro q.



- O ponteiro q aponta para onde o ponteiro t aponta.



**4º PASSO:** Retorna ao laço while.

- O ponteiro q é diferente de NULL? NÃO!! O ponteiro q aponta para NULL
- Terminam as iterações do laço while.

**5º PASSO:** Desaloca o ponteiro F. Assim, liberou toda a fila.

### 2.2.2.8. Retirar elemento

#### O que fazer? Dicas...

- Utilizar um ponteiro auxiliar, o qual deve apontar para o primeiro ou para o último elemento da fila.

\*\*\* Em uma fila retira-se somente o elemento do inicio ou do fim!

Tanto a função **retirar\_Inicio** quanto a função **retirar\_Fim**, apresentadas nas figuras 77 e 78, possuem as seguintes características:

- A função possui como parâmetros o ponteiro F, do tipo FilaDupla, que referencia ao ponteiro *ini* e *fim*, ambos do tipo ListaDupla.
- O retorno é do tipo inteiro, retornando o valor contido no campo de informação (*info*) do nó a ser removido.

#### 2.2.2.8.1. Retirar do início da Fila Dupla

#### O que fazer? Dicas...

O ponteiro que referencia o inicio da fila deve apontar para o segundo elemento, o qual se tornará o primeiro. Caso não haja um segundo nó o ponteiro que referencia o inicio deve apontar para NULL.

O código referente à função denominada **retirar\_Inicio** é apresentado na figura 77.

#### Observação

Na função principal (**main()**) a função para retirar do inicio da fila deve ser chamada como:

```
int valor_retirado = retirar_Inicio(F);
```

```

int retirar_Inicio (FilaDupla* F){

    1      ListDupla* t;
    2      int v;

    3      if(fila_vazia(F)){
    3          printf("\nFila vazia!\n");
    3          exit(1);
    }

    4      t = F -> ini;
    5      v = t -> info;
    6      F -> ini = t -> prox;

    7      if(t -> prox == NULL)
    7          F -> fim = NULL;
    8      else
    8          t -> prox -> ant = NULL;

    9      free(t);
    10     return v;
}

```

Figura 77. Função referente a retirada do primeiro elemento da fila dupla.

*Descrição das funcionalidades* apresentadas figura 77:

1. O ponteiro F possui acesso a dois ponteiros ini e fim. Somente o ponteiro ini será relevante, pois haverá a retirada do primeiro nó da fila, assim, declara-se um ponteiro auxiliar denominado de t para que não ocorra a perda da referência do ponteiro ini, referenciado pelo ponteiro F.
2. Declara-se uma variável, do tipo inteiro, denominada de v. Esta armazenará o valor do primeiro nó, o qual será retirado.
3. Chama-se a função fila\_vazia, se estiver vazia apresentará uma mensagem na tela e finalizará a chamada da função retirar\_Inicio.
4. O ponteiro t aponta para o início da fila ( $F \rightarrow ini$ ), ou seja, para o primeiro elemento.
5. A variável v armazena o valor do campo de informação do nó apontado pelo ponteiro t.
6. O ponteiro ini, referenciado pelo ponteiro F, aponta para o nó seguinte em relação ao ponteiro t, ou seja, para o segundo elemento, ou para NULL, caso não haja mais nós.
7. Entra na segunda condicional if. Se o ponteiro prox, referenciado pelo t ( $t \rightarrow prox$ ), o que equivale ao ponteiro ini, referenciado pelo ponteiro F ( $F \rightarrow ini$ ), já

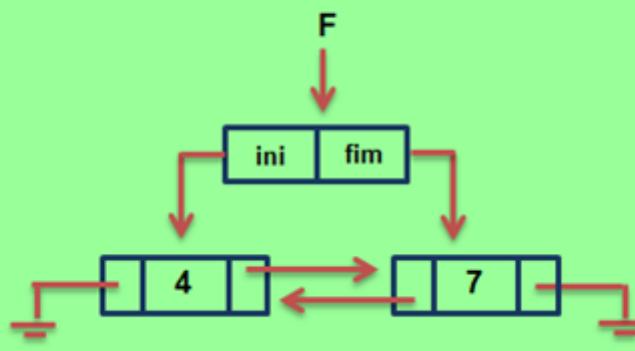
que  $F \rightarrow ini$  recebeu  $t \rightarrow prox$  no comando 6, for igual à NULL (fila com um único elemento) o ponteiro fim, referenciado pelo ponteiro F, apontará para NULL.

8. Entra na condicional else. O ponteiro prox, referenciado pelo ponteiro t ( $t \rightarrow prox$ ), aponta para o segundo elemento da fila, já que o ponteiro t aponta para o primeiro, o ponteiro ant deste segundo elemento apontará para NULL.
9. Desaloca o nó apontado pelo ponteiro t.
10. Retorna o valor contido no campo de informação do elemento removido.

Para melhor entendimento, graficamente, realiza-se um exemplo a seguir, utilizando o conceito *teste de mesa*.

### Exemplo

Considerando a seguinte fila:

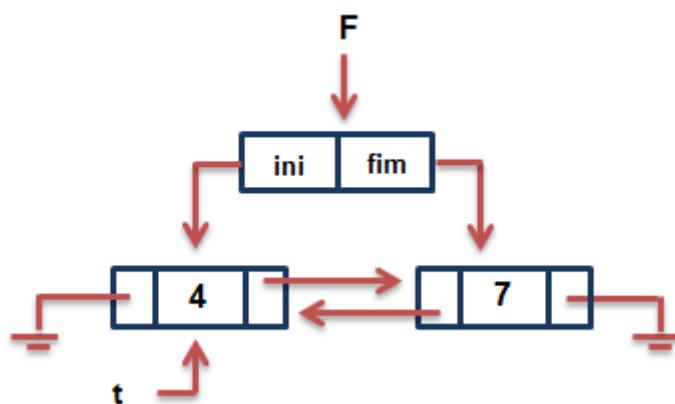


**1º PASSO:** Declaração do ponteiro t e da variável v.

**2º PASSO:** Entra na condicional if.

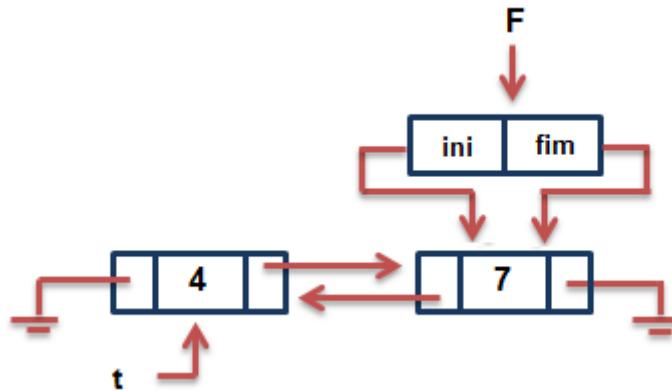
- Chama-se a função fila\_vazia, contudo retorna 0, ou seja, a fila não está vazia, possui um elementos.

**3º PASSO:** O ponteiro t aponta para o primeiro elemento, este apontado pelo ponteiro ini, referenciado pelo ponteiro F ( $F \rightarrow ini$ ).



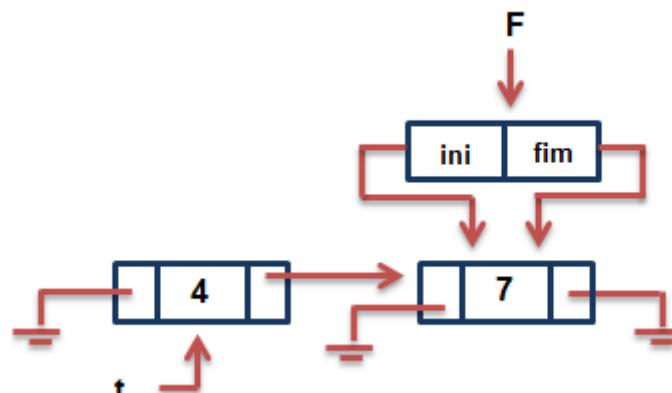
**4º PASSO:** A variável v armazena o valor 4 contido no campo de informação do nó apontado pelo ponteiro t ( $t \rightarrow \text{info}$ ).

**5º PASSO:** O ponteiro ini, referenciado pelo ponteiro F, aponta para o nó seguinte em relação ao ponteiro t, ou seja, o nó de valor 7.

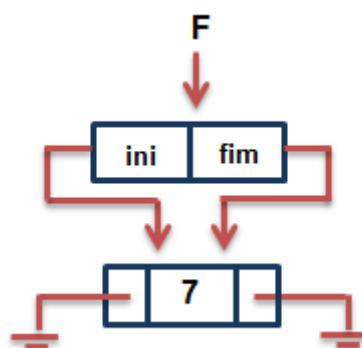


**6º PASSO:** Entra na condicional if.

- O ponteiro prox, referenciado pelo ponteiro t, é igual a NULL? NÃO!! Pois  $t \rightarrow \text{prox}$  aponta para o nó de valor 7.
- Entra na condicional else.
- O ponteiro ant do nó sucessor ao elemento apontado pelo ponteiro t aponta para NULL.



**7º PASSO:** Desaloca o nó apontado pelo ponteiro t.



**8º PASSO:** Retorna o valor 4 contido na variável v

### 2.2.2.8.2. Retirar do final da Fila Dupla

#### O que fazer? Dicas...

O ponteiro que referencia o final da fila deve apontar para o penúltimo elemento, o qual se tornará o último. Caso não haja um penúltimo nó o ponteiro que referencia o final deve apontar para NULL.

O código referente à função denominada **retirar\_Fim** é apresentado na figura 78.

```

int retirar_Fim (FilaDupla* F){

    1      ListaDupla* t;
    2      int v;

    3      if(fila_vazia(F)){
    3          printf("\nFila vazia!\n");
    3          exit(1);
    }

    4      t = F -> fim;
    5      v = t -> info;
    6      F -> fim = t -> ant;

    7      if(t -> ant == NULL)
    7          F -> ini = NULL;
    8      else
    8          t -> ant -> prox = NULL;

    9      free(t);
   10      return v;
}

```

Figura 78. Função referente a retirada do último elemento da fila dupla.

*Descrição das funcionalidades* apresentadas figura 78:

1. O ponteiro F possui acesso a dois ponteiros ini e fim. Somente o ponteiro fim será relevante, pois haverá a retirada do último nó da fila, assim, declara-se um ponteiro auxiliar denominado de t para que não ocorra a perda da referência.
2. Declara-se uma variável, do tipo inteiro, denominada de v. Esta armazenará o valor do último nó, o qual será retirado.

3. Chama-se a função fila\_vazia, se estiver vazia apresentará uma mensagem na tela e finalizará a chamada da função retirar\_Fim.
4. O ponteiro t aponta para o final da fila ( $F \rightarrow \text{fim}$ ), ou seja, para o último elemento.
5. A variável v armazena o valor do campo de informação do nó apontado pelo ponteiro t.
6. O ponteiro fim, referenciado pelo ponteiro F, aponta para o nó anterior em relação ao ponteiro t, ou seja, para o penúltimo elemento, ou para NULL, caso a fila tenha somente um nó antes da chamada da função retirar\_Fim.
7. Entra na segunda condicional if. Se o ponteiro ant, referenciado pelo t ( $t \rightarrow \text{ant}$ ) o que equivale ao ponteiro fim, referenciado pelo ponteiro F ( $F \rightarrow \text{fim}$ ), já que F  $\rightarrow$  fim recebeu t  $\rightarrow$  ant no comando 6, for igual à NULL (fila com um único elemento) o ponteiro ini, referenciado pelo ponteiro F, apontará para NULL.
8. Entra na condicional else. O ponteiro ant, referenciado pelo ponteiro t ( $t \rightarrow \text{ant}$ ), aponta para o penúltimo elemento da fila, já que o ponteiro t aponta para o último, o ponteiro prox deste penúltimo elemento apontará para NULL.
9. Desaloca o nó apontado pelo ponteiro t.
10. Retorna o valor contido no campo de informação do elemento removido.

### Observação

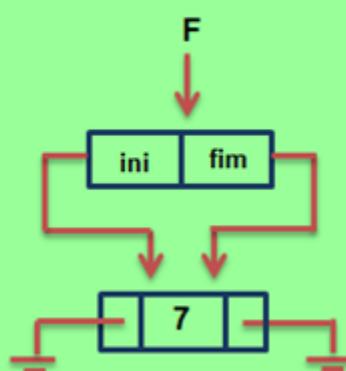
Na função principal (**main()**) a função para retirar do fim da fila deve ser chamada como:

```
int valor_retirado = retirar_Fim(F);
```

Para melhor entendimento, graficamente, realiza-se um exemplo a seguir, utilizando o conceito *teste de mesa*.

### Exemplo

Considerando a seguinte fila:

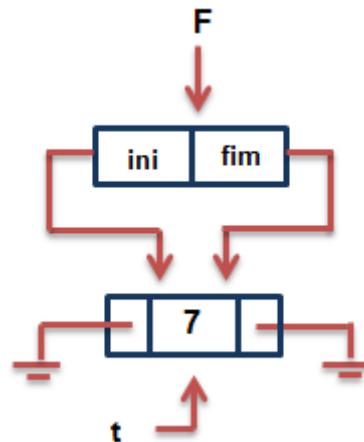


**1º PASSO:** Declaração do ponteiro t e da variável v.

**2º PASSO:** Entra na condicional if.

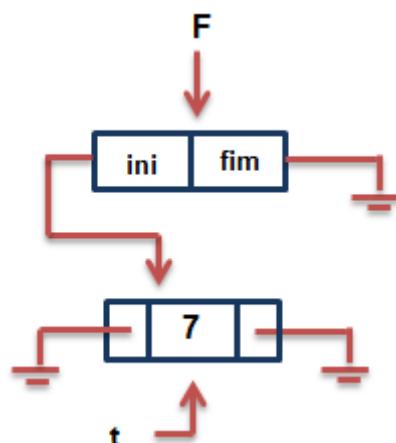
- Chama-se a função fila\_vazia, contudo retorna 0, ou seja, a fila não está vazia.

**3º PASSO:** O ponteiro t aponta para o último elemento, este apontado pelo F -> fim.



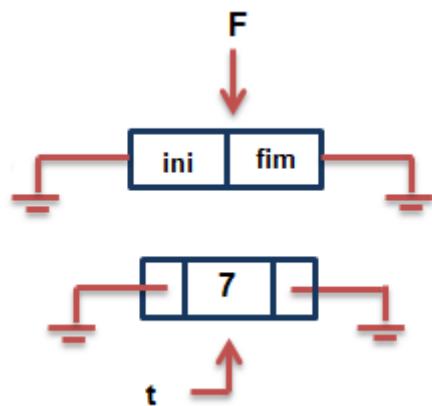
**4º PASSO:** A variável v armazena o valor 7 contido no campo de informação do nó apontado pelo ponteiro t (t ->info).

**5º PASSO:** O ponteiro fim, referenciado pelo ponteiro F, aponta para onde o ponteiro ant, referenciado pelo ponteiro t, aponta, ou seja, para NULL.

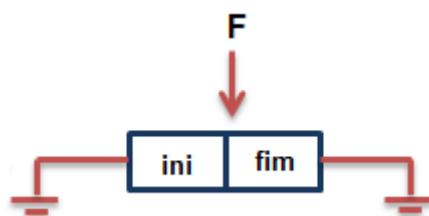


**6º PASSO:** Entra na condicional if.

- O ponteiro ant, referenciado pelo ponteiro t, é igual a NULL? SIM!! Há somente um elemento na fila.
- O ponteiro ini, referenciado pelo ponteiro F, aponta para NULL.



**7º PASSO:** Desaloca o nó apontado pelo ponteiro t.



**8º PASSO:** Retorna o valor 7 contido na variável v.

### 2.3. Fila Circular

### **2.3.1. Conceitos**

Uma fila circular é formada por um conjunto de elementos, denominados também de nós. Permite a reutilização de posições já ocupadas.

O último elemento tem como próximo o primeiro nó da fila, formando um ciclo, como mostra a figura 79. A fila pode ser representada por um ponteiro para um elemento inicial e final quaisquer.

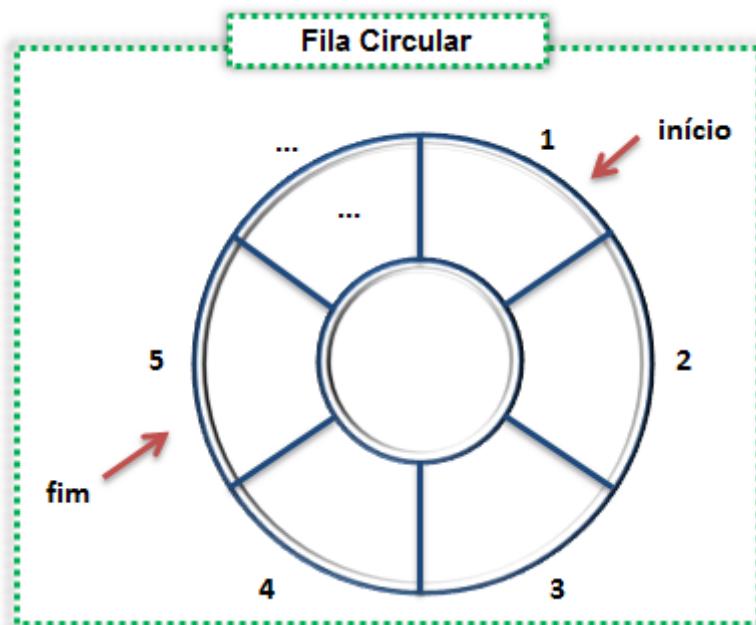


Figura 79. Representação de uma fila circular.

### **2.3.2. Manipulação**

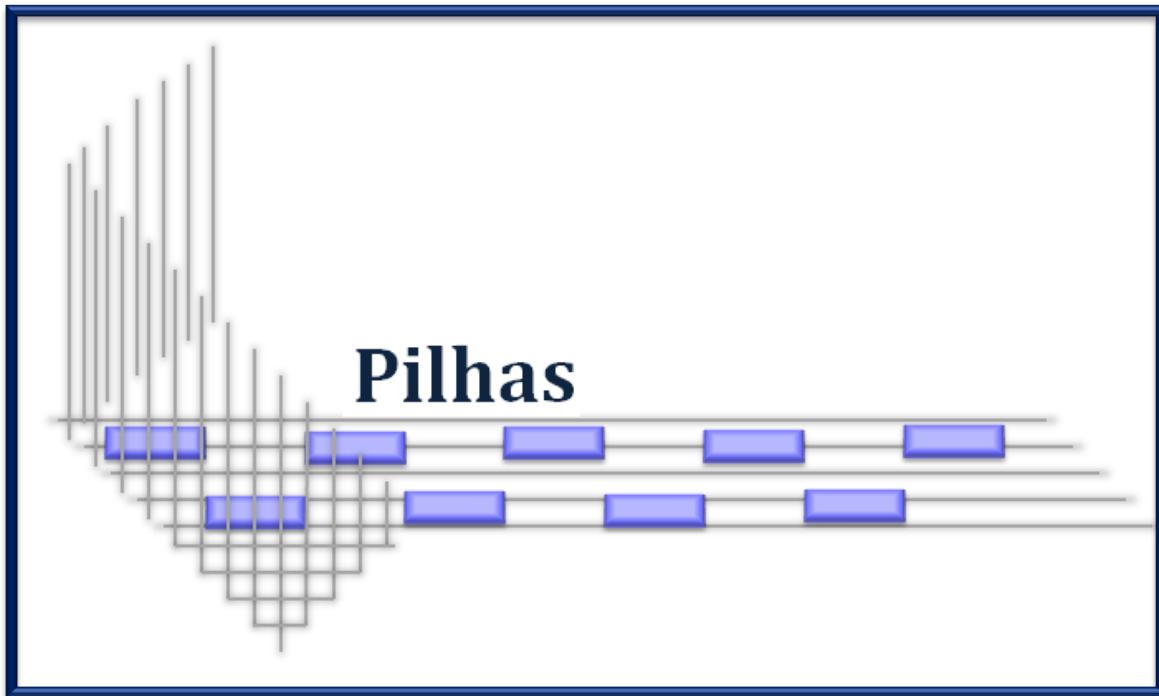
Pode-se programar a criação da fila circular e estabelecer funcionalidades a ela. Conceitos básicos:

- **Entra** elemento no **fim** da fila:
    - Se **não** estiver **cheia** (total = tamanho array).
      - Vetor[fim] = elemento.
      - Avança fim (“módulo tamanho do array” para “fazer a curva”, se preciso).
      - Atualiza total.

- **Sai** o primeiro elemento::
  - Se **não** estiver **vazia** ( $\text{total} \neq 0$ ).
    - elemento = vetor[início].
    - Avança início (“módulo tamanho do array” para “fazer a curva”, se preciso).
    - Atualiza total.

## 2.4. Exercícios

- 1) Implemente uma função que verifica se os elementos de uma fila, representada por uma lista simplesmente encadeada, estão ordenados de forma descrecente.
- 2) Faça uma fila circular, representada por um vetor, e implemente as seguintes funções:
  - a) Inserir.
  - b) Remover.
  - c) Imprimir.
- 3) Construa uma função que retorne a quantidade de números pares em uma fila, representada por uma lista duplamente encadeada. Assim como, implementar as funções referentes à inserção, remoção e impressão da fila.
- 4) Implemente uma função que verifique se duas filas, representadas por uma lista simplesmente encadeada, são iguais. Assim como, implementar as funções referentes à inserção, remoção e impressão da fila.
- 5) Implemente uma função que realize a impressão inversa de uma fila representada por uma lista simples.
- 6) Implemente uma função que receba uma Fila F1 e retorne esta sem os nós com valores ímpares. Para isso, deverá ser utilizada uma lista. Exemplo: Fila F1 = {2, 3, 4, 7}, após a execução da função a Fila F1 ficará {2, 4}.



### 3. PILHA SIMPLES

#### 3.1. Conceitos

Uma Pilha Simples, formada a partir de uma Lista Simplesmente Encadeada (seção 1.1), é um conjunto de elementos (nós), os quais são empilhados, assim, tem-se o topo da pilha. Um ponteiro, apontando inicialmente para NULL, é criado para realizar esta função (figura 80):

- Ponteiro que referencia o topo da pilha denominado de **topo**.

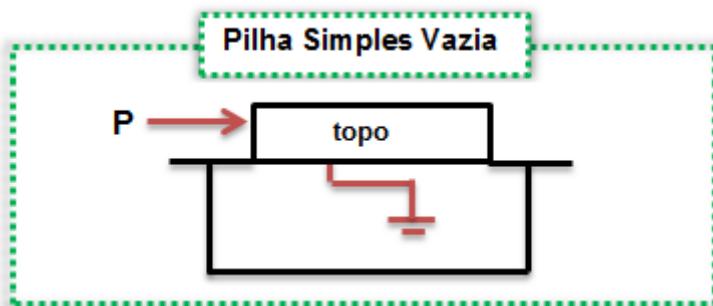


Figura 80. Representação de uma pilha simples vazia.

Uma Pilha Simples, como mostra a figura 81, é formada a partir de uma lista Simples, assim, dado o ponteiro de um nó não é possível ter acesso aos elementos adjacentes, pois mantém seus elementos em uma única direção, da *esquerda para direita*. Esta direção é determinada pelos campos do endereço seguinte de cada nó, os quais são ponteiros, que partem de um nó para seu sucessor na lista.

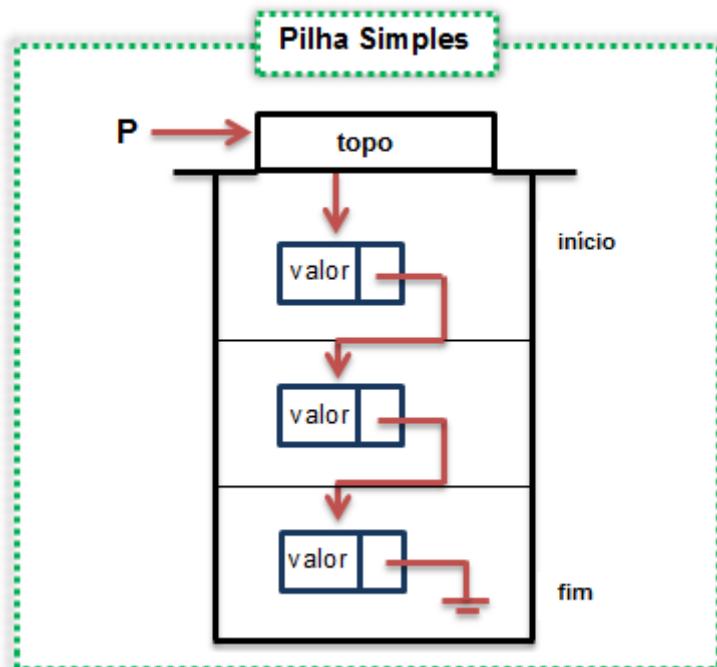


Figura 81. Representação de uma pilha simples.

### Observação

Utiliza-se os conceitos de Lista Simplesmente Encadeada para a criação de uma Pilha Simples. Contudo, como a estrutura a ser desenvolvida é referente a uma Pilha não utiliza-se o ponteiro **L** que possui a função de apontar para o inicio da lista e, sim, um ponteiro com a função de apontar para o topo da pilha.

## 3.2. Criação e Manipulação

Pode-se programar a criação da pilha simples e estabelecer as devidas funcionalidades a ela, entre estas se tem:

- 3.2.1. Criação de um Tipo Abstrato de Dado (TAD);
- 3.2.2. Criar a pilha.
- 3.2.3. Inserir elementos na pilha.
- 3.2.4. Averiguar se a pilha está vazia.
- 3.2.5. Imprimir os elementos.
- 3.2.6. Buscar um determinado elemento na pilha.
- 3.2.7. Liberar todos os elementos.
- 3.2.8. Retirar um nó.

### 3.2.1. Criação de um tipo abstrato de dado (TAD)

Criação de um novo tipo de dado, como mostra a figura 82.

```
typedef struct lista{
    1     int info;
    2     struct lista* prox;
}ListaSimp;

typedef struct pilha{
    3     ListaSimp* topo;
}PilhaSimp;
```

Figura 82. Estrutura de um novo tipo de dado do tipo PilhaSimp.

*Descrição das funcionalidades* apresentadas na figura 82:

Os marcadores 1 e 2 se referem ao tipo abstrato de dado **Lista Simples**, possuindo as seguintes funcionalidades (figura 83):

1. Declarou-se a informação (info) do *campo de informação*, do tipo inteiro, o qual armazena o valor contido no nó.
2. Declarou-se o ponteiro (prox) do *campo do endereço seguinte*, sendo do tipo struct lista.



Figura 83. Região declarada como info e outra como prox.

O marcador 3 se refere ao tipo abstrato de dado **Pilha Simples**, como visto anteriormente, uma pilha é composta por uma lista, assim, seu atributo é um ponteiro do tipo ListaSimp. Possuindo a seguinte funcionalidade (figura 84):

3. Declarou-se o ponteiro topo, do tipo ListaSimp, o qual representa o topo da pilha.

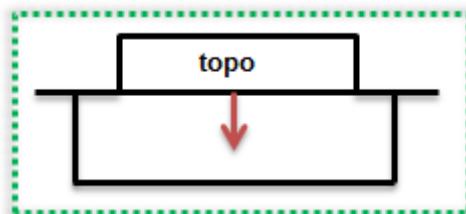


Figura 84. Região declarada como topo.

### 3.2.2. Criar a pilha

#### O que fazer? Dicas...

Toda pilha deve inicializar vazia!!!!

A função denominada **criar\_pilha**, apresentada na figura 85, possui as seguintes características:

- A função não possui parâmetros.
- O retorno é um ponteiro denominado de P alocado, do tipo **PilhaSimp**.

```
PilhaSimp* criar_pilha(void){  
  
    1     PilhaSimp* P = (PilhaSimp*)malloc(sizeof(PilhaSimp));  
  
    2     P -> topo = NULL;  
  
    3     return P;  
}
```

Figura 85. Função referente à atribuição NULL para a pilha simples.

*Descrição das funcionalidades* apresentadas figura 85:

1. Utilizando o conceito de alocação dinâmica, aloca-se o ponteiro denominado de **P**, do tipo PilhaSimp.
2. O **P -> topo** recebe o valor nulo, ou seja, aponta para NULL.
3. Retorna o ponteiro criado denominado de **P**, este tendo acesso ao ponteiro t.

#### Observação

Na função principal (**main()**) a função para criar a pilha deve ser chamada como:

**P = criar\_pilha();**

### 3.2.3. Inserir na pilha

#### O que fazer? Dicas...

- Deve alocar memória, de maneira dinâmica, para cada nó a ser inserido.
- O ponteiro **topo**, do tipo **ListaSimp**, composição do ponteiro **P**, do tipo **PilhaSimp**, aponta para o topo da pilha.  
\*\*\*Insere-se elementos sempre e somente no topo da pilha simples!
- Quando a pilha está vazia o ponteiro **topo**, referenciado pelo ponteiro **P**, (**P->topo**) aponta para o primeiro nó inserido.
- Após a realização da primeira inserção o ponteiro **topo**, referenciado pelo ponteiro **P**, apontará para o nó a ser inserido.

A função denominada **push\_pilha**, apresentada na figura 86, possui as seguintes características:

- A função possui como parâmetros o ponteiro **P**, do tipo **PilhaSimp**, que referencia o ponteiro **topo**, do tipo **ListaSimp**, e o valor a ser inserido, este do tipo inteiro, no campo de informação.
- A função é do tipo void, assim, nada retorna. As atribuições são realizadas dentro da função.

```
void push_pilha(PilhaSimp* P, int valor){
    1     ListaSimp* novo = (ListaSimp*)malloc(sizeof(ListaSimp));
    2     novo -> info = valor;
    3     novo -> prox = P -> topo;
    4     P -> topo = novo;
}
```

Figura 86. Função referente à inserir elementos na pilha simples.

*Descrição das funcionalidades* apresentadas figura 86:

1. Utilizando o conceito de alocação dinâmica, aloca-se o ponteiro denominado de **novo**, do tipo **ListaSimp**.
2. O **novo -> info** recebe um valor, do tipo inteiro, fornecido por meio do parâmetro da função, como pode ser observado na figura 87.
3. O **novo->prox** aponta para onde o ponteiro **topo**, referenciado pelo ponteiro **P**, aponta.

4. O ponteiro topo, referenciado pelo ponteiro P, aponta para o novo nó criado.  
Na função principal (**main()**) deve constar (figura 87):

```
main(void){
    1     PilhaSimp* P;
    2     P = criar_pilha();
    3     push_pilha(P, <valor a ser inserido>);
}
```

Figura 87. Função principal chamando a função criar\_pilha e inserir elementos na pilha simples (push\_pilha).

*Descrição das funcionalidades* apresentadas na figura 87:

1. Cria-se um ponteiro P, do tipo PilhaSimp, o qual referenciará o início ou chamado topo da pilha (P -> topo).
2. Chamar a função criar\_pilha para atribuir o primeiro valor, o qual é nulo, a pilha. O ponteiro P declarado recebe o retorno da função criar\_pilha.
3. Chamar a função push\_pilha passando como parâmetros o ponteiro P e o valor a ser inserido. Como visto não há retorno desta função.

Para melhor entendimento, graficamente, realiza-se um exemplo a seguir, utilizando o conceito *teste de mesa*.

**Exemplo**

Inserir os seguintes elementos 3 e 2 .

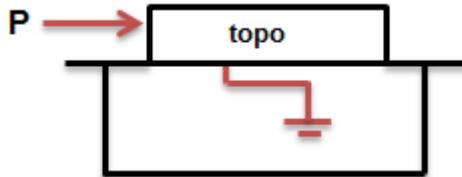
**1º PASSO:** Na função principal colocar os seguintes comandos apresentados na figura 88:

```
main(void){
    PilhaSimp* P;
    P = criar_pilha();
    push_pilha(P, 3);
    push_pilha(P, 2);
}
```

Figura 88. Comandos que devem ser colocados na função principal.

**2º PASSO:** Quando inicializada a função principal o ponteiro P, tipo PilhaSimp, é declarado.

**3º PASSO:** O ponteiro P recebe o retorno da função criar\_pilha, o qual será NULL para o ponteiro topo.



**4º PASSO:** a função push\_pilha(P, 3) realiza os seguintes procedimentos:

- Aloca-se um novo nó.



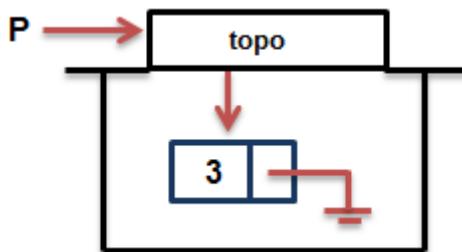
- Este nó recebe o valor 3 no campo de informação (novo -> info).



- O campo do endereço seguinte do nó criado (novo -> prox) aponta para onde o ponteiro topo, referenciado pelo ponteiro P, aponta, ou seja, para NULL.



- O ponteiro topo, referenciado pelo ponteiro P, aponta para o nó criado.



**5º PASSO:** a função push\_pilha(P, 2) realiza os seguintes procedimentos:

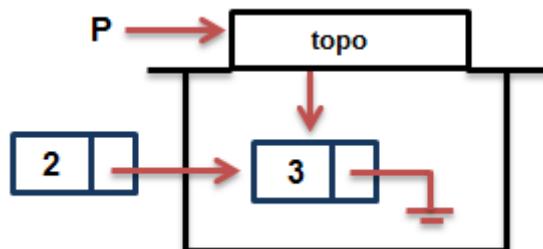
- Aloca-se um novo nó.



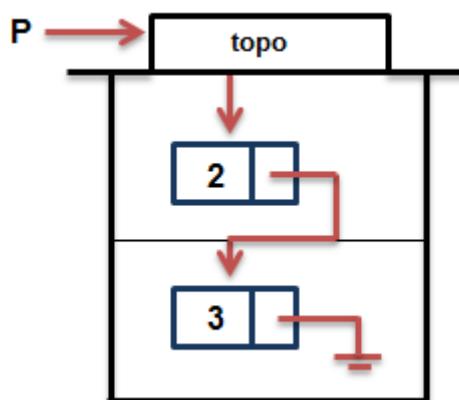
- Este nó recebe o valor 2 no campo de informação (novo -> info).



- O campo do endereço seguinte do nó criado (novo -> prox) aponta para onde o ponteiro topo, referenciado pelo ponteiro P, aponta, ou seja, para o elemento de valor 3.



- O ponteiro topo, referenciado pelo ponteiro P, aponta para o nó criado.



#### 3.2.4. Averiguar se a pilha está vazia

O que fazer? Dicas...

Verificar se o topo da pilha para NULL.

A função denominada **pilha\_vazia**, apresentada na figura 89, possui as seguintes características:

- A função possui como parâmetro o ponteiro P.
- O retorno é do tipo **booleano**, retornando valor 1 (verdadeiro) quando a pilha estiver vazia, caso contrário, retorna valor 0 (falso).

```

int pilha_vazia(PilhaSimp* P){
    return (P -> topo == NULL);
}

```

Figura 89. Função que verifica se a pilha está vazia.

### 3.2.5. Imprimir os elementos

#### O que fazer? Dicas...

- Percorrer a pilha imprimindo os valores de cada nó.
- Não manipular o ponteiro topo para não perder a referência do inicio da pilha

A função denominada **imprimir**, apresentada na figura 90, possui as seguintes características:

- A função possui como parâmetro o ponteiro P, o qual referencia o ponteiro topo.
- O retorno é do tipo void, retornando na tela os valores contidos na pilha.

```

void imprimir (PilhaSimp* P){

    1     ListaSimp* q = P -> topo;

    2     if(pilha_vazia(P))
    2         printf("\nPilha vazia!\n");
    3     else{
    3         while (q != NULL) {
    3             printf("%d ", q -> info);
    3             q = q -> prox;
    4         }
    5     }
}

```

Figura 90. Função imprimir.

*Descrição das funcionalidades* apresentadas na figura 90:

1. Declara-se um ponteiro auxiliar denominado de q, do tipo ListaSimp, o qual aponta para onde o ponteiro topo, referenciado pelo P, aponta, ou seja, para o primeiro elemento da pilha.

2. Dentro da condicional chama-se a função pilha\_vazia, caso retorne o valor 1 a pilha está vazia e, assim, imprimi-se “Pilha vazia!”.
3. Senão entra no laço while:
  - Se o ponteiro q for diferente de NULL imprime-se o valor contido no campo de informação do elemento apontado pelo ponteiro q.
  - O incremento do ponteiro q é representado por  $q = q \rightarrow prox$ .
  - Enquanto o ponteiro q for diferente de NULL ainda há elemento(s) na pilha.

**Observação**

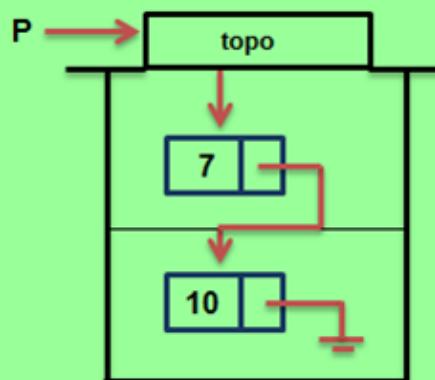
Na função principal (`main()`) a função para imprimir a pilha deve ser chamada como:

`imprimir(P);`

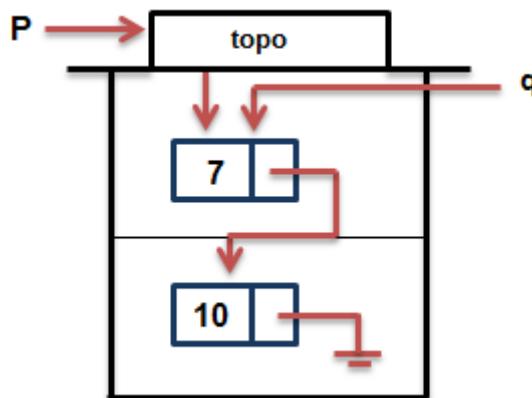
Para melhor entendimento, graficamente, realiza-se um exemplo a seguir, utilizando o conceito *teste de mesa*.

**Exemplo**

Considerando a seguinte pilha:



**1º PASSO:** O ponteiro q aponta para onde o ponteiro topo, referenciado pelo ponteiro P, aponta, ou seja, para o primeiro elemento da pilha.

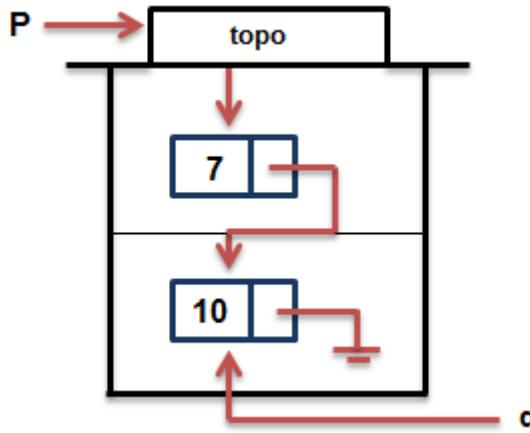


**2º PASSO:** Entra na condicional (if).

- Chama-se a função pilha\_vazia, a pilha está vazia? NÃO!! Pois o ponteiro topo, referenciado pelo ponteiro P, aponta para o nó de valor 7, ou seja, é diferente de NULL.

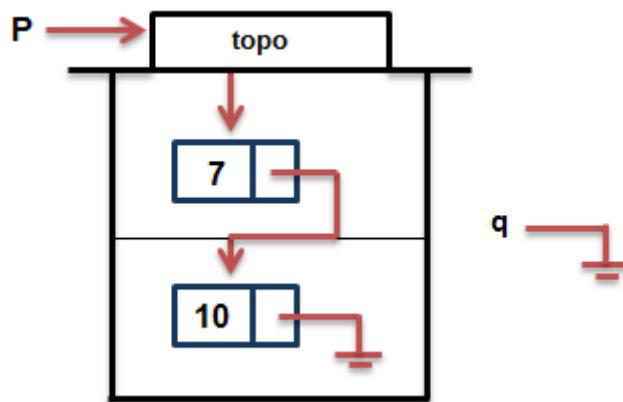
**3º PASSO:** Entra na condicional else. Percorre-se o laço while.

- O ponteiro q é diferente de NULL, ou seja, não aponta para NULL? SIM, assim imprime-se o valor 7 contido no elemento apontado pelo ponteiro q ( $q \rightarrow \text{info}$ ).
- O incremento do ponteiro q é dado por  $q = q \rightarrow \text{prox}$ , onde  $q \rightarrow \text{prox}$  é o elemento de valor 10. Assim, o ponteiro q aponta para este nó.



**4º PASSO:** Retorna ao laço while:

- O ponteiro q é diferente de NULL, ou seja, não aponta para NULL? SIM, assim imprime-se o valor 10 contido no elemento apontado pelo ponteiro q ( $q \rightarrow \text{info}$ ).
- O incremento do ponteiro q é dado por  $q = q \rightarrow \text{prox}$ , onde  $q \rightarrow \text{prox}$  possui valor nulo (NULL). Assim, o ponteiro q aponta para NULL.



**5º PASSO:** Retorna ao laço while:

- O ponteiro q é diferente de NULL, ou seja, não aponta para NULL? NÃO, ou seja, imprimiram-se todos os elementos contidos na pilha.

**6º PASSO:** Termina o laço, assim, encerra-se a impressão!!

### 3.2.6. Buscar um determinado elemento na pilha

#### O que fazer? Dicas...

- Percorrer a pilha comparando o valor que o nó possui com o valor a ser buscado.
- Caso encontre, retorna o nó que contem o valor. Caso contrário, retorna NULL.

A função denominada **buscar\_elemento**, apresentada na figura 91, possui as seguintes características:

- A função possui como parâmetro o valor a ser procurado e o ponteiro P, o qual possui acesso ao ponteiro topo.
- O retorno é do tipo PilhaSimp. Caso o elemento seja encontrado o ponteiro q aponta para ele, assim, retorna este nó, caso contrário, retorna o valor nulo (NULL).

#### Observação

Na função principal (**main()**) a função buscar deve ser chamada como:

**ListaSimp\* valor\_obtido = buscar\_elemento(P, <valor a ser procurado>);**

```
PilhaSimp* buscar_elemento (PilhaSimp* P, int valor ){
```

```
    ListaSimp* q;
    1
    for(q = P -> topo; q != NULL; q = q -> prox) {
        2
        if(q -> info == valor)
            3
            return q;
        }
    5
    return NULL;
}
```

Figura 91. Função buscar elemento na pilha.

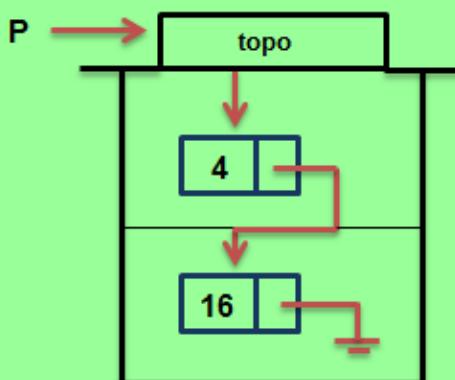
*Descrição das funcionalidades* apresentadas na figura 91:

1. O ponteiro topo, referenciado pelo ponteiro P, aponta para o primeiro nó da pilha, este não deve ser manipulado para não perder a referência, então, utiliza-se um ponteiro auxiliar denominado de q, o qual aponta para onde P -> topo aponta, ou seja, para o primeiro nó;
2. Enquanto o ponteiro q for diferente de NULL ainda há elemento(s) na pilha.
3. O incremento do ponteiro q é representado por  $q = q -> prox$ .
4. Se o campo de informação do nó apontado pelo ponteiro q ( $q->info$ ) for igual ao valor procurado retorna-se o ponteiro q, o qual terá acesso ao campo de informação (info) e ao campo do endereço seguinte (prox), já que q é do tipo ListaSimp.
5. Caso não encontrado, retorna-se o valor nulo (NULL).

Para melhor entendimento, graficamente, realiza-se um exemplo a seguir, utilizando o conceito *teste de mesa*.

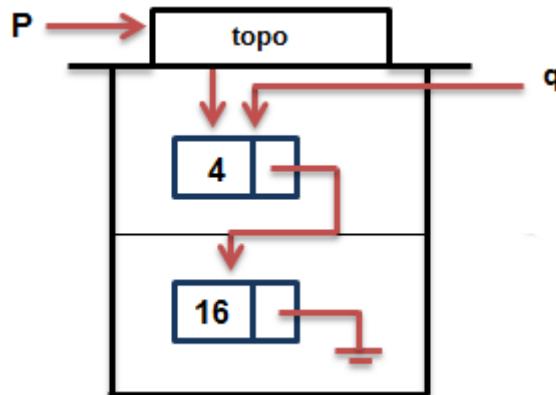
### Exemplo

Considerando a seguinte pilha e o número 16 a ser procurado:



**1º PASSO:** Declara-se um ponteiro auxiliar denominado de q e percorrer o laço for:

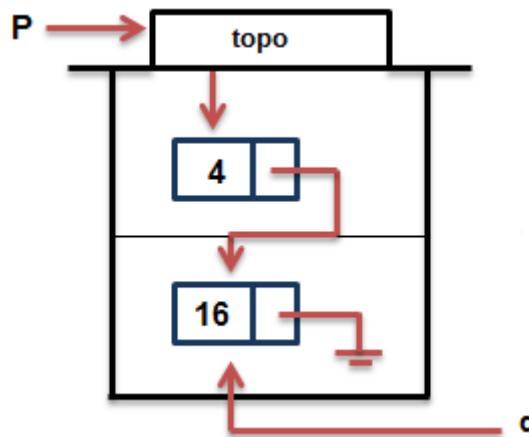
- O ponteiro q aponta para onde o ponteiro topo, referenciado pelo ponteiro P ( $P \rightarrow topo$ ) aponta, ou seja, para o primeiro nó.



- O ponteiro q é diferente de NULL? SIM!! O ponteiro q aponta para o nó de valor 4.
- Entra na condicional if.
- O ponteiro q que na região info ( $q \rightarrow info$ ) vale 4, é igual ao valor procurado que é 16, passado como parâmetro? NÃO!!

**2º PASSO:** Retorna ao laço for.

- O incremento do ponteiro q é dado por  $q = q \rightarrow prox$ , onde  $q \rightarrow prox$  é o elemento de valor 16. Assim, o ponteiro p aponta para este nó.



- O ponteiro q é diferente de NULL? SIM!! O ponteiro q aponta para o nó de valor 16.
- Entra na condicional if.
- O ponteiro q que na região info ( $q \rightarrow info$ ) vale 16, é igual ao valor procurado que é 16, passado como parâmetro? SIM!!
- Termina o laço for.

**3º PASSO:** Retorna o nó apontado pelo ponteiro q, o qual possui valor 16.

### 3.2.7. Liberar todos os elementos

#### O que fazer? Dicas...

- Percorrer a pilha desalocando cada elemento.
- Utilizar um ponteiro auxiliar que aponte para o topo da pilha, e ainda, utilizar um outro ponteiro que aponte para o nó seguinte ao ponteiro auxiliar criado.

A função denominada **liberar\_pilha**, apresentada na figura 92, possui as seguintes características:

- A função possui como parâmetro o ponteiro P.
- Não há retorno para a função principal, é do tipo void.

```
void liberar_pilha (PilhaSimp* P){
```

```
1    ListaSimp* q = P->topo;
2
3    while (q != NULL) {
4        ListaSimp* t = q->prox;
5        free(q);
6        q = t;
7    }
8    free(P);
9}
```

Figura 92. Função liberar a pilha.

*Descrição das funcionalidades* apresentadas na figura 92:

1. Declara-se um ponteiro auxiliar denominado de q, do tipo ListaSimp, o qual aponta para onde o ponteiro topo, referenciado pelo P, aponta, ou seja, para o primeiro elemento da pilha.
2. No laço while, enquanto o ponteiro q for diferente de NULL ainda há elemento(s) na pilha, então, executa-se os comandos encontrados dentro do laço.

3. Utiliza-se um ponteiro auxiliar denominado de t que aponta para o nó a frente do ponteiro q ou para NULL, caso o ponteiro q aponte para o último nó.
4. Desaloca o nó apontado pelo ponteiro q.
5. O ponteiro q aponta para onde o ponteiro t aponta.
6. Após desalocar todos os elementos desaloca-se o ponteiro P.

**Observação**

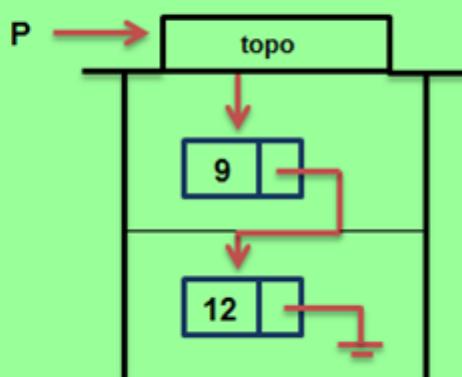
Na função principal (**main()**) a função liberar deve ser chamada como:

**liberar\_pilha(P);**

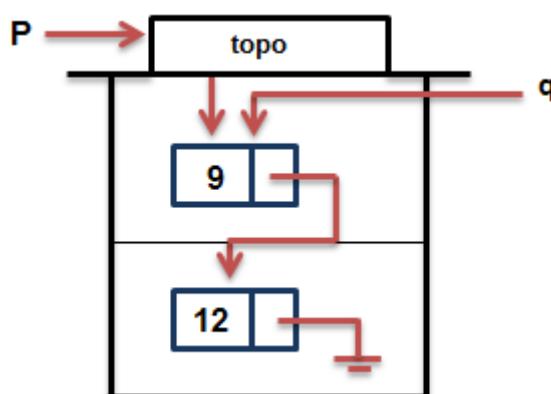
Para melhor entendimento, graficamente, realiza-se um exemplo a seguir, utilizando o conceito *teste de mesa*.

**Exemplo**

Considerando a seguinte pilha:

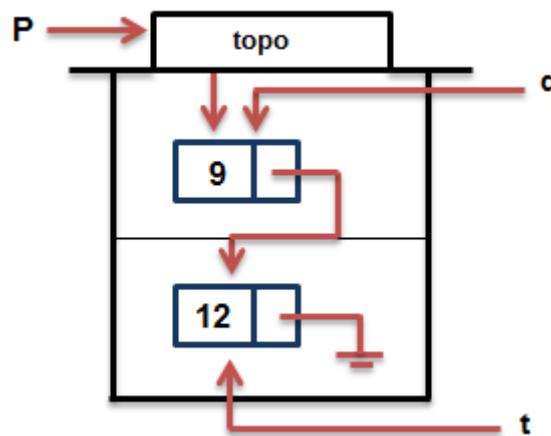


**1º PASSO:** Ponteiro q aponta para onde o ponteiro topo, referenciado pelo ponteiro P, aponta, ou seja, para o início ou topo da pilha.

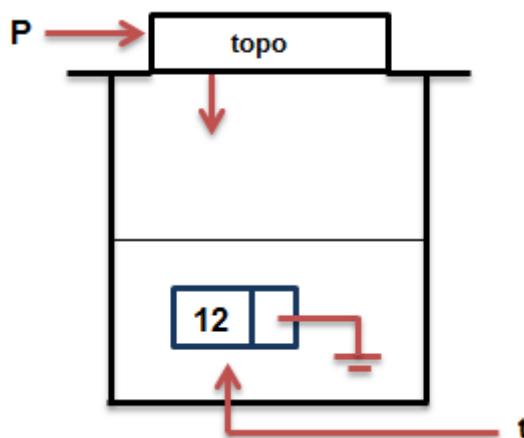


**2º PASSO:** Entra no laço while.

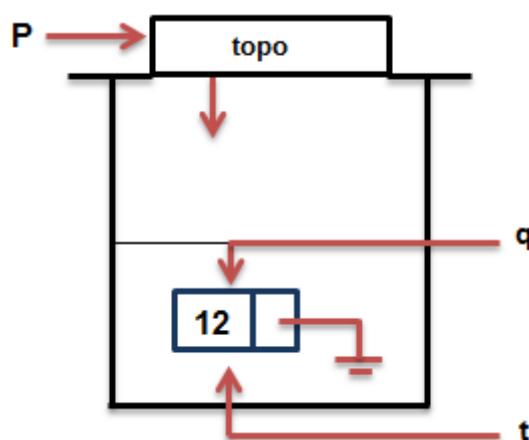
- O ponteiro  $q$  é diferente de NULL? SIM!! O ponteiro  $q$  aponta para o nó de valor 9.
- O ponteiro  $t$  aponta para o próximo nó do ponteiro  $q$  ( $q->prox$ ). No caso, apontará para o nó de valor 12.



- Desaloca o nó apontado pelo ponteiro  $q$ .

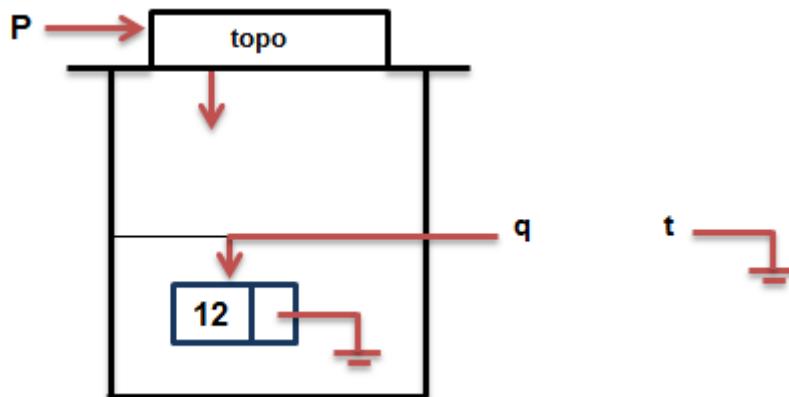


- O ponteiro  $q$  aponta para onde o ponteiro  $t$  aponta.



**3º PASSO:** Retorna ao laço while.

- O ponteiro q é diferente de NULL? SIM!! O ponteiro q aponta para o nó de valor 12.
- O ponteiro t aponta para o próximo nó do ponteiro q ( $q->prox$ ). No caso, apontará para NULL.



- Desaloca o nó apontado pelo ponteiro q.



- O ponteiro q aponta para onde o ponteiro t aponta.



**4º PASSO:** Retorna ao laço while.

- O ponteiro q é diferente de NULL? NÃO!! O ponteiro q aponta para NULL.
- Terminam as iterações do laço while.

**5º PASSO:** Desaloca o ponteiro P. Assim, liberou toda a pilha.

### 3.2.8. Retirar um nó

#### O que fazer? Dicas...

- Utilizar um ponteiro auxiliar, o qual deve apontar para o primeiro elemento da pilha.
- O ponteiro que referencia o inicio da pilha deve apontar para o segundo elemento, o qual se tornará o primeiro após a retirada do nó.

\*\*\* Em uma pilha simples retira-se sempre e somente do inicio!!!

A função denominada **pop\_pilha**, apresentada na figura 93, possui as seguintes características:

- A função possui como parâmetro o ponteiro P, do tipo PilhaSimp.
- O retorno é do tipo inteiro, retornando o valor contido no campo de informação (*info*) do nó a ser removido.

```
int pop_pilha (PilhaSimp* P){
    1      ListaSimp* t;
    2      int v;

    3      if(pilha_vazia(P)){
    3          printf("\nPilha vazia!\n");
    3          exit(1);
    }

    4      t = P -> topo;
    5      v = t -> info;
    6      P -> topo = t -> prox;

    7      free(t);
    8      return v;
}
```

Figura 93. Função para retirar o primeiro elemento pilha.

*Descrição das funcionalidades* apresentadas na figura 93:

1. O ponteiro P possui acesso ao ponteiro topo. Declara-se um ponteiro auxiliar denominado de t para que não ocorra a perda da referência do ponteiro topo, referenciado pelo ponteiro P.
2. Declara-se uma variável, do tipo inteiro, denominada de v. Esta armazenará o valor do primeiro nó que será retirado.
3. Chama-se a função pilha\_vazia, se estiver vazia apresentará uma mensagem na tela e finalizará a chamada da função pop\_pilha.
4. O ponteiro t aponta para o início da pilha ( $t \rightarrow \text{topo}$ ), ou seja, para o primeiro elemento.
5. A variável v armazena o valor do campo de informação do nó apontado pelo ponteiro t.
6. O ponteiro topo, referenciado pelo ponteiro P, aponta para o nó seguinte em relação ao ponteiro t, ou seja, para o segundo elemento, ou para NULL, caso a pilha tenha somente um nó antes da chamada da função pop\_pilha.
7. Desaloca o nó apontado pelo ponteiro t.
8. Retorna o valor contido no campo de informação do elemento removido.

#### Observação

Na função principal (**main()**) a função retirar deve ser chamada como:

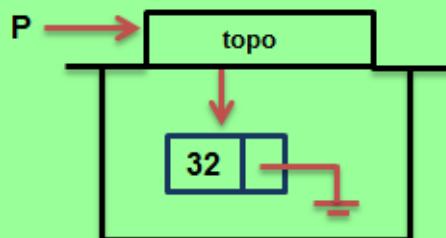
```
int valor_retirado = pop_pilha(P);
```

Para melhor entendimento, graficamente, realiza-se dois exemplos a seguir.

- 1. A pilha possui um único elemento.**
- 2. A pilha possui três elementos.**

#### Exemplo 1

Considerando a seguinte pilha:

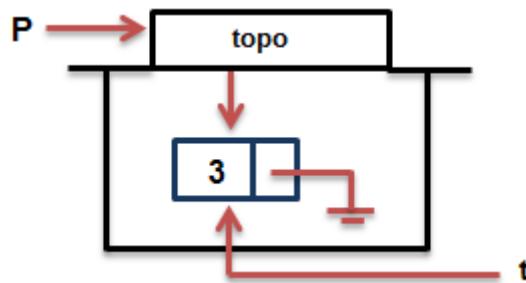


**1º PASSO:** Declaração do ponteiro t e da variável v.

**2º PASSO:** Entra na condicional if.

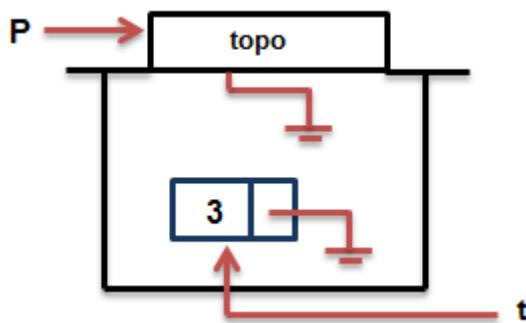
- Chama-se a função pilha\_vazia, contudo, retorna 0, ou seja, a pilha não está vazia, possui um elemento com valor 32.

**3º PASSO:** O ponteiro t aponta para o primeiro elemento, este apontado pelo ponteiro topo, referenciado pelo ponteiro P ( $P \rightarrow topo$ ).

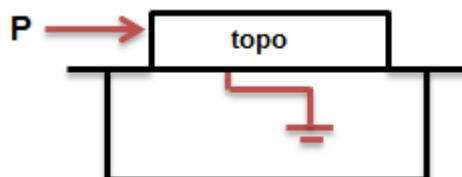


**4º PASSO:** A variável v armazena o valor 32 contido no campo de informação do nó apontado pelo ponteiro t ( $t \rightarrow info$ ).

**5º PASSO:** O ponteiro topo, referenciado pelo ponteiro P, aponta para o nó seguinte em relação ao ponteiro t ( $t \rightarrow prox$ ), o qual é para NULL .



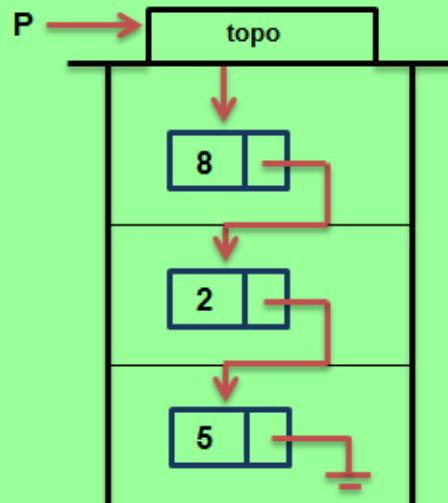
**6º PASSO:** Desaloca o nó apontado pelo ponteiro t.



**7º PASSO:** Retorna o valor 32 contido na variável v.

Exemplo 2

Considerando a seguinte pilha:

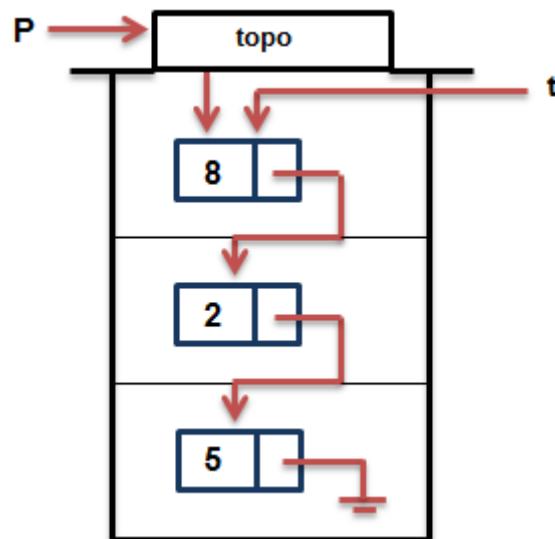


**1º PASSO:** Declaração do ponteiro t e da variável v.

**2º PASSO:** Entra na condicional if.

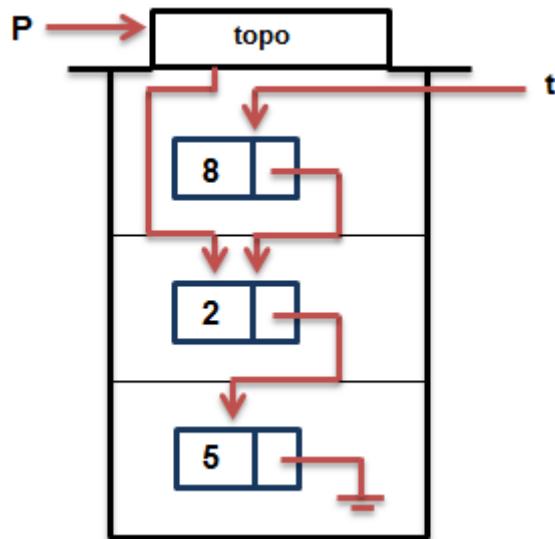
- Chama-se a função pilha\_vazia, contudo, retorna 0, ou seja, a pilha não está vazia.

**3º PASSO:** O ponteiro t aponta para o primeiro elemento, este apontado pelo ponteiro topo, referenciado pelo ponteiro P ( $P \rightarrow topo$ ).

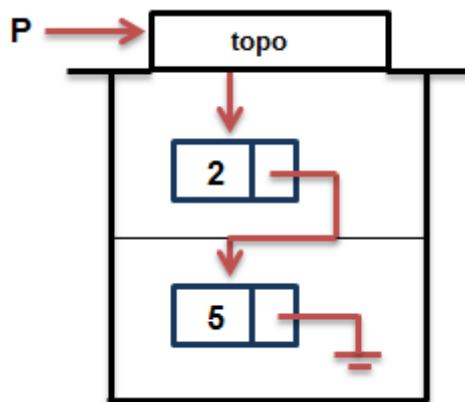


**4º PASSO:** A variável v armazena o valor 8 contido no campo de informação do nó apontado pelo ponteiro t ( $t \rightarrow info$ ).

**5º PASSO:** O ponteiro topo, referenciado pelo ponteiro P, aponta para o nó seguinte em relação ao ponteiro t ( $t \rightarrow \text{prox}$ ), o qual é o elemento de valor 2 .



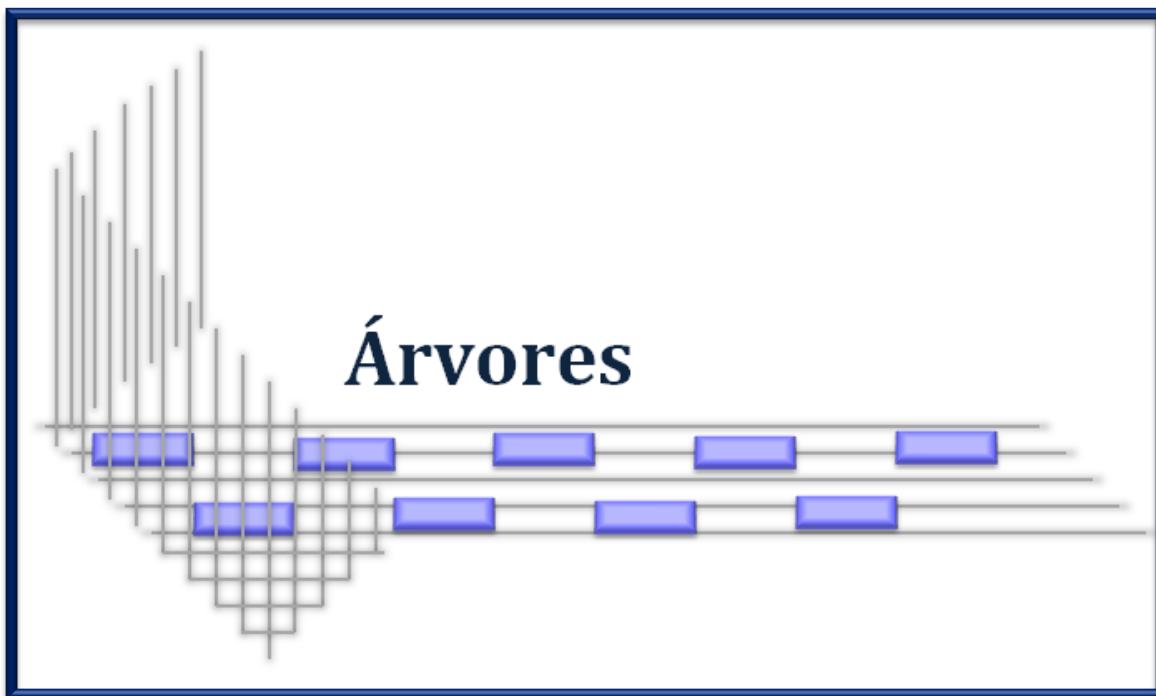
**6º PASSO:** Desaloca o nó apontado pelo ponteiro t.



**7º PASSO:** Retorna o valor 8 contido na variável v.

### 3.1. Exercícios

- 1) Implemente uma função que verifica se os elementos de uma pilha estão ordenados de forma crescente.
- 2) Faça uma função que receba uma Pilha P1 e retorne esta com os elementos invertidos.
- 3) Construa uma função que retorne a quantidade de números primos em uma pilha e retireá-los dela.
- 4) Implemente uma função que verifique se duas pilhas são iguais.
- 5) Simule, graficamente, a inserção dos seguintes nós: **2 5 7 1 8**
- 6) Implemente uma função que receba uma Pilha F1 e retorne esta sem os nós com valores pares. Para isso, deverá ser utilizada uma fila. Exemplo: Pilha P1 = {3, 2, 8, 1}, após a execução da função a Pilha P1 ficará {3, 1}.
- 7) Simule, graficamente, a remoção do nó de valor 6 da pilha {2, 4, 6, 3}.



#### 4. Árvores

Árvores são estruturas de dados que utilizam o conceito de recursividade. Podem ser classificadas como árvore:

- **Binária.** A inserção dos nós é descrita em uma única função principal (main), definida pelo usuário, pois os nós podem ser inseridos sem ordem e duplicados, como mostra a figura 94.

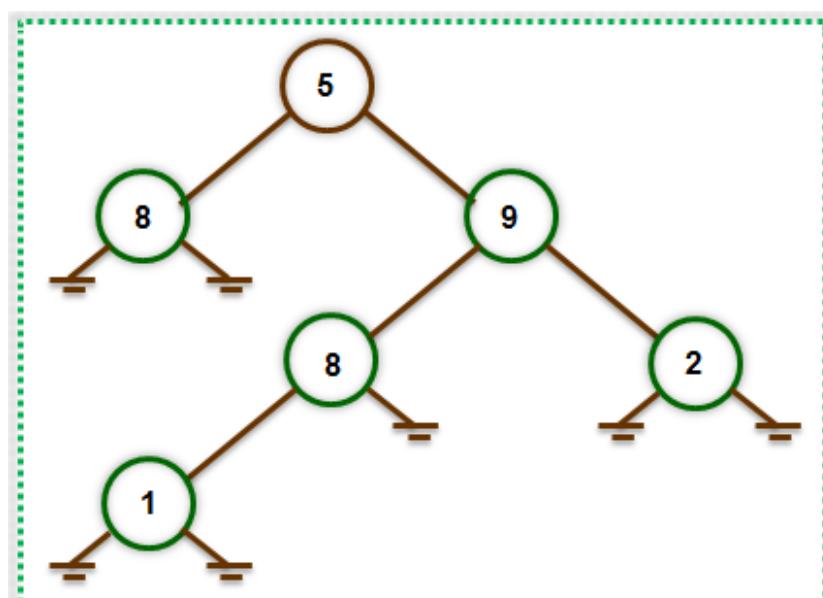


Figura 94. Exemplo de uma Árvore Binária.

- **Binária de Busca.** A inserção dos nós é descrita conforme a necessidade do usuário, ou seja, a partir do primeiro valor passado como parâmetro os demais inseridos serão comparados a este e enviados para a sub-árvores devidas, como mostra a figura 95.

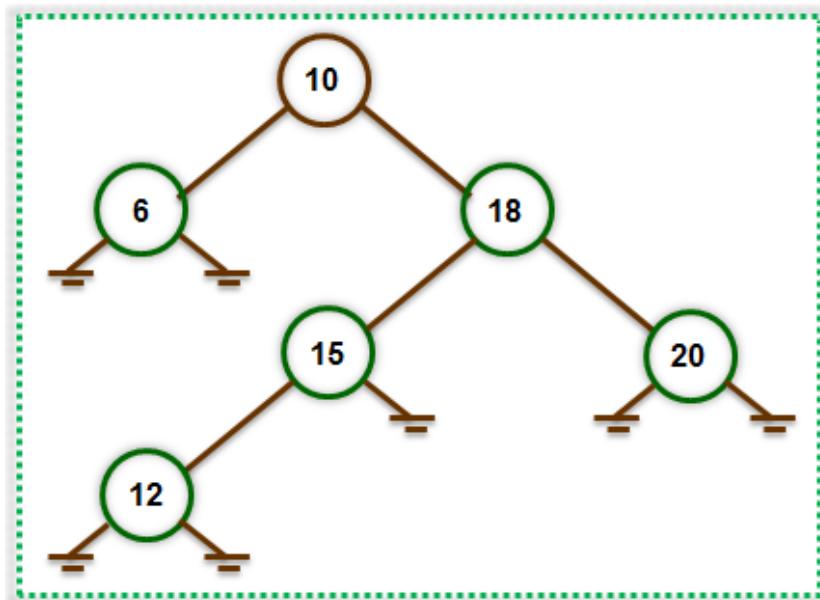


Figura 95. Exemplo de uma Árvore Binária de Busca.

- **AVL.** O processo de inserção é semelhante ao da árvore binária de busca, contudo, ocorre o balanceamento da árvore, ou seja, o caso de uma árvore degenerada não ocorre, como mostra a figura 96.

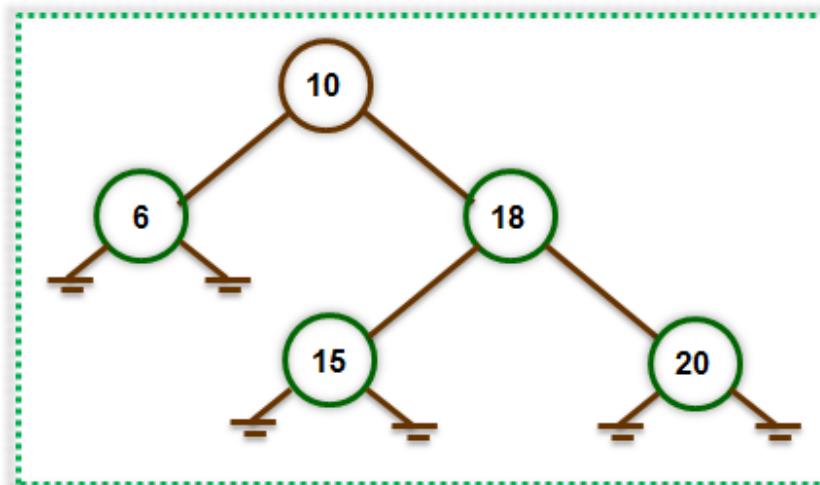


Figura 96. Exemplo de uma Árvore AVL.

- **B.** Não é necessário formar a árvore a partir de uma raiz como nas demais árvores apresentadas anteriormente. Maior quantidade de ponteiros que pode ser armazenado em um nó, , como mostra a figura 97.

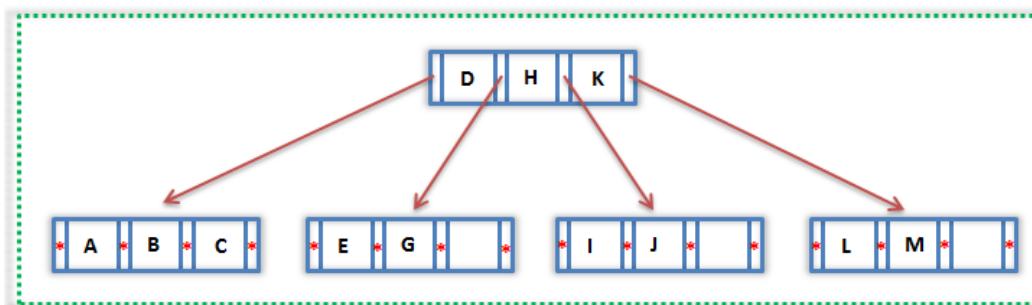


Figura 97. Exemplo de uma Árvore B.

Assim, as seguintes características, apresentadas no quadro 2, demostram as diferenças entre estas.

Quadro 2. Comparação considerando três tipos de árvore.

Árvore			
Características	Binária	Binária de Busca	AVL
Ocorrência de elementos duplicados	Sim	Não	Não
Ordenação	Não	Sim	Sim
Balanceamento	Não	Não	Sim

### Observação

Toda árvore AVL é uma árvore binária de busca otimizada.

Toda árvore binária de busca é uma árvore binária otimizada.

As árvores binária, binária de busca e AVL são composta por um conjunto de elementos, denominados de nós da árvore. Cada um pode possuir uma sub-árvore, direita (**sad**) ou esquerda (**sae**), as duas (como mostram as figura 98), ou nenhuma, sendo assim, denominada de **árvore vazia**, como mostra a figura 99.

Os nós que as compõem podem ser classificados como:

- **Raiz.** É o nó principal, do qual partem os demais nós denominados de *nós raízes*.
- **Nós raízes.** São os nós que se originaram do nó principal denominado de *Raiz*.
- **Nós internos.** São os nós que possuem filhos, mas não são filhos da *raiz*.

- **Folhas ou nós externos.** São os nós que não possuem sub-árvores, sem filhos, como mostra a figura 98.

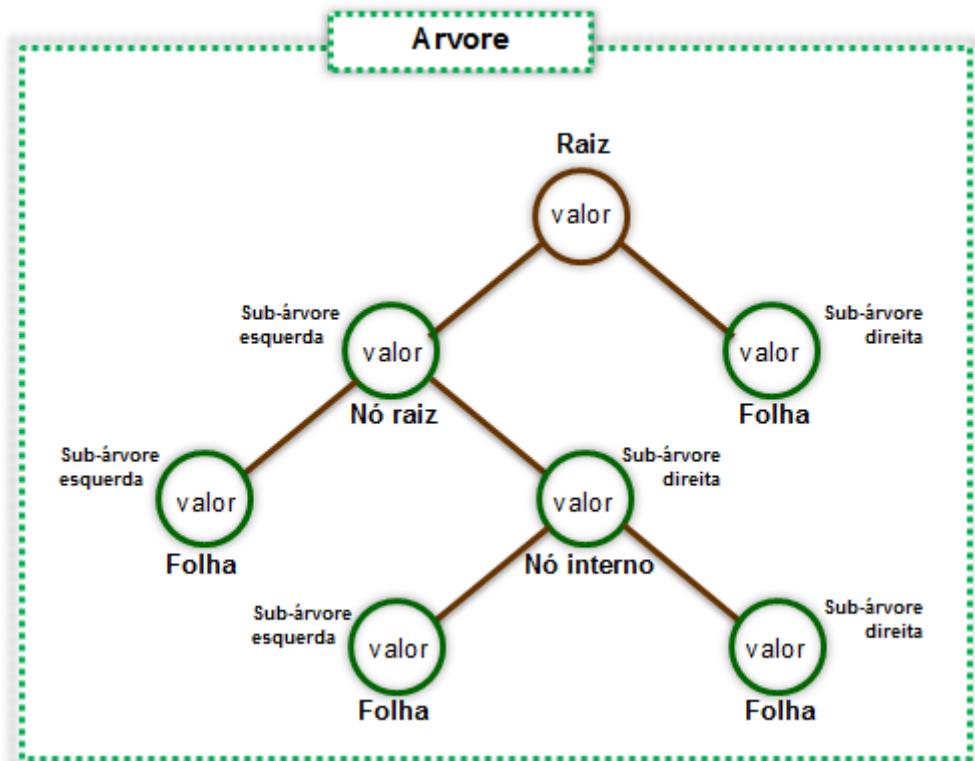


Figura 98. Nós, que compõem uma Árvore Binária de Busca ou AVL, classificados.

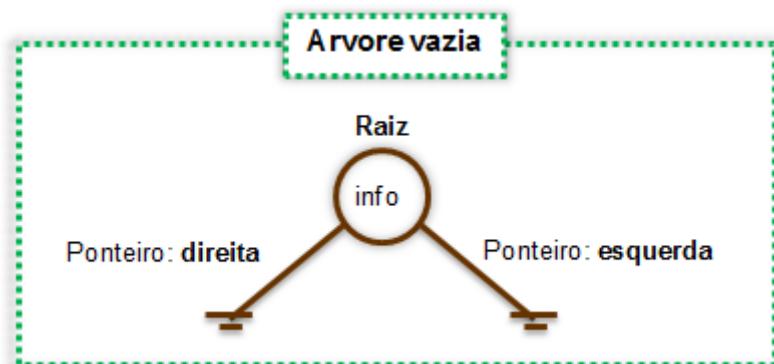


Figura 99. Árvore Binária, Binária de Busca ou AVL, vazia.

## 4.1. Árvore Binária

### 4.1.1. Conceitos

Uma Árvore Binária é caracterizada por **três** fatores:

- Pode ocorrer elementos duplicados.
- Não ordenada.
- Não balanceada.

Entende-se como ordenada quando os nós pertencentes à sub-árvore esquerda possuem valores menores que o valor encontrado na raiz enquanto os nós pertencentes à sub-árvore direita possuem valores. E como balanceada quando as alturas das duas sub-árvores de cada nó nunca diferem em mais de 1.

### 4.1.2. Criação e Manipulação

Pode-se programar a criação da árvore binária (AB) e estabelecer as devidas funcionalidades a ela, entre estas se tem:

- 4.1.2.1. Criação de um Tipo Abstrato de Dado (TAD).
- 4.1.2.2. Criar a árvore binária.
- 4.1.2.3. Inserir elementos.
- 4.1.2.4. Averiguar se a árvore binária está vazia.
- 4.1.2.5. Imprimir os elementos.
- 4.1.2.6. Buscar um determinado elemento.
- 4.1.2.7. Liberar todos os elementos.

#### 4.1.2.1. Criação de um tipo abstrato de dado (TAD)

Criação de um novo tipo de dado, como mostra a figura 100.

```
typedef struct arv_binaria{
    int info;
    struct arv_binaria* esquerda;
    struct arv_binaria* direita;
}Arv_B;
```

Figura 100. Estrutura de um novo tipo de dado do tipo Arv\_B.

**Descrição das funcionalidades** apresentadas na figura 100:

1. Declarou-se uma variável (info), do tipo inteiro, a qual armazena o valor contido no elemento (figura 101).
2. Declarou-se o ponteiro denominado de esquerda, do tipo struct arv\_binaria, o qual se refere à sub-árvore esquerda (figura 101).
3. Declarou-se o ponteiro denominado de direita, do tipo struct arv\_binaria, o qual se refere à sub-árvore direita (figura 101).

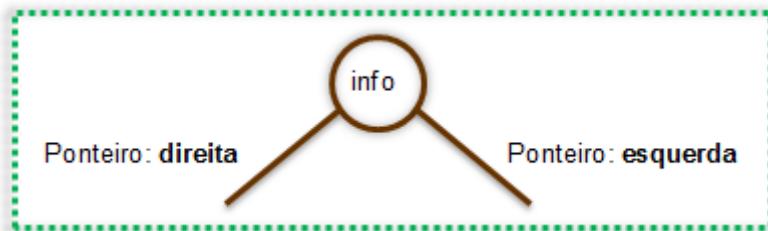


Figura 101. Regiões referentes ao ponteiro *esquerda*, ao ponteiro *direita* e a variável *info*.

#### 4.1.2.2. Criar a árvore binária

##### O que fazer? Dicas...

Toda árvore deve inicializar vazia!!!

A função denominada **criar\_arv**, apresentada na figura 102, possui as seguintes características (graficamente, está função está representada na figura 99, apresentada anteriormente):

- A função não possui parâmetros.
- O retorno é do tipo **Arv\_B**, retornando o valor nulo (NULL).

```

Arv_B* criar_arv(void){
    return NULL;
}
    
```

Figura 102. Função referente à atribuição NULL para a árvore binária.

##### Observação

Na função principal (**main()**) a função para criar a árvore deve ser chamada como:

**AB = criar\_arv();**

#### 4.1.2.3. Inserir elementos

##### O que fazer? Dicas...

- Deve alocar memória, de maneira dinâmica, para cada nó a ser inserido.
- O ponteiro **esquerda** e **direita**, do tipo **Arv\_B**, composição do ponteiro **AB**, apontam para a sub-árvore esquerda e direita, respectivamente, ou para **NULL**.

A função denominada **inserir\_nos**, apresentada na figura 103, possui as seguintes características:

- A função possui como parâmetros o ponteiro denominado de **sae** e outro de **sad**, os quais recebem as sub-árvores esquerda e direita, respectivamente, do tipo **Arv\_B**, e o valor a ser inserido, do tipo inteiro.
- O retorno da função é do tipo **Arv\_B**, retornando o novo nó, assim, encadeando o elemento na árvore existente.

```
Arv_B* inserir_nos(int valor, Arv_B* sae, Arv_B* sad){
```

```
1      Arv_B* novo = (Arv_B*)malloc(sizeof(Arv_B));
2      novo -> info = valor;
3      novo -> esquerda = sae;
4      novo -> direita = sad;
5      return novo;
}
```

Figura 103. Função referente à inserir elementos na árvore binária.

*Descrição das funcionalidades* apresentadas figura 103:

1. Utilizando o conceito de alocação dinâmica, aloca-se um nó denominado de *novo* do tipo **Arv\_B**.
2. O *novo -> info* recebe o valor, do tipo inteiro, fornecido por meio do parâmetro da função.
3. O *novo -> esquerda* aponta para onde o ponteiro **sae** aponta, se for para **NULL** apontará para **NULL**.
4. O *novo -> direita* aponta para onde o ponteiro **sad** aponta, se for para **NULL** apontará para **NULL**.
5. Retorna o novo nó criado para o ponteiro **AB**.

Na função principal (**main()**) deve constar (figura 104):

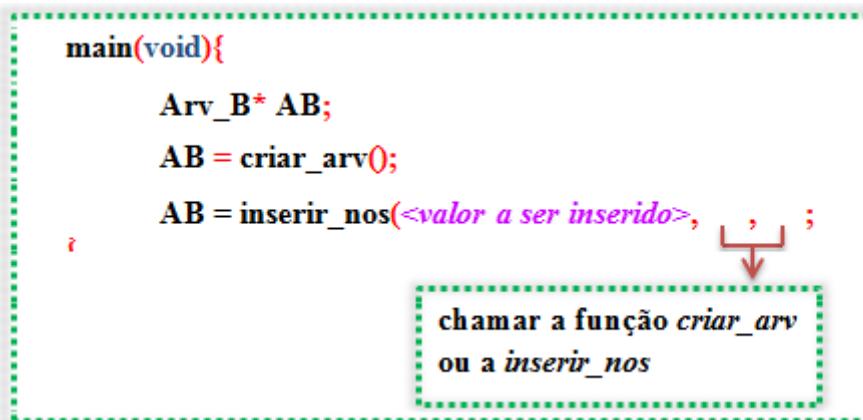


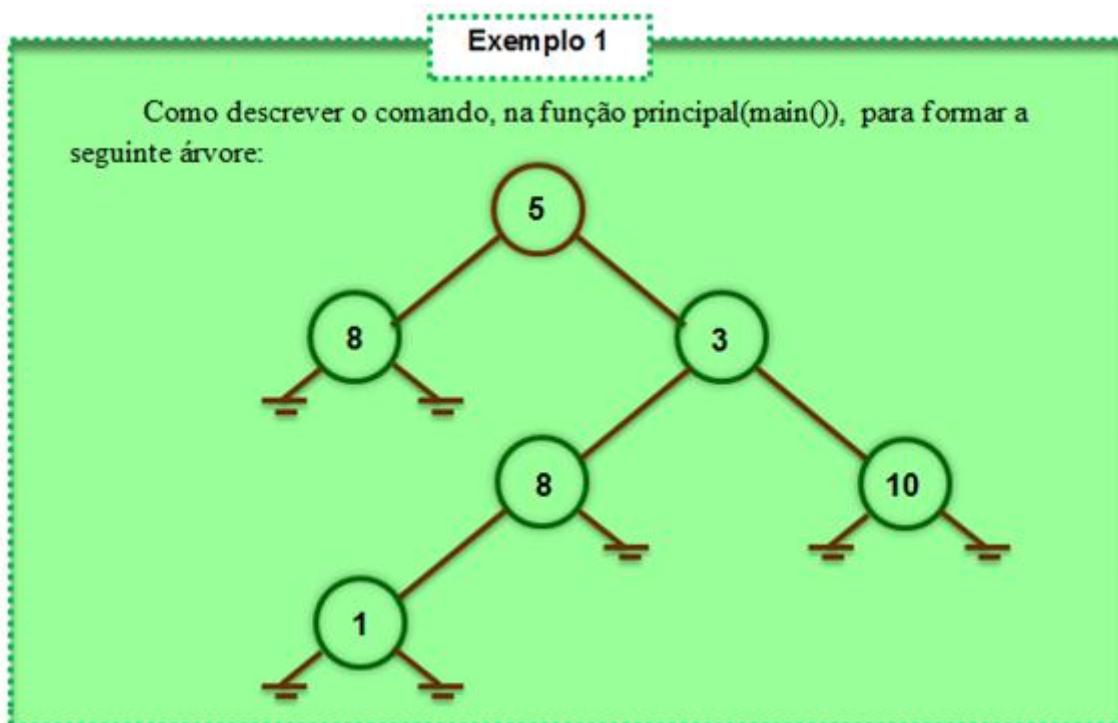
Figura 104. Função principal chamar a função *criar\_arv* e *inserir\_nos* na árvore binária.

*Descrição das funcionalidades* apresentadas na figura 104:

1. Cria-se um ponteiro AB, do tipo Arv\_B, o qual referenciará a raiz da árvore.
2. Chamar a função *criar\_arv* para atribuir o primeiro valor a árvore, o qual é nulo. O ponteiro AB declarado recebe o retorno da função *criar\_arv*.
3. Chamar a função *inserir\_nos* passando como parâmetros o valor a ser inserido, e duas chamadas recursivas para os dois outros parâmetros, as quais podem ser *criar\_arv* ou *inserir\_nos*.

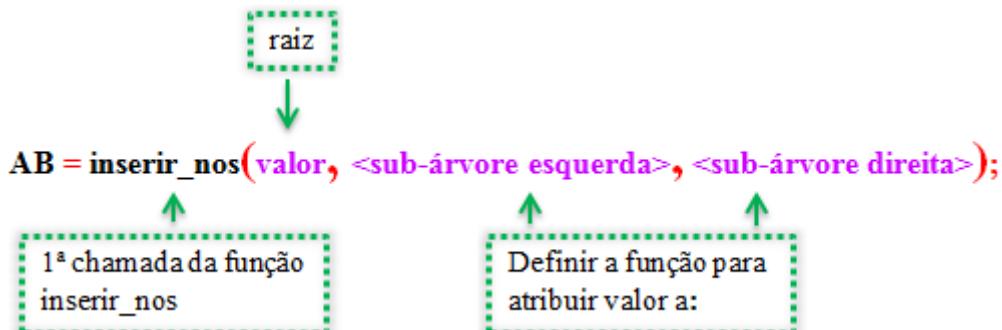
Para melhor entendimento, graficamente, realiza-se dois exemplos a seguir, utilizando o conceito *teste de mesa*.

- **Exemplo 1:** Como formar o comando na função principal (main()) para inserir os elementos na árvore.
- **Exemplo 2:** Funcionamento da função *inserir\_nos*.



Na função principal, para formar esta árvore, um único comando será executado.

- Como visto, na figura 104, utiliza-se os conceitos de recursividade. Assim, tem-se:



- Se o comando for: **AB = inserir\_nos(5, criar\_arv(), criar\_arv());** A árvore, apresentada na figura 105, será formada:

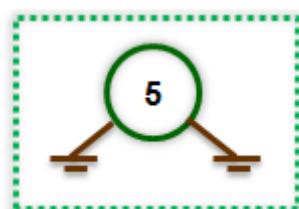


Figura 105 Inserção do primeiro elemento de valor 5, este sem filhos.

- Utilizando dos conceitos de recursividade a função inserir\_nos será chamada dentro da própria função. A região sublinhada a seguir se refere a sub-árvore esquerda, enquanto a sub-árvore direita não recebe inserção.

**AB = inserir\_nos(5, inserir\_nos(valor, <sub-árvore esquerda>, sub-árvore direita), criar\_arv());**

- Assim, para inserir o elemento de valor de 8, no lado esquerdo da árvore, o qual não possui filhos, na sub-árvore esquerda, região sublinhada, chama-se novamente a função inserir\_nos em vez de criar\_arv(). Como o elemento 8 não possui filhos tanto a sub-árvore esquerda quanto a direita receberão valor nulo (NULL), assim, chama-se a função criar\_arv(). O comando a seguir realiza esta função e forma a árvore apresentada na figura 106:

**AB = inserir\_nos(5, inserir\_nos(8, criar\_arv(), criar\_arv()), criar\_arv());**

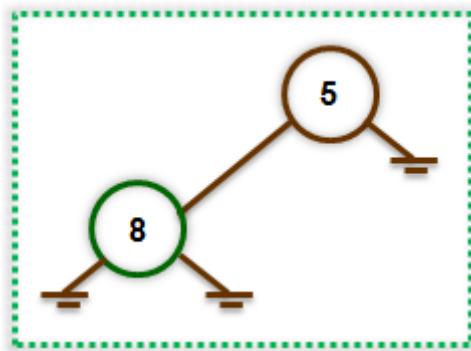


Figura 106. Inserção do elemento de valor 8, este sem filhos.

- Para a inserção do valor 3, no lado direito da árvore,, sem filhos, haverá uma alteração do comando anterior em vez de chamar a função criar\_arv, no local referente a sub-árvore direita do elemento de valor 5, será chamada a função inserir\_nos. O comando a seguir realiza esta função e forma a árvore apresentada na figura 107:

```

AB = inserir_nos(5,           Sub-árvore esquerda do nó 8           Sub-árvore direita do nó 8           Sub-árvore esquerda do nó 5
inserir_nos(8, criar_arv(), criar_arv()),                     |                     |
inserir_nos(3, criar_arv(), criar_arv()));                     |                     |
                                                                   |                     |
                                                 Sub-árvore esquerda do nó 3   Sub-árvore direita do nó 3   Sub-árvore direita do nó 5
  
```

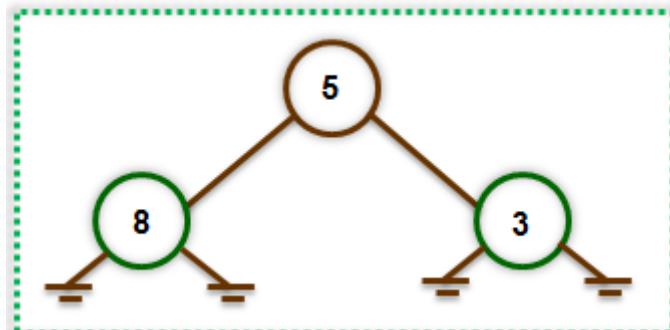


Figura 107. Inserção do elemento de valor 3, este sem filhos.

- Para a inserção do valor 8, no lado esquerdo do nó de valor 3, sem filhos, haverá uma alteração do comando anterior em vez de chamar a função criar\_arv, no local referente a sub-árvore esquerda do elemento de valor 3, será chamada a função inserir\_nos. O comando a seguir realiza esta função e forma a árvore apresentada na figura 108:

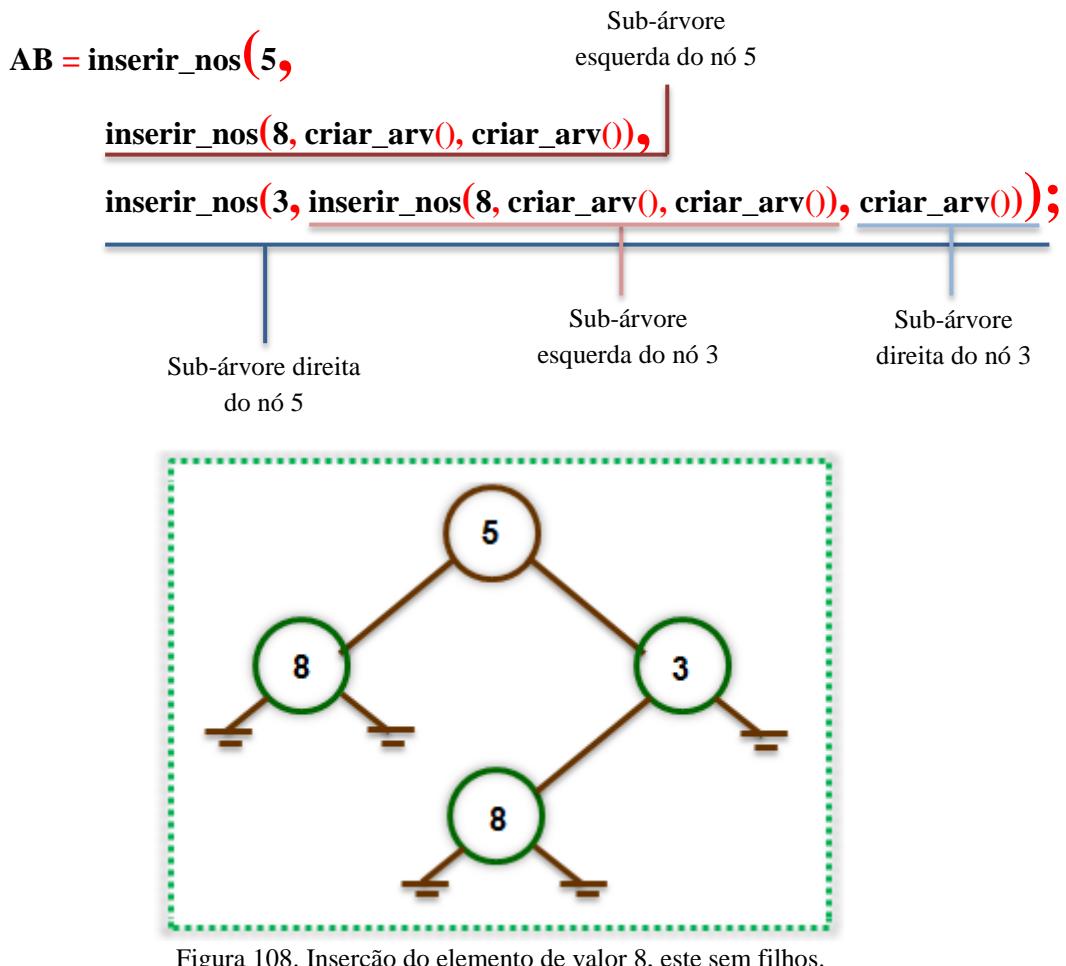
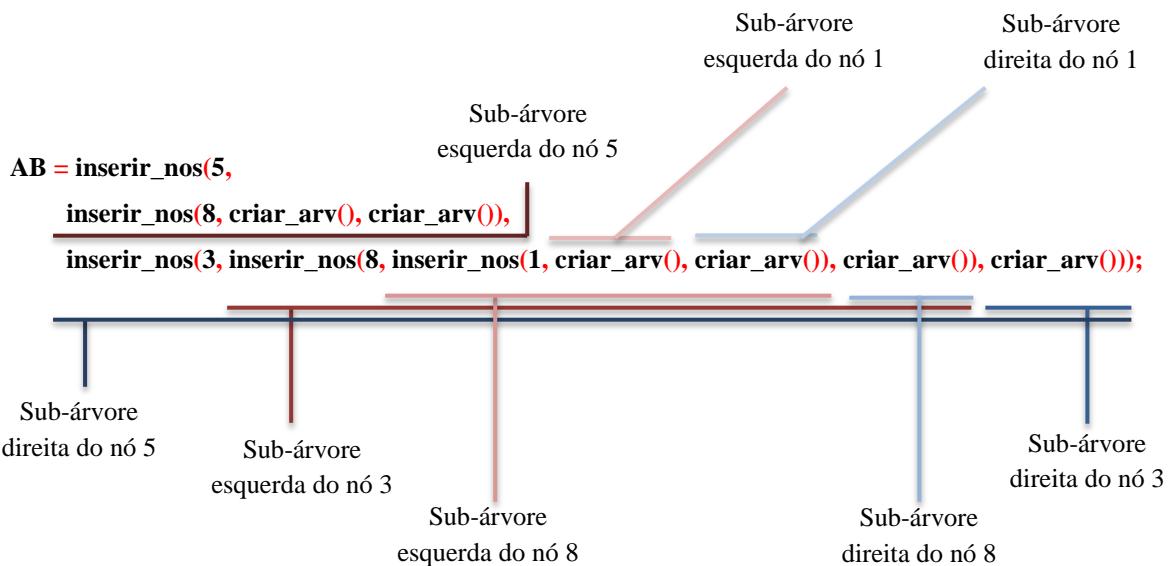


Figura 108. Inserção do elemento de valor 8, este sem filhos.

- Para a inserção do valor 1, no lado esquerdo do segundo nó de valor 8, sem filhos, haverá uma alteração do comando anterior em vez de chamar a função `criar_arv`, no local referente a sub-árvore esquerda do elemento de valor 8, será chamada a função `inserir_nos`. O comando a seguir realiza esta função e forma a árvore apresentada na figura 109:



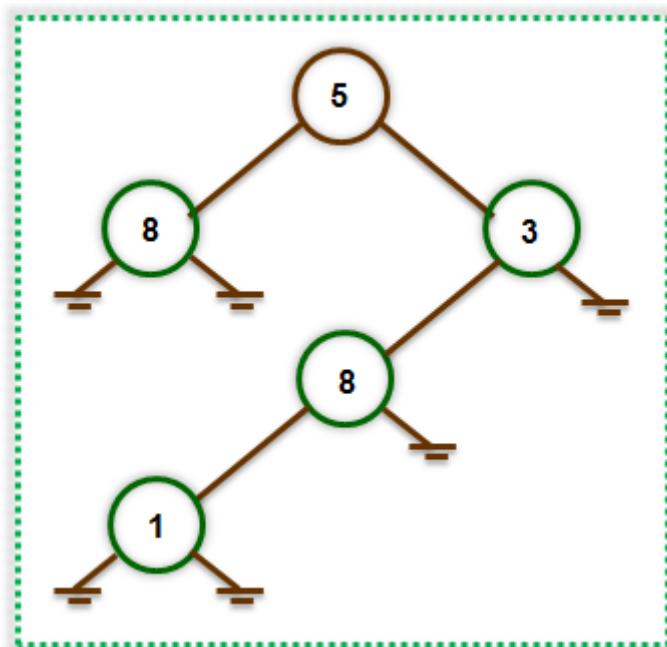


Figura 109. Inserção do elemento de valor 1, este sem filhos.

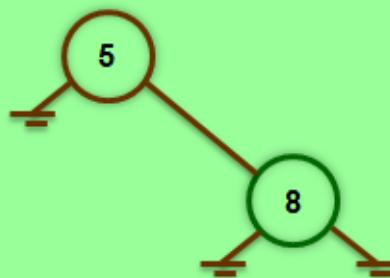
- Na função principal o seguinte comando realizará a função de formar a árvores:

```

main(void){
    Arv_B* AB;
    AB = criar_arv();
    AB = inserir_nos(5,
                    inserir_nos(8, criar_arv(), criar_arv()),
                    inserir_nos(3, inserir_nos(8,
                                                inserir_nos(1, criar_arv(), criar_arv()), criar_arv(), criar_arv())));
}
  
```

### Exemplo 2

Inserir os elementos 5 e 8 de forma que a seguinte árvore seja formada:



**1º PASSO:** Comandos a serem colocados na função principal (main()).

```

main(void){
    Arv_B* AB;
    AB = criar_arv();
    AB = inserir_nos(5, criar_arv()), inserir_nos(8, criar_arv(), criar_arv());
}

```

**2º PASSO:** Executar os comandos da função principal. Quando inicializada a função principal o ponteiro AB, tipo Arv\_B, é declarado.

**3º PASSO:** O ponteiro AB recebe o retorno da função criar\_arv, o qual será NULL para os ponteiros AB.



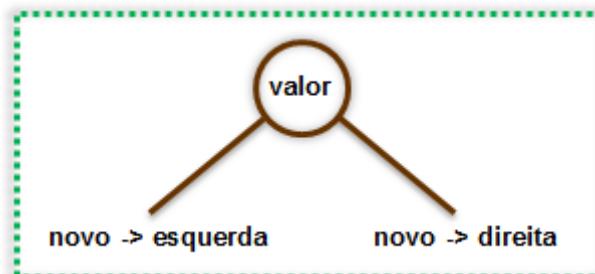
**4º PASSO:** Chama-se a função inserir\_nos dentro de outra função com a mesma funcionalidade, assim, o retorno da mais interna será usada como parâmetro para a externa.

```

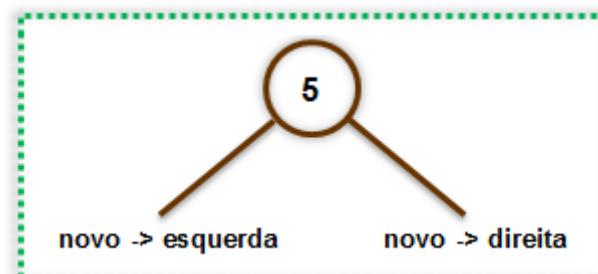
Arv_B* inserir_nos(int valor, Arv_B* sae, Arv_B* sad);
AB = inserir_nos(5, criar_arv()), inserir_nos(8, criar_arv(), criar_arv());

```

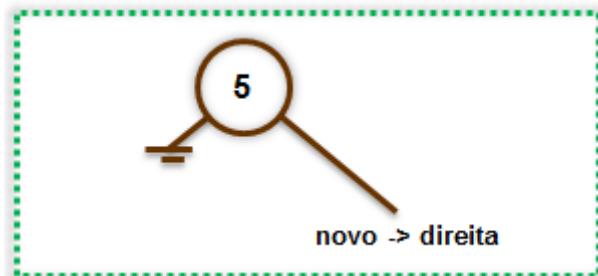
- Aloca-se um novo elemento.



- Este nó (*novo->info*) recebe o valor 5.

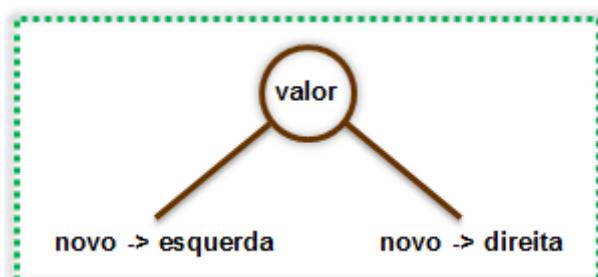


- O comando `novo -> esquerda` apontará para onde o ponteiro sae aponta. O ponteiro sae recebe o retorno da função `criar_arv()`, o qual é NULL. Assim, `novo -> esquerda` aponta para NULL.

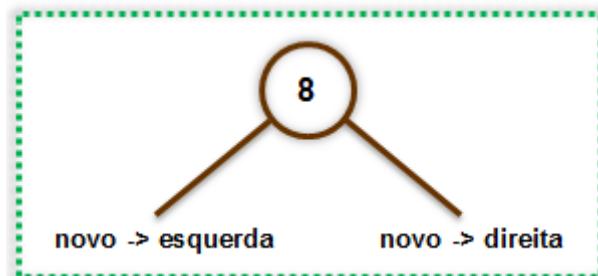


**5º PASSO:** O comando `novo -> direita` apontará para onde o ponteiro sad aponta. O ponteiro sad recebe o retorno da função `inserir_nos(8, criar_arv(), criar_arv())`. A função `inserir_nos` possui o comando `return novo`, contudo, este não será executado pois o `novo -> direita` faz com que a função inserir seja chamada novamente.

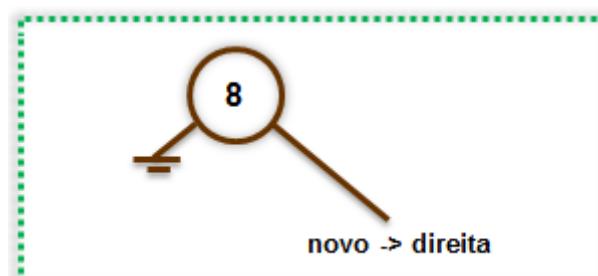
- Aloca-se um novo elemento.



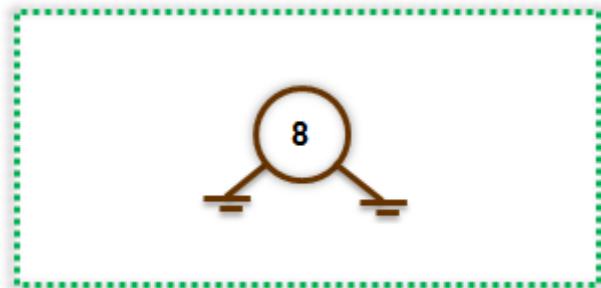
- Este nó (`novo->info`) recebe o valor 8.



- O comando `novo -> esquerda` apontará para onde o ponteiro sae aponta. O ponteiro sae recebe o retorno da função `criar_arv()`, o qual é NULL. Assim, `novo -> esquerda` aponta para NULL.



- O comando `novo -> direita` apontará para onde o ponteiro `sae` aponta. O ponteiro `sae` recebe o retorno da função `criar_arv()`, o qual é `NULL`. Assim, `novo -> direita` aponta para `NULL`.

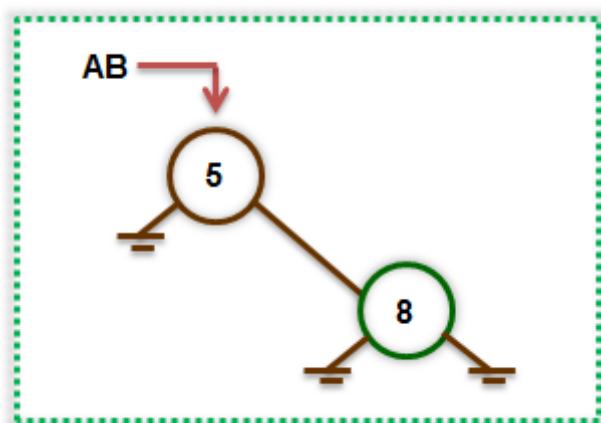


- O nó de valor 8, sem filhos, formado será o retorno da função `inserir_nos`, a qual se encontra dentro de outra função `inserir_nos`. Relembrando o comando:

**AB = inserir\_nos(5, criar\_arv()), inserir\_nos(8, criar\_arv(), criar\_arv());**

↑  
1ª chamada da função  
inserir\_nos

Assim, a **primeira** chamada da função `inserir_nos` retorna o que foi inserido. A árvore formada a partir do comando anterior será:



#### 4.1.2.4. Averiguar se a árvore binária está vazia

**O que fazer? Dicas...**

Verificar se a árvore aponta para o valor nulo (`NULL`).

A função denominada **arv\_vazia**, apresentada na figura 110, possui as seguintes características:

- A função possui como parâmetro o ponteiro AB, o qual aponta para a raiz da árvore.
- O retorno é do tipo **booleano**, retornando valor 1 (verdadeiro) quando a árvore estiver vazia, caso contrário, retorna valor 0 (falso);

```
int arv_vazia(Arv_B* AB){
    return AB == NULL;
}
```

Figura 110. Função que verifica se a árvore está vazia.

#### 4.1.2.5. Imprimir os elementos

##### O que fazer? Dicas...

- Escolher qual tipo de percurso será utilizado para percorrer a árvore.

A função denominada **arv\_imprimir**, apresentada na figura 111, possui as seguintes características:

- A função possui como parâmetro o ponteiro AB, o qual aponta para a raiz árvore.
- O retorno é do tipo void, retornando na tela os valores contidos na árvore.

```
void arv_imprimir_pre_ordem(Arv_B* AB){
    1 if(!arv_vazia(AB)){
        printf("%d ", AB -> info); /* mostra raiz*/
        arv_imprimir(AB -> esquerda); /* mostra sae*/
        arv_imprimir(AB -> direita); /* mostra sad*/
    }
}
```

Figura 111. Função imprimir em pré-ordem.

*Descrição das funcionalidades* apresentadas na figura 111:

1. Dentro da condicional chama-se a função arv\_vazia (a árvore não está vazia?), caso retorne o valor 1 a árvore não está vazia e, assim, considerando a impressão do tipo **pré-ordem**:

- Imprimi-se o valor contido no elemento.
- Chamar a função arv\_imprimir, recursivamente, passando como parâmetro a região **esquerda** da árvore.
- Chamar a função arv\_imprimir, recursivamente, passando como parâmetro a região **direita** da árvore.

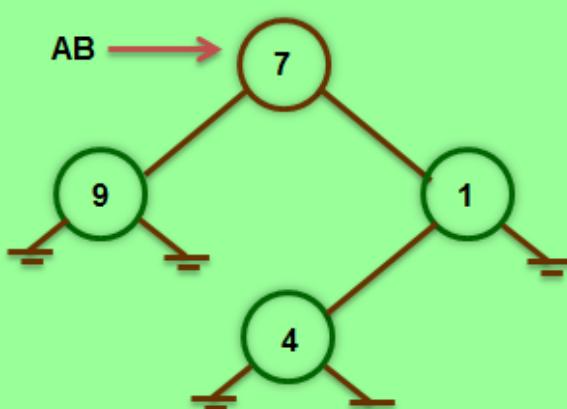
#### Observação

Na função principal (**main()**) a função para imprimir os valores contidos na árvore deve ser chamada como:

**arv\_imprimir(AB);**

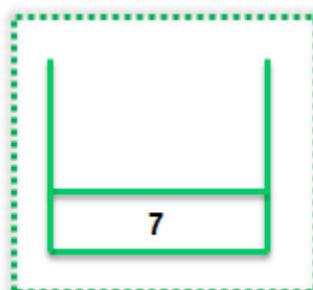
Para melhor entendimento, graficamente, realiza-se um exemplo a seguir, utilizando o conceito *teste de mesa*.

#### Exemplo



**1º PASSO:** Entra na condicional if. A função é definida por:

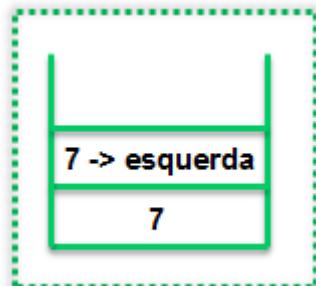
**arv\_imprimir(Arv\_B\* AB) = arv\_imprimir(<nó de valor 7>)**



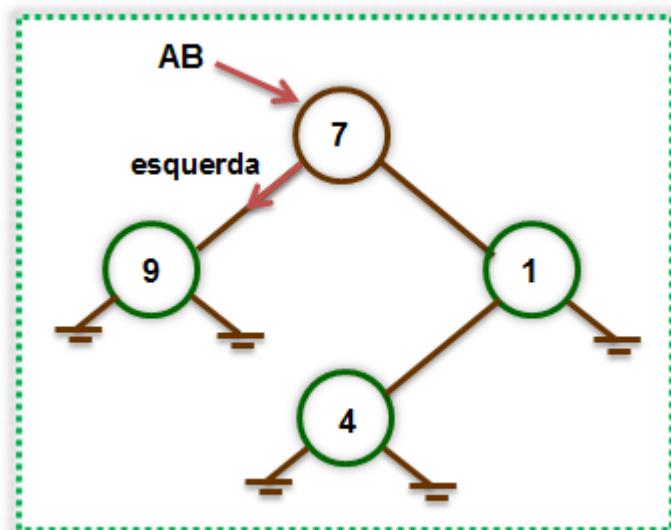
Empilha o 7.

- Chama-se a função arv\_vazia, a árvore não está vazia? SIM!! Pois o ponteiro AB, aponta para a raiz, a qual possui valor 7, ou seja, diferente de NULL.
- Imprimi-se o valor 7 contido no elemento.

**2º PASSO:** Chama a função arv\_imprimir, recursivamente, passando como parâmetro a **sub-árvore esquerda** do nó de valor 7. O ponteiro **AB** aponta para este, sendo que o ponteiro **esquerda** deste aponta para o elemento de valor 9.



Empilha a sub-árvore esquerda do nó de valor 7.



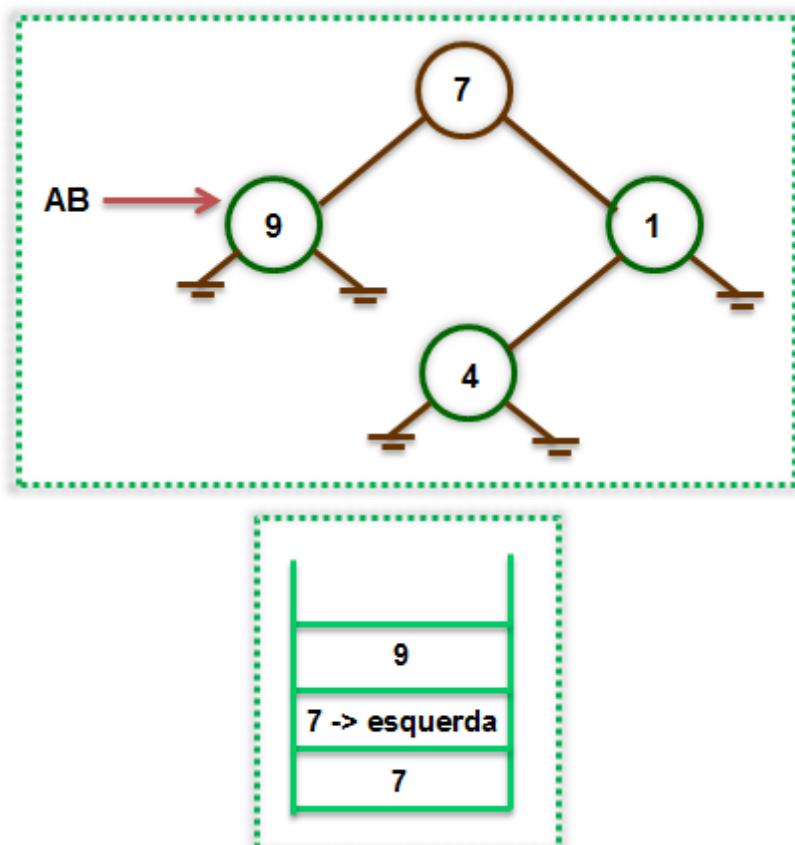
No parâmetro  $AB = AB \rightarrow \text{esquerda}$ :

$\text{arv\_imprimir}(\text{Arv\_B}^* \text{AB})$

↑  
 $\text{arv\_imprimir}(\text{AB} \rightarrow \text{esquerda})$

Assim, a função é definida por:

$\text{arv\_imprimir}(\text{Arv\_B}^* \text{AB}) = \text{arv\_imprimir}(<\text{nó de valor 9}>)$

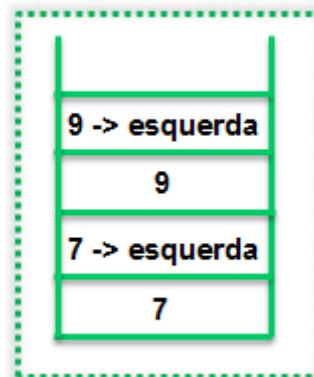


Empilha o nó de valor 9.

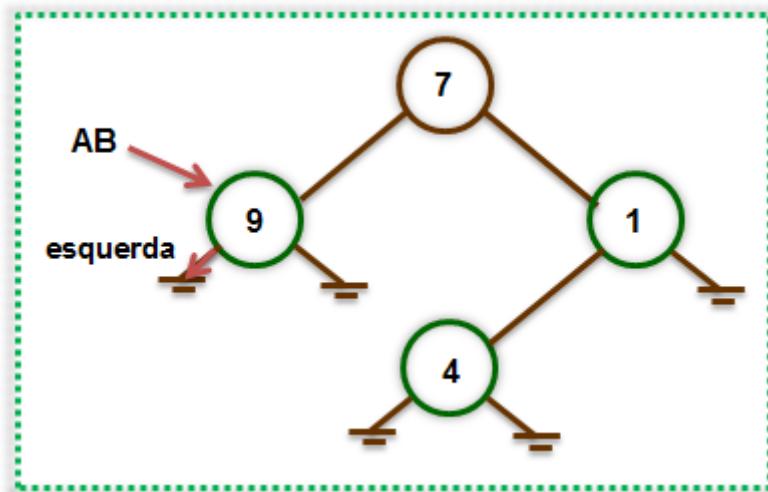
**4º PASSO:** Entra na condicional if.

- Chama-se a função `arv_vazia`, a árvore não está vazia? SIM!! Pois o ponteiro AB, aponta para a raiz, a qual possui valor 9, ou seja, diferente de NULL.
- Imprimi-se o valor 9 contido no elemento.

**5º PASSO:** Chamar a função `arv_imprimir`, recursivamente, passando como parâmetro a sub-árvore **esquerda** do nó de valor 9. O ponteiro **AB** aponta para este, sendo que o ponteiro **esquerda** deste aponta para NULL.



Empilha a sub-árvore esquerda do nó de valor 9.



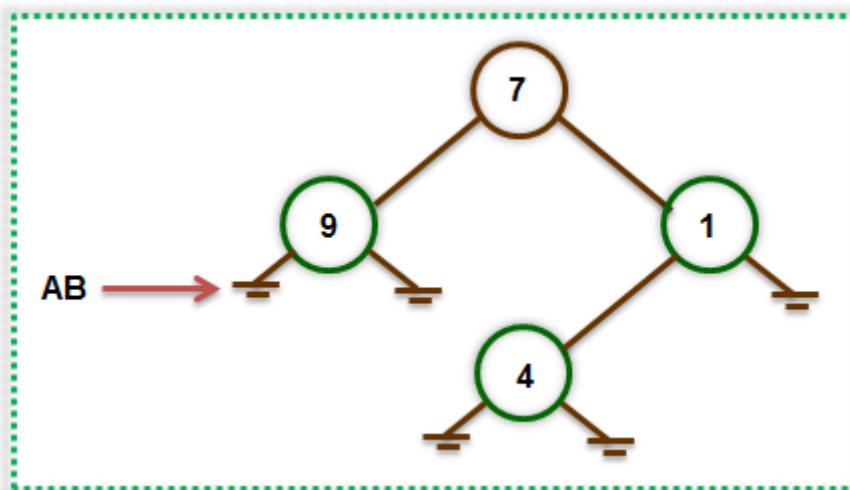
No parâmetro  $AB = AB \rightarrow esquerda$ :

```

arv_imprimir(Arv_B* AB)
    |
    +--> arv_imprimir(AB -> esquerda)
  
```

Assim, a função é definida por:

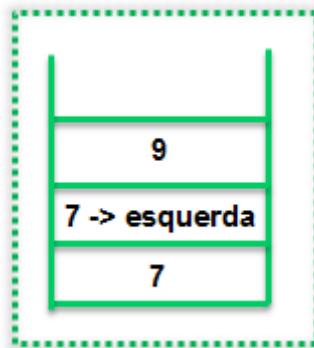
$$\text{arv\_imprimir}(\text{Arv\_B}^* \text{AB}) = \text{arv\_imprimir}(<\text{NULL}>)$$



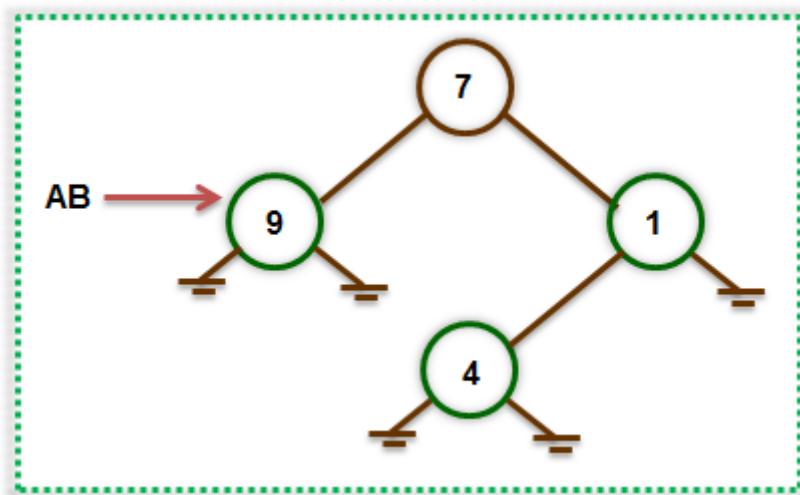
**6º PASSO:** Entra na condicional if.NULL.

- Chama-se a função arv\_vazia, a árvore não está vazia? NÃO!! Pois o ponteiro AB, aponta para NULL.

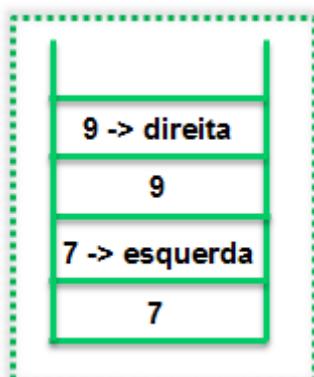
**7º PASSO:** Como encontrou o valor nulo para a função `arv_imprimir(AB -> esquerda)` haverá o retorno para a função “pai” desta sub-árvore, assim, o ponteiro AB aponta para o nó que o originou, o qual é o de valor 9.



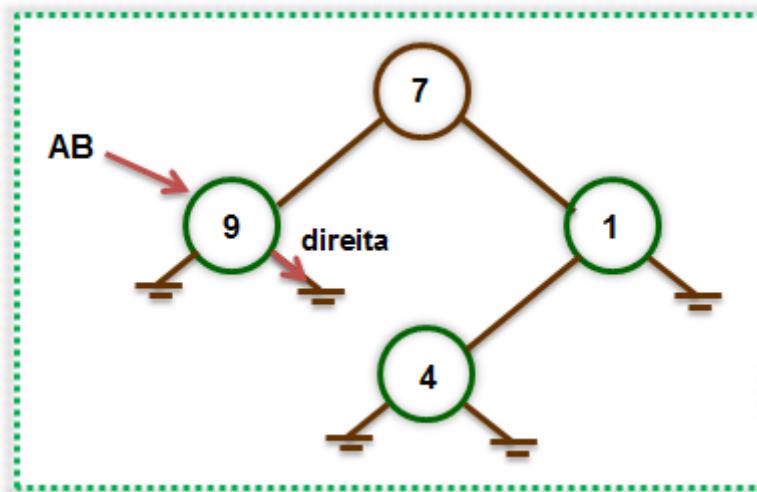
Sub-árvore esquerda do nó de valor 9 desempilhada, retorno NULL.



**8º PASSO:** A **sub-árvore esquerda** do nó de valor 9 foi analisada. O ponteiro AB aponta para este, assim, chama-se a função `arv_imprimir`, recursivamente, passando como parâmetro a **sub-árvore direita** (`arv_imprimir(AB -> direita)`) do nó de valor 9. O ponteiro **AB** aponta para este, sendo que o ponteiro **direita** aponta para NULL.



Empilha a sub-árvore direita do nó de valor 9.



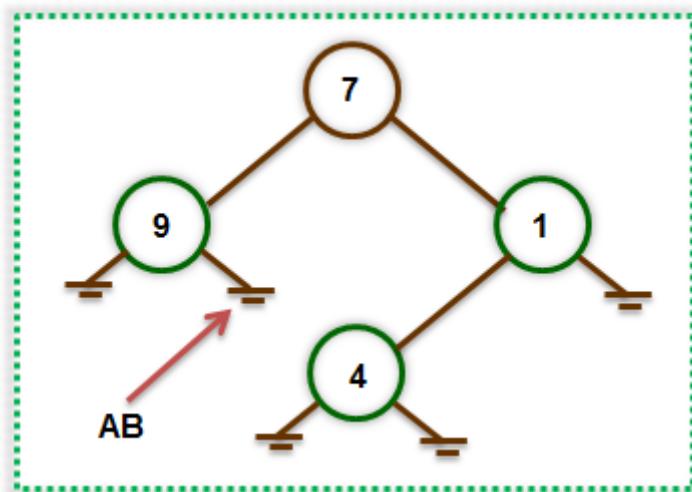
No parâmetro  $AB = AB \rightarrow direita$ :

`arv_imprimir(Arv_B* AB)`

↑  
`arv_imprimir(AB -> direita)`

Assim, a função é definida por:

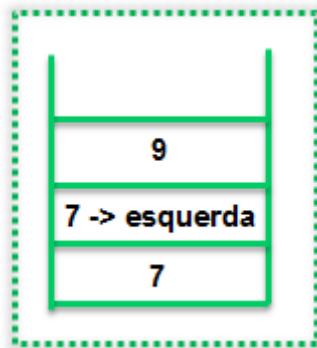
`arv_imprimir(Arv_B* AB) = arv_imprimir(<NULL>)`



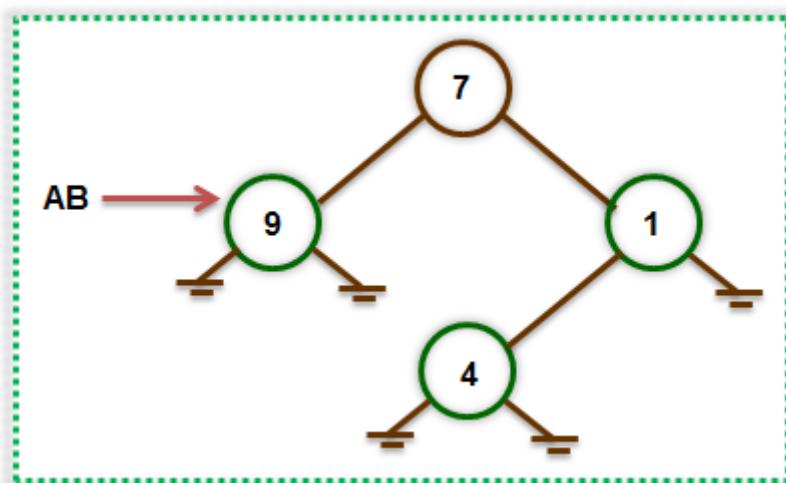
**9º PASSO:** Entra na condicional if.

- Chama-se a função `arv_vazia`, a árvore não está vazia? NÃO!! Pois o ponteiro AB, aponta para NULL.

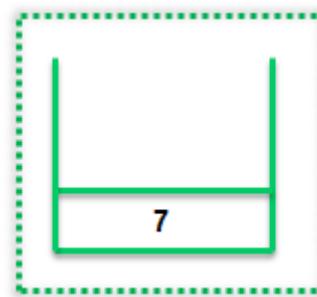
**10º PASSO:** Como encontrou o valor nulo para a função `arv_imprimir (AB -> direita)` haverá o retorno para a função “pai” desta sub-árvore, assim, o ponteiro AB aponta para o nó que o originou, o qual é o de valor 9.



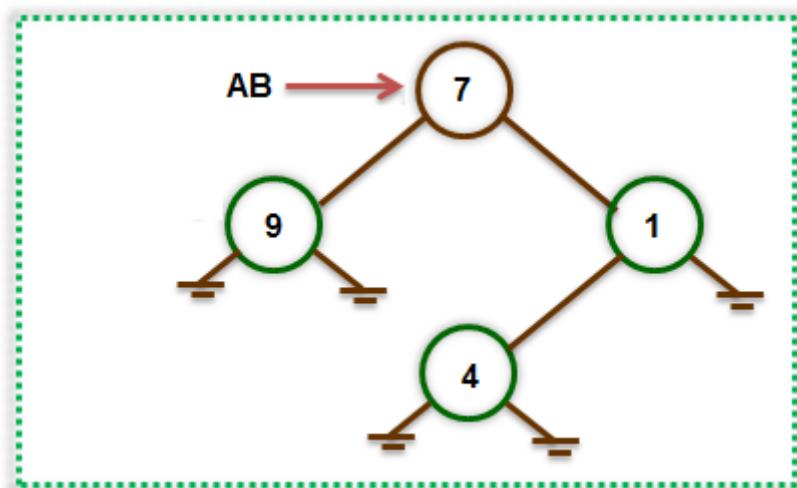
Sub-árvore direita do nó de valor 9 desempilhada, retorno NULL.



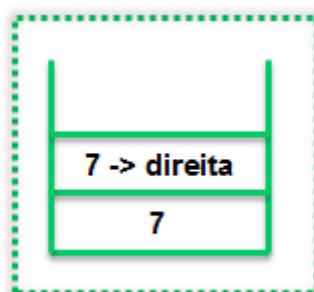
**11º PASSO:** As **sub-árvores esquerda e direita** do nó de valor **9** foram analisadas, assim, haverá o retorno para a função “pai” do elemento de valor 9, ou seja, a sub-árvore esquerda do nó de valor 7 foi analisada, assim, o ponteiro **AB** aponta para o elemento de valor **7**.



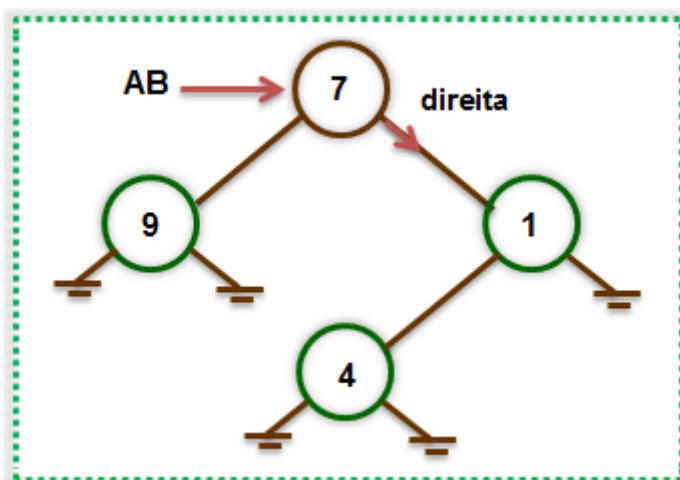
Desempilha o nó de valor 9 e a sub-árvore esquerda do elemento de valor 7.



**12º PASSO:** A **sub-árvore esquerda** do nó de valor 7 foi analisada. O ponteiro AB aponta para este, assim, chama-se a função `arv_imprimir`, recursivamente, passando como parâmetro a **sub-árvore direita** (`arv_imprimir(AB -> direita)`) do nó de valor 7. O ponteiro **AB** aponta para este, sendo que o ponteiro **direita** aponta para o nó de valor 1.



Empilha a sub-árvore direita do nó de valor 7.



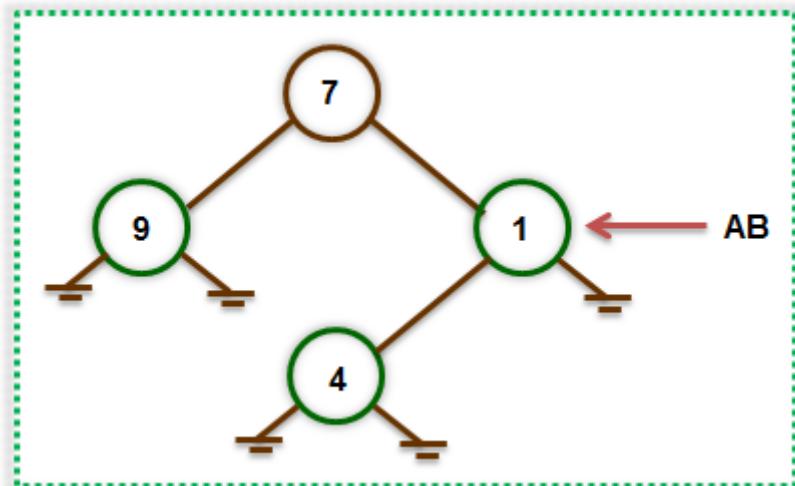
No parâmetro  $AB = AB \rightarrow direita$ :

$\text{arv\_imprimir}(\text{Arv\_B}^* \text{AB})$

$\uparrow$   
 $\text{arv\_imprimir}(\text{AB} \rightarrow \text{direita})$

Assim, a função é definida por:

$$\text{arv\_imprimir}(\text{Arv\_B}^* \text{AB}) = \text{arv\_imprimir}(<\text{nó de valor } 1>)$$

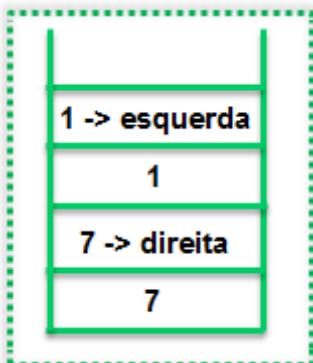


Empilha o nó de valor 1.

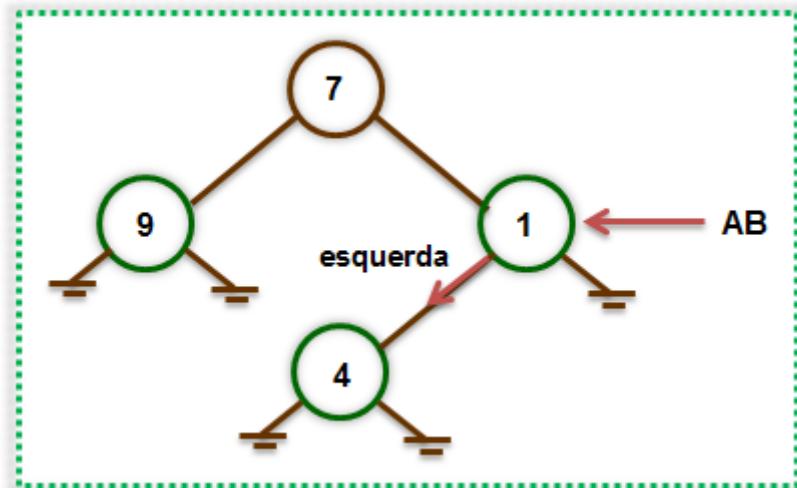
**13º PASSO:** Entra na condicional if.

- Chama-se a função `arv_vazia`, a árvore não está vazia? SIM!! Pois o ponteiro **AB**, aponta para o nó de valor 1.
- Imprimi-se o valor 1 contido no elemento.

**14º PASSO:** Chamar a função `arv_imprimir`, recursivamente, passando como parâmetro a **sub-árvore esquerda** do nó de valor 1. O ponteiro **AB** aponta para este, sendo que o ponteiro **esquerda** deste aponta para o nó de valor 4.



Empilha a sub-árvore esquerda do nó de valor 1.



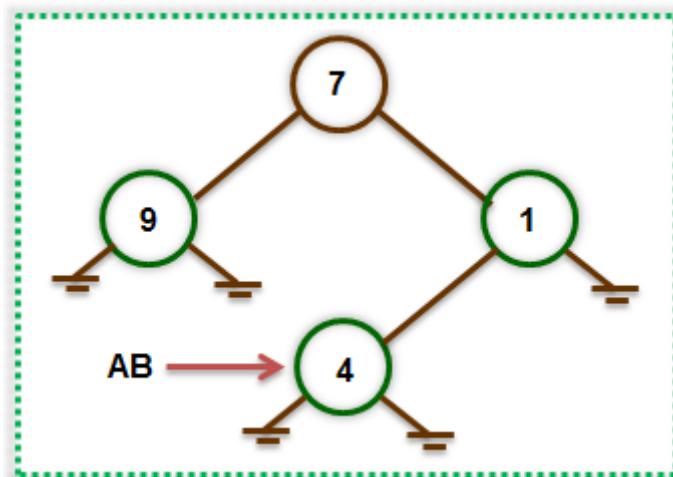
No parâmetro  $AB = AB \rightarrow \text{esquerda}$ :

`arv_imprimir(Arv_B* AB)`

$\uparrow$   
`arv_imprimir(AB -> esquerda)`

Assim, a função é definida por:

`arv_imprimir(Arv_B* AB) = arv_imprimir(<nó de valor 4>)`

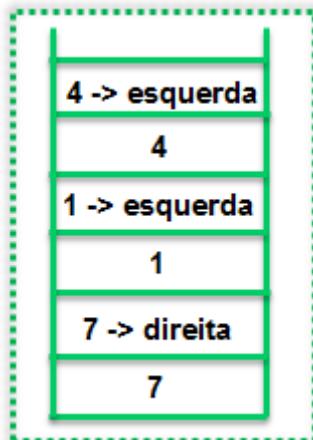


Empilha o nó de valor 4.

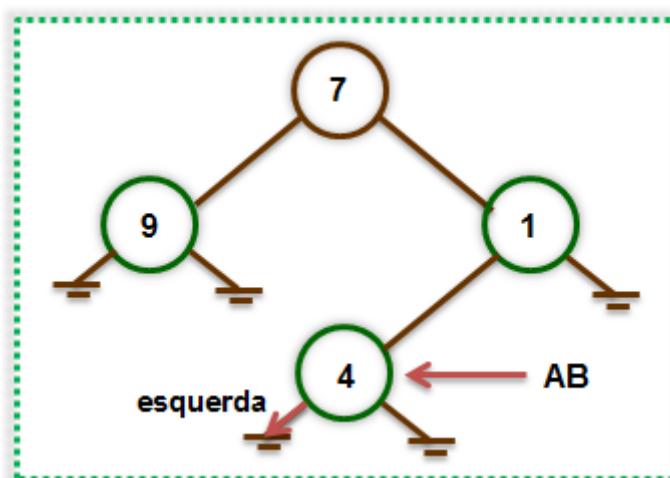
**15º PASSO:** Entra na condicional if.

- Chama-se a função `arv_vazia`, a árvore não está vazia? SIM!! Pois o ponteiro `AB`, aponta para o nó de valor 4.
- Imprimi-se o valor 4 contido no elemento.

**16º PASSO:** Chamar a função `arv_imprimir`, recursivamente, passando como parâmetro a **sub-árvore esquerda** do nó de valor 4. O ponteiro `AB` aponta para este, sendo que o ponteiro **esquerda** deste aponta para `NULL`.



Empilha a sub-árvore esquerda do nó de valor 4.



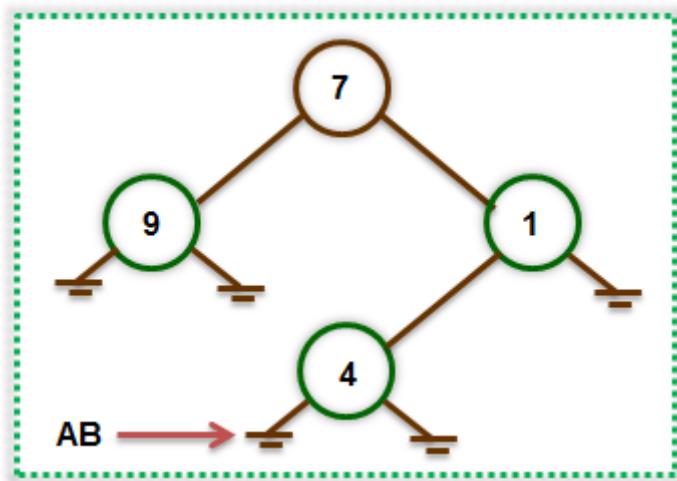
No parâmetro `AB = AB -> esquerda`:

`arv_imprimir(Arv_B* AB)`

`arv_imprimir(AB -> esquerda)`

Assim, a função é definida por:

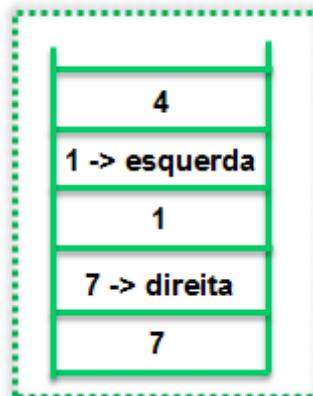
`arv_imprimir(Arv_B* AB) = arv_imprimir(<NULL>)`



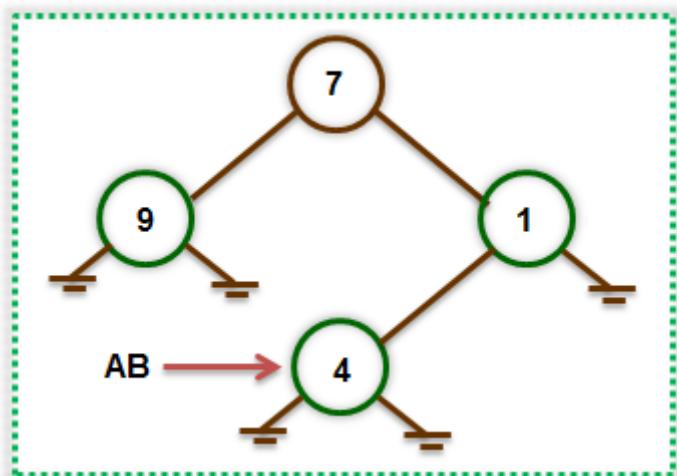
**17º PASSO:** Entra na condicional if.

- Chama-se a função `arv_vazia`, a árvore não está vazia? SIM!! Pois o ponteiro AB, aponta para NULL.

**18º PASSO:** Como encontrou o valor nulo para a função `arv_imprimir(AB -> esquerda)` haverá o retorno para a função “pai” desta sub-árvore, assim, o ponteiro AB aponta para o nó que o originou, o qual é o de valor 4.



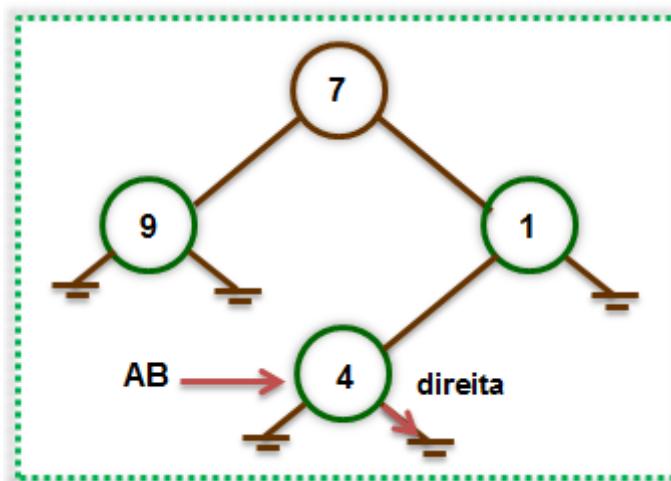
Sub-árvore esquerda do nó de valor 4 desempilhada, retorno NULL.



**19º PASSO:** A **sub-árvore esquerda** do nó de valor 4 foi analisada. O ponteiro AB aponta para este, assim, chama-se a função `arv_imprimir`, recursivamente, passando como parâmetro a **sub-árvore direita** (`arv_imprimir(AB -> direita)`) do nó de valor 4. O ponteiro **AB** aponta para este, sendo que o ponteiro **direita** aponta para NULL.



Empilha a sub-árvore direita do nó de valor 4.



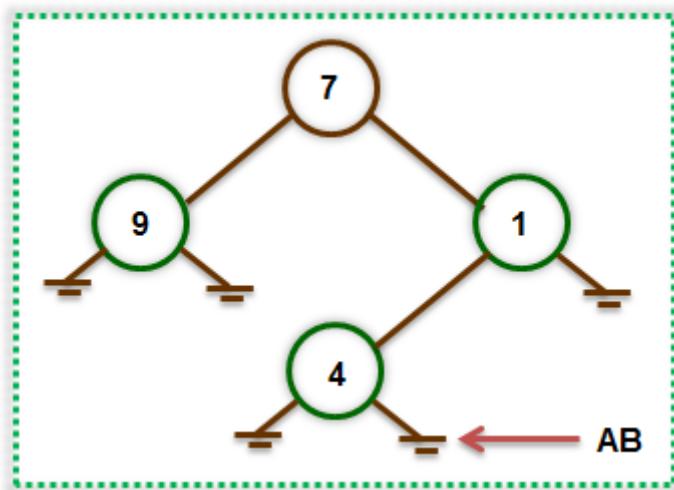
No parâmetro  $AB = AB \rightarrow direita$ :

$\text{arv\_imprimir}(\text{Arv\_B}^* \text{AB})$

$\overbrace{\hspace{10em}}$   
 $\text{arv\_imprimir}(\text{AB} \rightarrow \text{direita})$

Assim, a função é definida por:

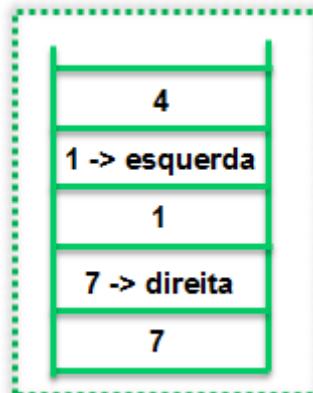
$\text{arv\_imprimir}(\text{Arv\_B}^* \text{AB}) = \text{arv\_imprimir}(<\text{NULL}>)$



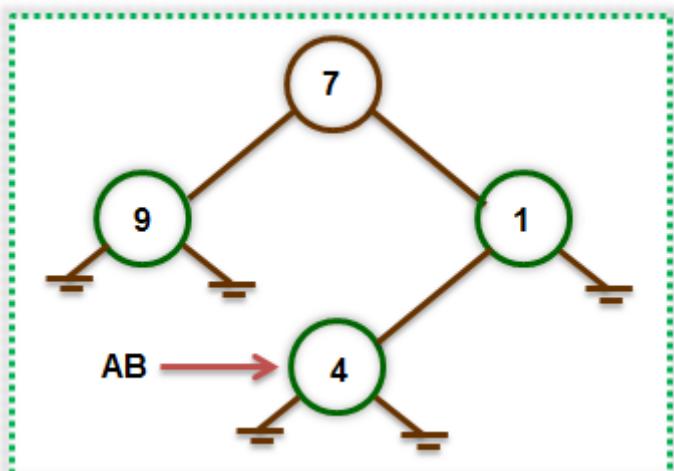
**20º PASSO:** Entra na condicional if.

- Chama-se a função `arv_vazia`, a árvore não está vazia? NÃO!! Pois o ponteiro AB, aponta para NULL.

**21º PASSO:** Como encontrou o valor nulo para a função `arv_imprimir(AB -> direita)` haverá o retorno para a função “pai” desta sub-árvore, assim, o ponteiro AB aponta para o nó que o originou, o qual é o de valor 4.



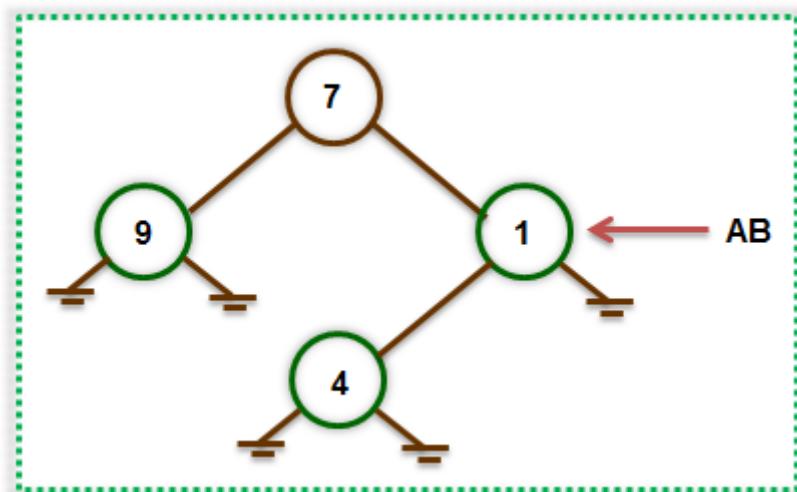
Sub-árvore direita do nó de valor 4 desempilhada, retorno NULL.



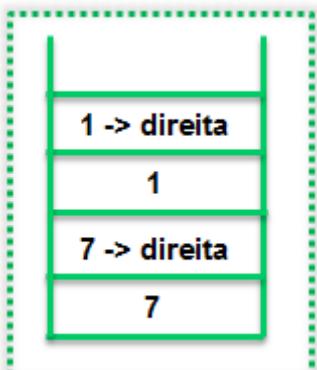
**22º PASSO:** As **sub-árvores esquerda e direita** do nó de valor 4 foram analisadas, assim, haverá o retorno para a função “pai” do elemento de valor 4, ou seja, a sub-árvore esquerda do nó de valor 1 foi analisada, assim, o ponteiro **AB** aponta para o **1**.



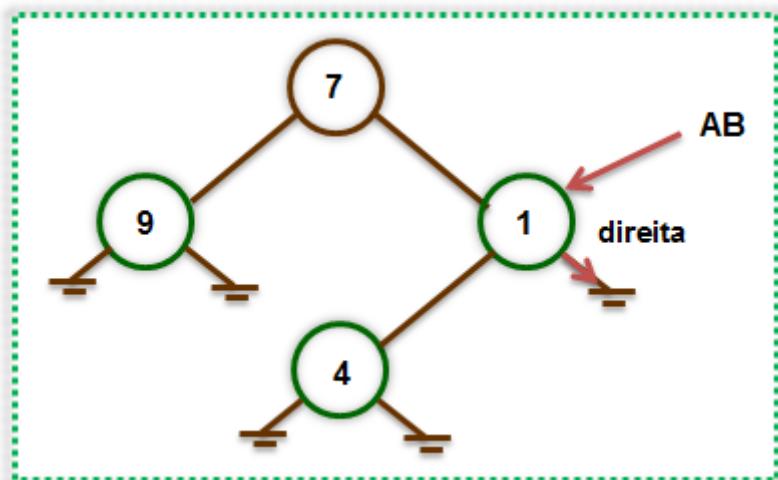
Desempilha o nó de valor 4 e a sub-árvore esquerda do elemento de valor 1.



**23º PASSO:** A **sub-árvore esquerda** do nó de valor 1 foi analisada. O ponteiro AB aponta para este, assim, chama-se a função `arv_imprimir`, recursivamente, passando como parâmetro a **sub-árvore direita** (`arv_imprimir(AB -> direita)`) do nó de valor **1**. O ponteiro **AB** aponta para este, sendo que o ponteiro **direita** aponta para **NULL**.



Empilha a sub-árvore direita do nó de valor 1.



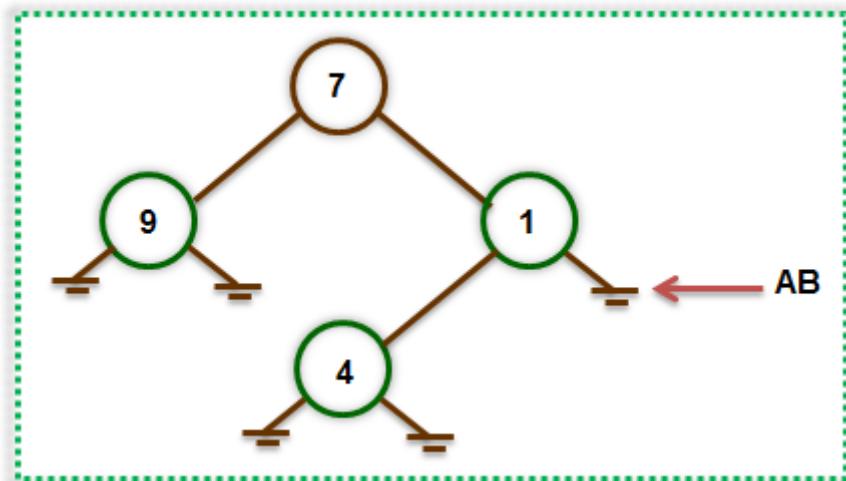
No parâmetro  $AB = AB \rightarrow direita$ :

arv\_imprimir(**Arv\_B\*** AB)

arv\_imprimir(AB -> direita)

Assim, a função é definida por:

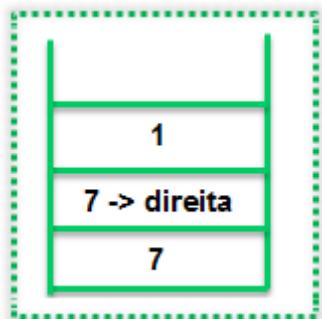
arv\_imprimir(**Arv\_B\*** AB) = arv\_imprimir(<NULL>)



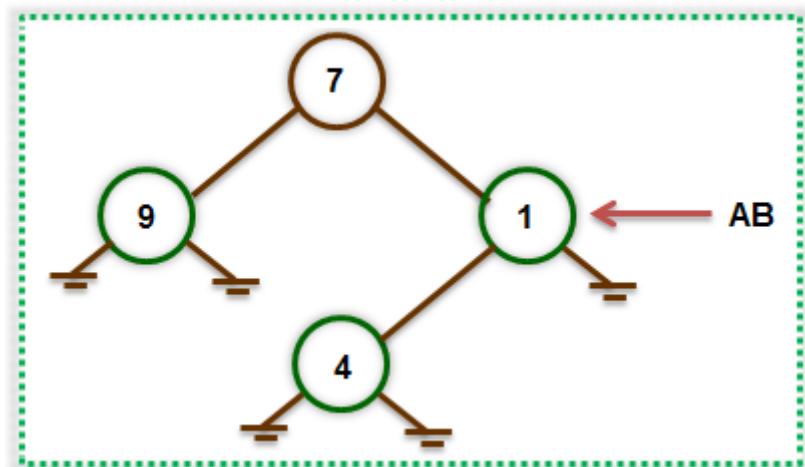
**24º PASSO:** Entra na condicional if.

- Chama-se a função `arv_vazia`, a árvore não está vazia? NÃO!! Pois o ponteiro `AB`, aponta para `NULL`.

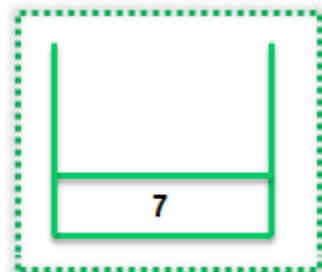
**25º PASSO:** Como encontrou o valor nulo para a função `arv_imprimir(AB -> direita)` haverá o retorno para a função “pai” desta sub-árvore, assim, o ponteiro `AB` aponta para o nó que o originou, o qual é o de valor 1.



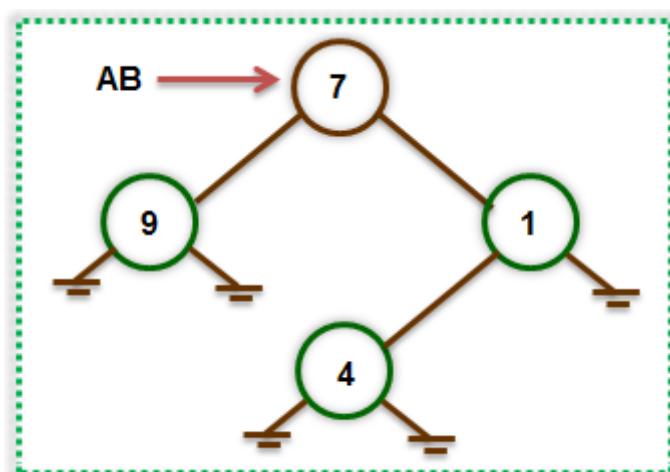
Sub-árvore direita do nó de valor 1 desempilhada, retorno NULL.



**26º PASSO:** As sub-árvores esquerda e direita do nó de valor 1 foram analisadas, assim, haverá o retorno para a função “pai” do elemento de valor 1, ou seja, a sub-árvore direita do nó de valor 7 foi analisada, assim, o ponteiro AB aponta para o 7.



Desempilha o nó de valor 1 e a sub-árvore direita do elemento de valor 7.



**27º PASSO:** As sub-árvore esquerda e direita do nó de valor 7 foram analisadas, assim, terminou-se a impressão!

#### 4.1.2.6. Verificar se um determinado elemento pertence a árvore

##### O que fazer? Dicas...

- Percorrer a árvore comparando o valor que o nó possui com o valor a ser buscado.
- Caso encontre, retorna 1, ou seja, indica a ocorrência deste valor na árvore.

A função denominada **arv\_pertence**, apresentada na figura 112, possui as seguintes características:

- A função possui como parâmetro o valor a ser procurado e o ponteiro AB, o qual possui acesso a raiz.
- O retorno é do tipo **booleano**, retornando valor 1 indicando a **ocorrência** do valor, caso contrário, retorna valor 0 (falso).

```
int arv_pertence(Arv_B* AB, int valor){
    1   if(arv_vazia(AB))
        return 0; /* árvore vazia: não encontrou*/
    2   else
        return AB -> info == valor ||
               arv_pertence(AB -> esquerda, valor) ||
               arv_pertence(AB -> direita, valor)
}
```

Figura 112. Função para verificar se um dado elemento pertence a árvore.

*Descrição das funcionalidades* apresentadas na figura 112:

1. Dentro da condicional chama-se a função **arv\_vazia**, caso retorne o valor 1 a árvore está vazia e, assim, retorna 0.
2. Senão:
  - Compara-se o valor contido no elemento apontado pelo ponteiro AB com o valor a ser procurado, caso o encontre retorna 1.
  - Se a árvore não está vazia chama-se recursivamente a função **arv\_pertence** tanto para a região esquerda quanto para a região direita.

**Observação**

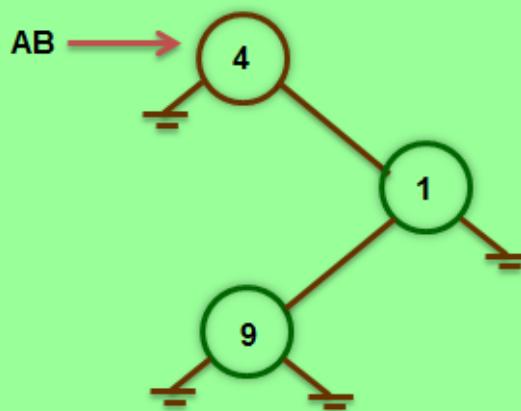
Na função principal (`main()`) a função para verificar a presença de um valor contido na árvore deve ser chamada como:

```
int resultado = arv_pertence(AB, <valor a ser procurado>);
```

Para melhor entendimento, graficamente, realiza-se um exemplo a seguir, utilizando o conceito *teste de mesa*.

**Exemplo**

Verificar se o valor 1 se encontra na árvore:



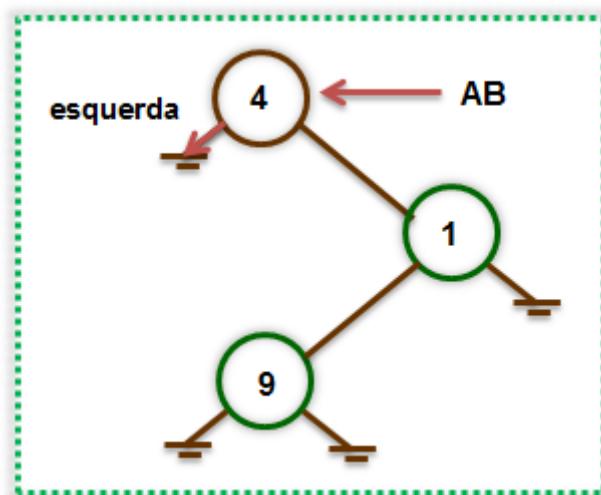
**1º PASSO:** Entra na condicional if. O parâmetro da função `arv_imprimir` é o **nó de valor 4**.

- Chama-se a função `arv_vazia`, a árvore está vazia? NÃO!! Pois o ponteiro AB aponta para a raiz, o qual possui valor 4, ou seja, diferente de NULL.

**2º PASSO:** Entra na condicional else.

- *Verifica a primeira condição:* o campo de informação do nó apontado pelo ponteiro AB (`AB -> info`) é igual ao valor a ser buscado, ou seja, 4 é igual à 1? NÃO!

**3º PASSO:** Chamar a função `arv_pertence`, recursivamente, passando como parâmetro a **sub-árvore esquerda** do nó de valor 4. O ponteiro **AB** aponta para este, sendo que o ponteiro **esquerda** deste aponta para NULL.



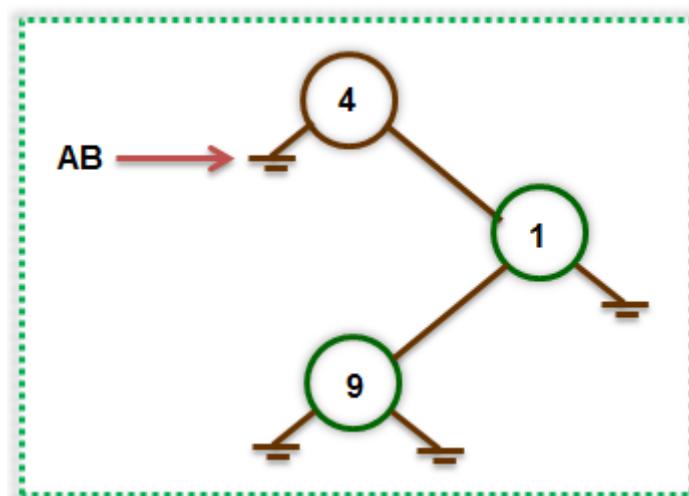
No parâmetro  $AB = AB \rightarrow \text{esquerda}$ :

$\text{arv\_pertence}(\text{Arv\_B}^* \text{AB})$

$\uparrow$   
 $\text{arv\_pertence}(\text{AB} \rightarrow \text{esquerda})$

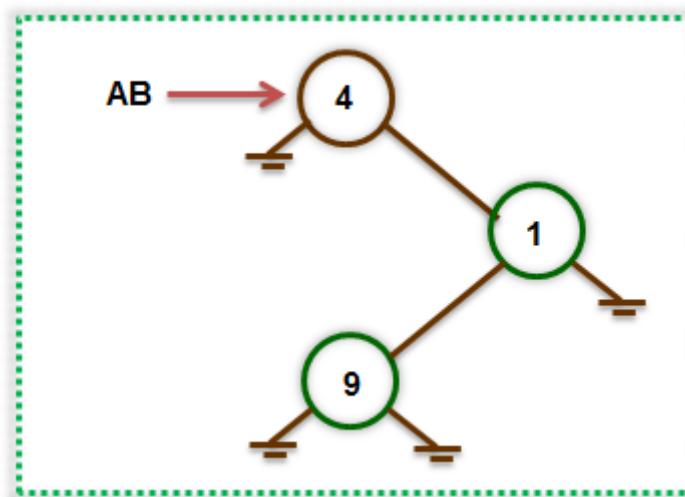
Assim, a função é definida por:

$\text{arv\_pertence}(\text{Arv\_B}^* \text{AB}, 1) = \text{arv\_pertence}(<\text{NULL}>, 1)$

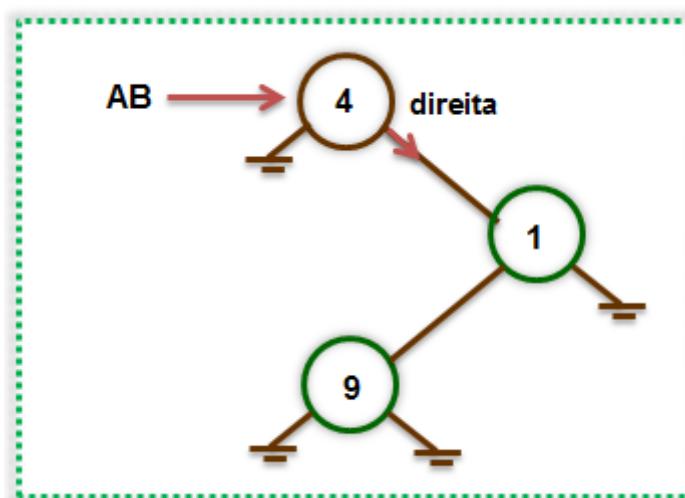


**4º PASSO:** Entra na condicional if.

- Chama-se a função  $\text{arv\_vazia}$ , a árvore está vazia? SIM!! Pois o ponteiro AB, aponta para NULL, retorna zero para a chamada desta.
- Como encontrou o valor nulo para a função  $\text{arv\_ pertence}(\text{AB} \rightarrow \text{esquerda})$  haverá o retorno para a função “pai” desta sub-árvore, assim, o ponteiro AB aponta para o nó que o originou, o qual é o de valor 4.



**5º PASSO:** A **sub-árvore esquerda** do nó de valor 4 foi analisada. O ponteiro AB aponta para este, assim, chama-se a função `arv_pertence`, recursivamente, passando como parâmetro a **sub-árvore direita** (`arv_pertence(AB -> direita, 1)`) do nó de valor 4. O ponteiro **AB** aponta para este, sendo que o ponteiro **direita** aponta para o nó de valor 1.



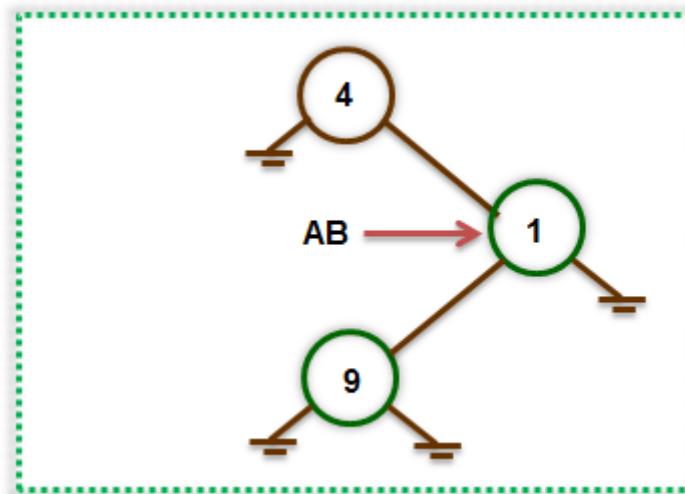
No parâmetro  $AB = AB \rightarrow direita$ :

$\text{arv\_pertence}(\text{Arv\_B}^* \text{AB}, 1)$

$\overbrace{\text{arv\_pertence}(AB \rightarrow direita, 1)}$   
 ↑

Assim, a função é definida por:

$\text{arv\_pertence}(\text{Arv\_B}^* \text{AB}, 1) = \text{arv\_pertence}(<\text{nó de valor } 1>, 1)$



**6º PASSO:** Entra na condicional if.

- Chama-se a função `arv_vazia`, a árvore está vazia? NÃO!! Pois o ponteiro AB aponta para o nó de valor 1, ou seja, diferente de NULL.

**7º PASSO:** Entra na condicional else.

- *Verifica a primeira condição:* o campo de informação do nó apontado pelo ponteiro AB (`AB -> info`) é igual ao valor a ser buscado, ou seja, 1 é igual à 1? SIM!
- Retorna 1, ou seja, o valor 1 se encontra na árvore, assim, termina-se a busca.

#### 4.1.2.6. Liberar todos os elementos

##### O que fazer? Dicas...

- Percorrer a árvore desalocando cada nó.
- Após desalocar todos os elementos retornar NULL para o ponteiro que referencia a árvore.

A função denominada `arv_liberar`, apresentada na figura 113, possui as seguintes características:

- A função possui como parâmetro o ponteiro AB.
- O retorno é do tipo criado.

```

Arv_B* arv_liberar(Arv_B* AB){

1   if(!arv_vazia(AB)){
        arv_liberar(AB -> esquerda); /* liberar sae*/
        arv_liberar(AB -> direita); /* liberar sad*/
        free(AB);                  /* liberar raiz*/
    }
2   return NULL;
}

```

Figura 113. Função liberar a árvore.

**Descrição das funcionalidades** apresentadas na figura 113:

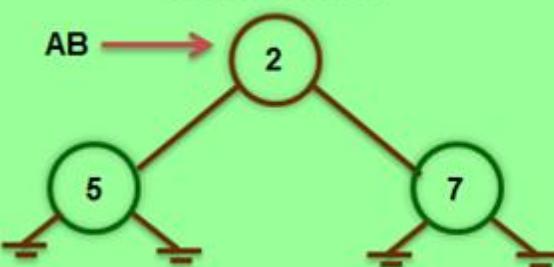
1. Dentro da condicional chama-se a função `arv_vazia`, caso retorne o valor 1 a árvore está vazia e, assim, retorna 0.
2. Senão:
  - Chamar a função `arv_liberar`, recursivamente, passando como parâmetro a região **esquerda** da árvore.
  - Chamar a função `arv_liberar`, recursivamente, passando como parâmetro a região **direita** da árvore.
  - Libera o nó apontado pelo ponteiro `AB`.
  - Após a liberação de toda a árvore o valor nulo é retornado para o ponteiro `AB`.

**Observação**

Na função principal (`main()`) a função para liberar a árvore deve ser chamada como:

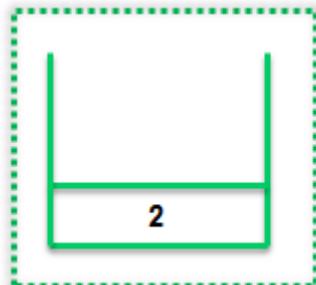
`Arv_B* AB = arv_liberar(AB);`

Para melhor entendimento, graficamente, realiza-se um exemplo a seguir, utilizando o conceito *teste de mesa*.

**Exemplo**

**1º PASSO:** Entra na condicional if. A função é definida por:

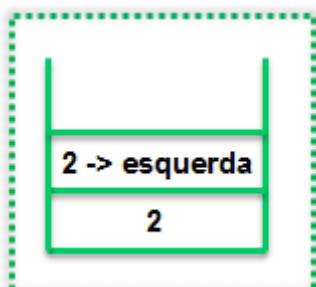
$$\text{arv\_liberar}(\text{Arv\_B}^* \text{AB}) = \text{arv\_liberar}(<\text{nó de valor } 2>)$$



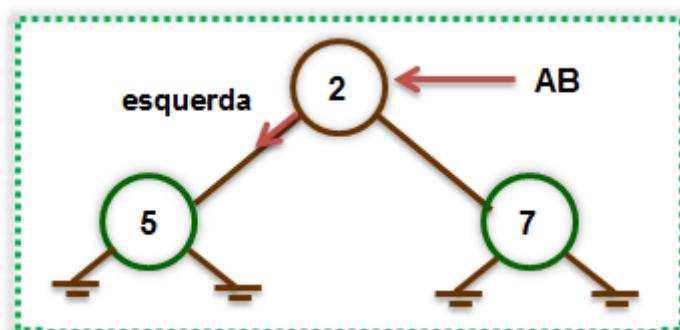
Empilha o nó de valor 2.

- Chama-se a função arv\_vazia, a árvore não está vazia? SIM!! Pois o ponteiro AB, aponta para a raiz, a qual possui valor 2, ou seja, diferente de NULL.

**2º PASSO:** Chama a função arv\_liberar, recursivamente, passando como parâmetro a **sub-árvore esquerda** do nó de valor 2. O ponteiro **AB** aponta para este, sendo que o ponteiro **esquerda** deste aponta para o elemento de valor 5.



Empilha a sub-árvore esquerda do nó de valor 2.



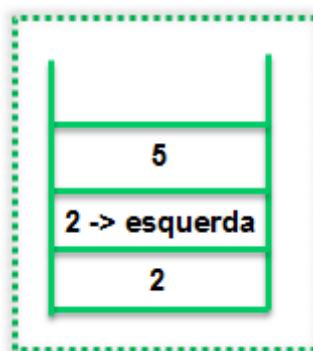
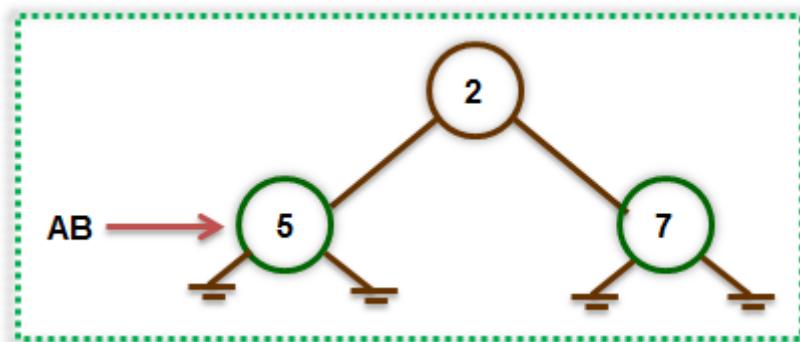
No parâmetro  $AB = AB \rightarrow \text{esquerda}$ :

$$\text{arv\_liberar}(\text{Arv\_B}^* \text{AB})$$

$$\text{arv\_liberar}(\text{AB} \rightarrow \text{esquerda})$$

Assim, a função é definida por:

$$\text{arv\_liberar}(\text{Arv\_B}^* \text{AB}) = \text{arv\_liberar}(<\text{nó de valor } 5>)$$

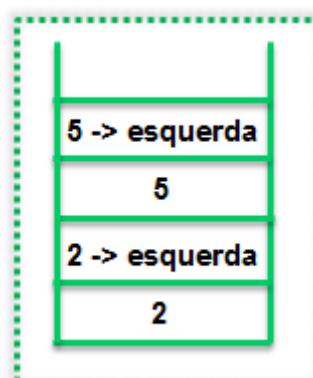


Empilha o nó de valor 5.

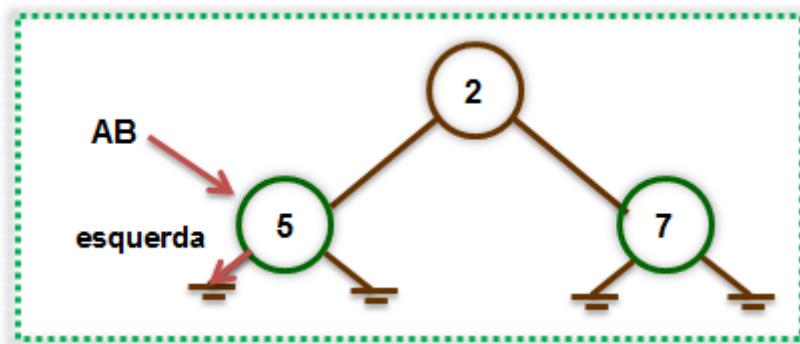
**3º PASSO:** Entra na condicional if.

- Chama-se a função `arv_vazia`, a árvore não está vazia? SIM!! Pois o ponteiro **AB**, aponta para a raiz, a qual possui valor 5, ou seja, diferente de `NULL`.

**4º PASSO:** Chama a função `arv_liberar`, recursivamente, passando como parâmetro a **sub-árvore esquerda** do nó de valor 5. O ponteiro **AB** aponta para este, sendo que o ponteiro **esquerda** deste aponta para `NULL`.



Empilha a sub-árvore esquerda do nó de valor 5.



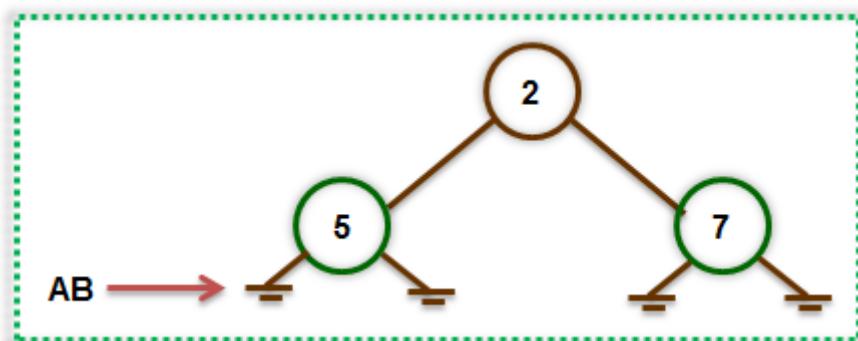
No parâmetro  $AB = AB \rightarrow \text{esquerda}$ :

$\text{arv\_liberar}(\text{Arv\_B}^* \text{AB})$

$\uparrow$   
 $\text{arv\_liberar}(\text{AB} \rightarrow \text{esquerda})$

Assim, a função é definida por:

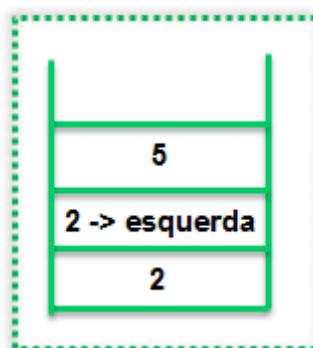
$$\text{arv\_liberar}(\text{Arv\_B}^* \text{AB}) = \text{arv\_liberar}(<\text{NULL}>)$$



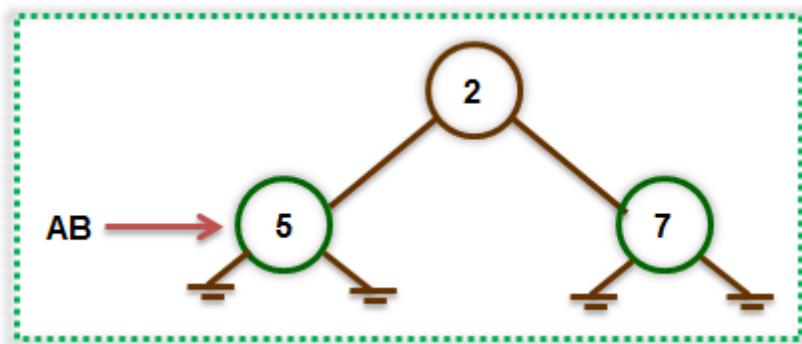
**4º PASSO:** Entra na condicional if.

- Chama-se a função `arv_vazia`, a árvore não está vazia? NÃO!! Pois o ponteiro `AB`, aponta para `NULL`.

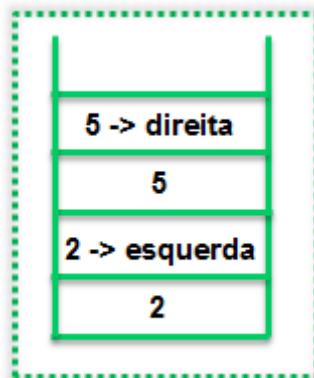
**5º PASSO:** Retorna `NULL` para a função `arv_liberar(AB->esquerda)`, haverá o retorno para a função “pai” desta sub-árvore, assim, o ponteiro `AB` aponta para o nó que o originou, o qual é o de valor 5.



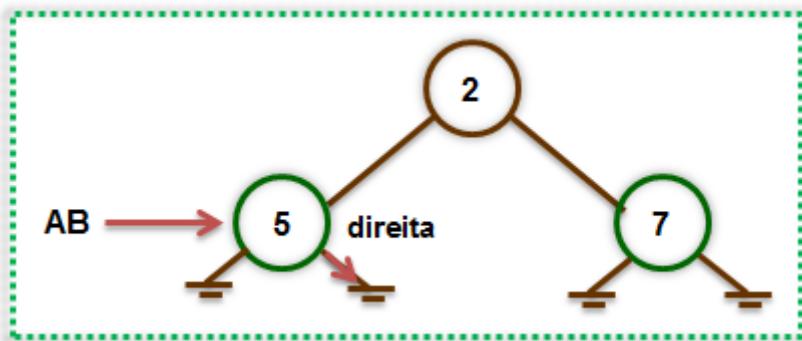
Sub-árvore esquerda do nó de valor 5 desempilhada, retorno `NULL`.



**6º PASSO:** A **sub-árvore esquerda** do nó de valor 5 foi analisada. O ponteiro AB aponta para este, assim, chama-se a função `arv_liberar`, recursivamente, passando como parâmetro a **sub-árvore direita** (`arv_liberar(AB -> direita)`) do nó de valor 5. O ponteiro **AB** aponta para este, sendo que o ponteiro **direita** aponta para NULL.



Empilha a sub-árvore direita do nó de valor 5.



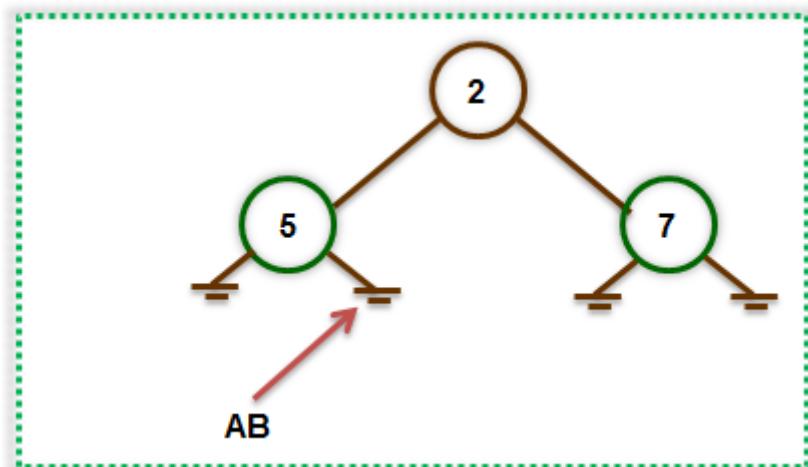
No parâmetro  $AB = AB \rightarrow direita$ :

$\text{arv\_liberar}(\text{Arv\_B}^* \text{AB})$

↑  
 $\text{arv\_liberar}(AB \rightarrow direita)$

Assim, a função é definida por:

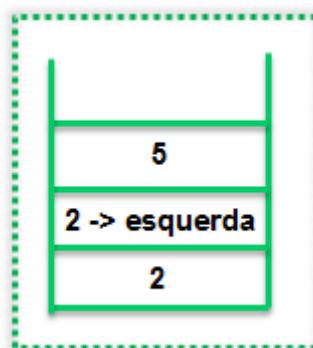
$\text{arv\_liberar}(\text{Arv\_B}^* \text{AB}) = \text{arv\_liberar}(<\text{NULL}>)$



**7º PASSO:** Entra na condicional if.

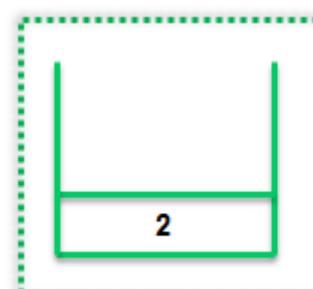
- Chama-se a função `arv_vazia`, a árvore não está vazia? NÃO!! Pois o ponteiro AB, aponta para NULL.

**8º PASSO:** Retorna NULL para a função `arv_liberar(AB -> direita)`, haverá o retorno para a função “pai” desta sub-árvore, assim, o ponteiro AB aponta para o nó que o originou, o qual é o de valor 5.

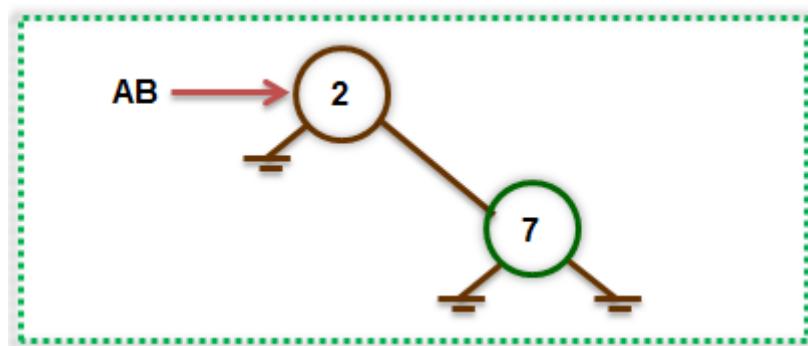


Sub-árvore direita do nó de valor 5 desempilhada, retorno NULL.

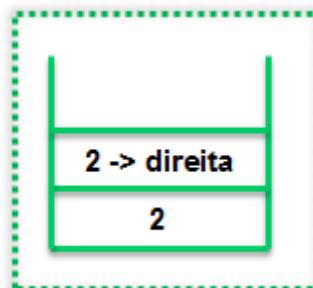
**9º PASSO:** As sub-árvores **esquerda** e **direita** do nó de valor **5** foram analisadas, assim, **desaloca** o nó de valor 5 e haverá o retorno NULL para a função “pai” do elemento de valor 5, ou seja, para sub-árvore esquerda do nó de valor 2, assim, o ponteiro **AB** aponta para o elemento de valor 2.



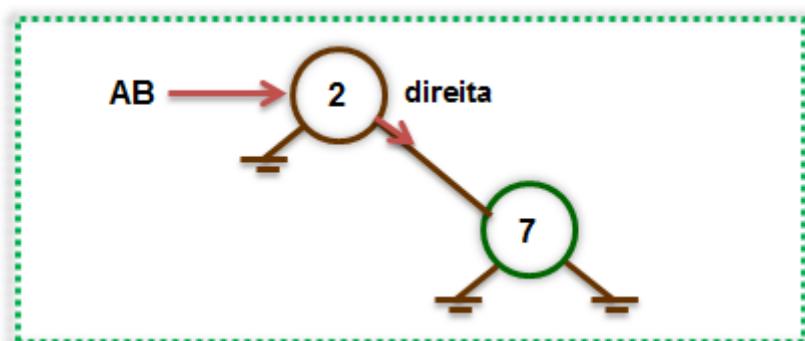
Desempilha o nó de valor 5 e a sub-árvore esquerda do elemento de valor 2.



**8º PASSO:** A **sub-árvore esquerda** do nó de valor 2 foi analisada. O ponteiro AB aponta para este, assim, chama-se a função `arv_liberar`, recursivamente, passando como parâmetro a **sub-árvore direita** (`arv_liberar(AB -> direita)`) do nó de valor 2. O ponteiro **AB** aponta para este, sendo que o ponteiro **direita** aponta para o nó de valor 7.



Empilha a sub-árvore direita do nó de valor 2.

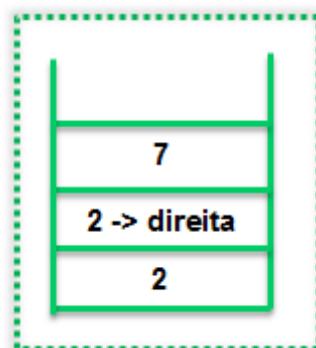
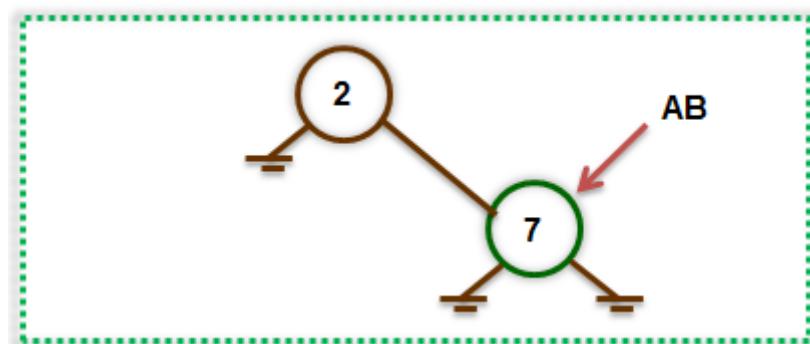


No parâmetro  $AB = AB \rightarrow direita$ :

$\text{arv\_liberar}(\text{Arv\_B}^* \text{AB})$   
 ↓  
 $\text{arv\_liberar}(\text{AB} \rightarrow \text{direita})$

Assim, a função é definida por:

$\text{arv\_liberar}(\text{Arv\_B}^* \text{AB}) = \text{arv\_liberar}(<\text{nó de valor } 7>)$

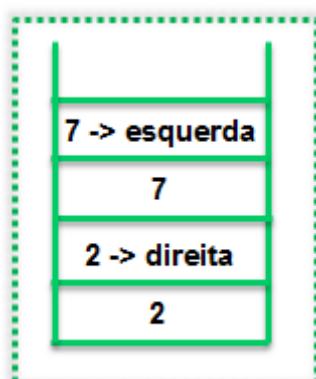


Empilha o nó de valor 7.

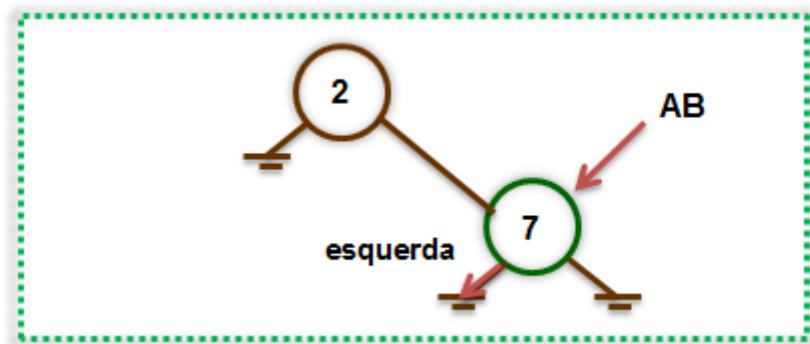
**9º PASSO:** Entra na condicional if.

- Chama-se a função `arv_vazia`, a árvore não está vazia? SIM!! Pois o ponteiro **AB**, aponta para a raiz, a qual possui valor 7, ou seja, diferente de `NULL`.

**10º PASSO:** Chama a função `arv_liberar`, recursivamente, passando como parâmetro a **sub-árvore esquerda** do nó de valor 7. O ponteiro **AB** aponta para este, sendo que o ponteiro **esquerda** deste aponta para `NULL`.



Empilha a sub-árvore esquerda do nó de valor 7.



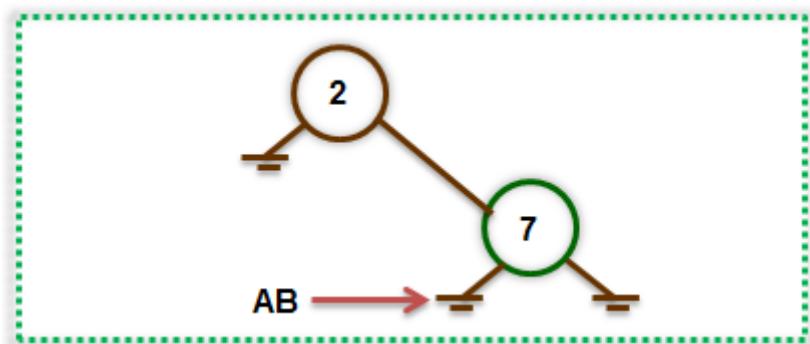
No parâmetro  $AB = AB \rightarrow \text{esquerda}$ :

$\text{arv\_liberar}(\text{Arv\_B}^* \text{AB})$

$\uparrow$   
 $\text{arv\_liberar}(\text{AB} \rightarrow \text{esquerda})$

Assim, a função é definida por:

$\text{arv\_liberar}(\text{Arv\_B}^* \text{AB}) = \text{arv\_liberar}(<\text{NULL}>)$



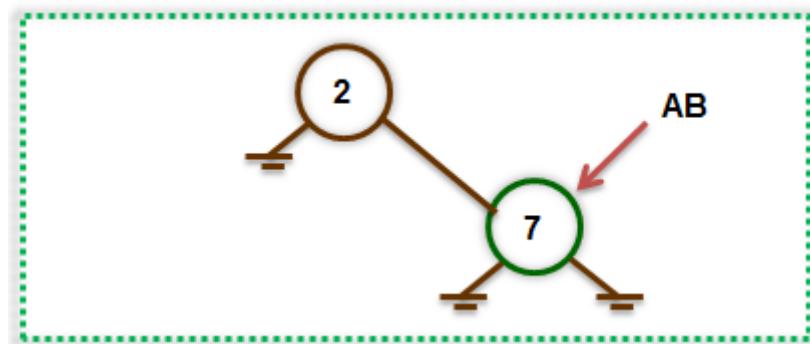
**11º PASSO:** Entra na condicional if.

- Chama-se a função `arv_vazia`, a árvore não está vazia? NÃO!! Pois o ponteiro `AB`, aponta para `NULL`.

**12º PASSO:** Retorna `NULL` para a função `arv_liberar(AB -> esquerda)`, haverá o retorno para a função “pai” desta sub-árvore, assim, o ponteiro `AB` aponta para o nó que o originou, o qual é o de valor 7.



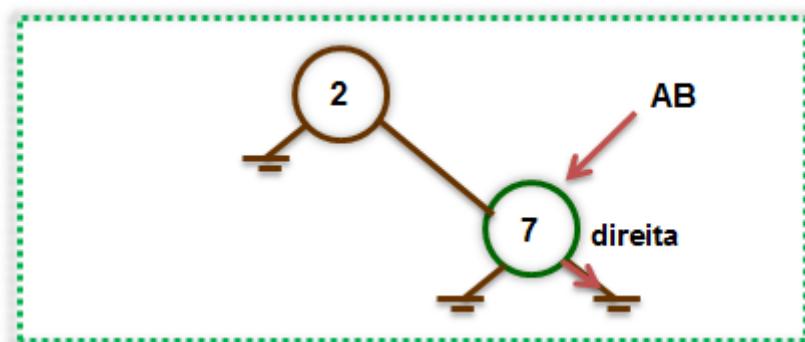
Sub-árvore esquerda do nó de valor 7 desempilhada, retorno `NULL`.



**13º PASSO:** A **sub-árvore esquerda** do nó de valor 7 foi analisada. O ponteiro AB aponta para este, assim, chama-se a função `arv_liberar`, recursivamente, passando como parâmetro a **sub-árvore direita** (`arv_liberar(AB -> direita)`) do nó de valor 7. O ponteiro **AB** aponta para este, sendo que o ponteiro **direita** aponta para NULL.



Empilha a sub-árvore direita do nó de valor 7.



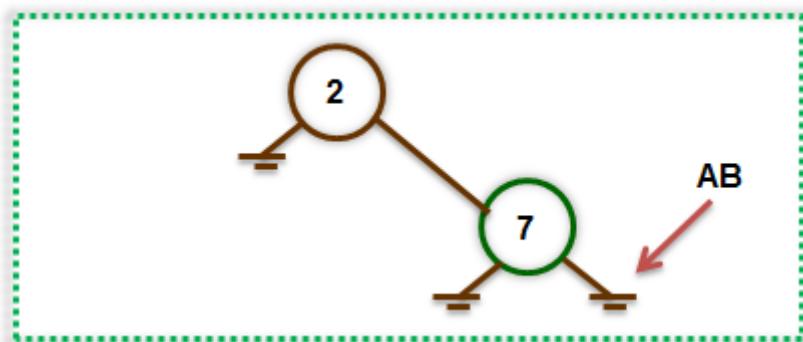
No parâmetro  $AB = AB \rightarrow direita$ :

$\text{arv\_liberar}(\text{Arv\_B}^* \text{AB})$

↑  
 $\text{arv\_liberar}(AB \rightarrow direita)$

Assim, a função é definida por:

$\text{arv\_liberar}(\text{Arv\_B}^* \text{AB}) = \text{arv\_liberar}(<\text{NULL}>)$



**14º PASSO:** Entra na condicional if.

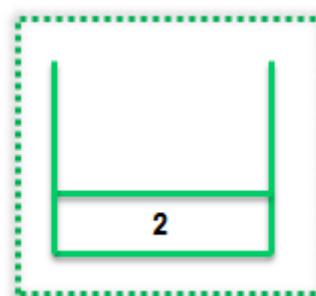
- Chama-se a função `arv_vazia`, a árvore não está vazia? NÃO!! Pois o ponteiro AB, aponta para NULL.

**15º PASSO:** Retorna NULL para a função `arv_liberar(AB -> direita)`, haverá o retorno para a função “pai” desta sub-árvore, assim, o ponteiro AB aponta para o nó que o originou, o qual é o de valor 7.

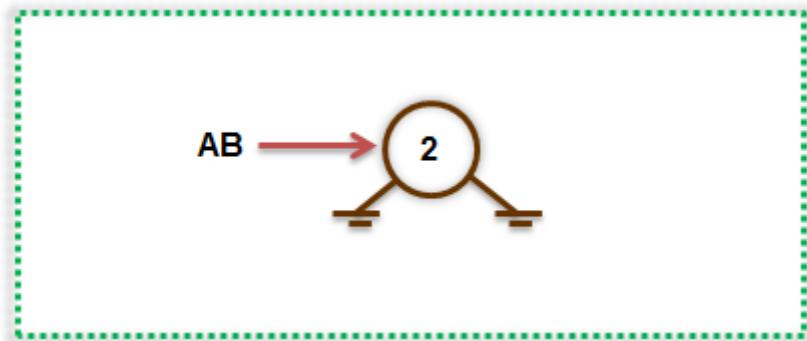


Sub-árvore direita do nó de valor 7 desempilhada, retorno NULL.

**16º PASSO:** As **sub-árvores esquerda e direita** do nó de valor 7 foram analisadas, assim, **desaloca** o nó de valor 7 e haverá o retorno NULL para a função “pai” do elemento de valor 7, ou seja, para sub-árvore direita do nó de valor 2, assim, o ponteiro AB aponta para o 2.



Desempilha o nó de valor 7 e a sub-árvore direita do elemento de valor 2.



**17º PASSO:** As **sub-árvores esquerda e direita** do nó de valor 2 foram analisadas, assim, **desaloca** o nó de valor 2.

**18º PASSO:** Retorna NULL para o ponteiro AB.

## 4.2. Árvore binária de busca

### 4.2.1. Conceitos

Uma Árvore Binária de Busca é caracterizada por **três** fatores:

- Não há elementos duplicados.
- Ordenada.
- Não balanceada.

### 4.2.2. Criação e Manipulação

Pode-se programar a criação da árvore binária de busca (ABB) e estabelecer as devidas funcionalidades a ela, entre estas se tem:

4.2.2.1. Criação de um Tipo Abstrato de Dado (TAD).

4.2.2.2. Criar a árvore binária.

4.2.2.3. Inserir elementos.

4.2.2.4. Averiguar se a árvore binária de busca está vazia.

4.2.2.5. Imprimir os elementos.

4.2.2.6. Buscar um determinado elemento.

4.2.2.7. Liberar todos os elementos.

4.2.2.8. Retirar um elemento.

#### 4.2.2.1. Criação de um tipo abstrato de dado (TAD)

Criação de um novo tipo de dado, como mostra a figura 114.

```
typedef struct arv_binaria_busca{
    int info;
    struct arv_binaria* esquerda;
    struct arv_binaria* direita;
}Arv_BBusca;
```

Figura 114. Estrutura de um novo tipo de dado do tipo Arv\_BBusca.

*Descrição das funcionalidades* apresentadas na figura 114:

1. Declarou-se uma variável (info), do tipo inteiro, a qual armazena o valor contido no elemento (figura 115).
2. Declarou-se o ponteiro denominado de esquerda, do tipo struct arv\_binaria\_busca, o qual se refere à sub-árvore esquerda (figura 115).
3. Declarou-se o ponteiro denominado de direita, do tipo struct arv\_binaria\_busca, o qual se refere à sub-árvore direita (figura 115).

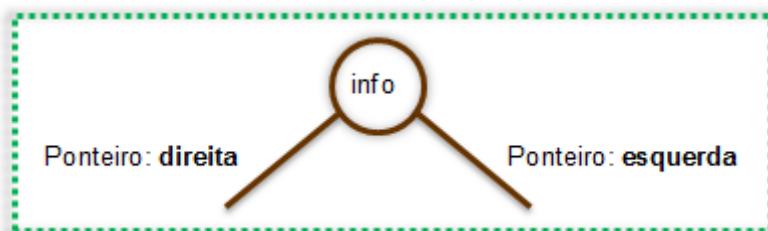


Figura 115. Regiões referentes ao ponteiro *esquerda*, ao ponteiro *direita* e a variável *info*.

#### 4.2.2.2. Criar a árvore binária de busca

##### O que fazer? Dicas...

Toda árvore deve inicializar vazia!!!

A função denominada **criar\_arv**, apresentada na figura 116, possui as seguintes características (graficamente, está função está representada na figura 99, apresentada anteriormente):

- A função não possui parâmetros.
- O retorno é do tipo **Arv\_BBusca**, retornando o valor nulo (NULL).

```
Arv_BBusca* criar_arv(void){  
    return NULL;  
}
```

Figura 116. Função referente à atribuição NULL para a árvore binária de busca.

##### Observação

Na função principal (**main()**) a função para criar a árvore deve ser chamada como:

**ABB = criar\_arv();**

#### 4.2.2.3. Inserir elementos

##### O que fazer? Dicas...

- Deve alocar memória, de maneira dinâmica, para cada nó a ser inserido.
- Os valores dos nós presentes na **sub-árvore esquerda** devem ser **menores** quando comparados ao valor da raiz.
- Os valores dos nós presentes na **sub-árvore direita** devem ser **maiores** quando comparados ao valor da raiz.
- O ponteiro **esquerda** e **direita**, do tipo **Arv\_BBusca**, composição do ponteiro **ABB**, apontam para a sub-árvore esquerda e direita, respectivamente, ou para **NULL**.

A função denominada **arv\_inserir**, apresentada na figura 117, possui as seguintes características:

- A função possui como parâmetros o ponteiro denominado de **sae** e outro de **sad**, os quais recebem as sub-árvores esquerda e direita, respectivamente, do tipo **Arv\_BBusca**, e o valor a ser inserido, do tipo inteiro.
- O retorno da função é do tipo **Arv\_BBusca**, retornando o novo nó, assim, encadeando o elemento na árvore existente.

```
Arv_BBusca* arv_inserir (Arv_BBusca* ABB, valor){
```

- 1       **if(ABB == NULL){**
- ABB = (Arv\_BBusca\*)malloc(sizeof(Arv\_BBusca));**
- ABB -> info = valor;**
- ABB -> esquerda = NULL;**
- ABB -> direita = NULL;**
- }**
- 2       **else if(valor < ABB -> info)**
- ABB -> esquerda = arv\_inserir(ABB -> esquerda, valor);**
- 3       **else**
- ABB -> direita = arv\_inserir (ABB -> direita, valor);**
- 4       **return ABB;**
- }**

Figura 117. Função referente à inserir elementos árvore binária de busca.

*Descrição das funcionalidades* apresentadas figura 117:

1. Condicional if. Se o ponteiro ABB aponta para NULL a árvore está vazia, assim, a raiz será criada, fornecendo um valor a esta e seus ponteiros esquerda e direita inicialmente apontam para NULL, para posteriormente os demais nós serem inseridos.
2. Primeira condicional else. A raiz ou o nó raiz já existe, assim, se o valor a ser inserido for **menor** que o valor encontrado na raiz ou no nó raiz será alocado na sub-árvore esquerda.
3. Segunda condicional else. A raiz ou o nó raiz já existe, assim, se o valor a ser inserido for **maior** que o valor encontrado na raiz ou no nó raiz será alocado na sub-árvore direita.
4. Retorna a árvore criada.

Na função principal (**main()**) deve constar (figura 118):

```
main(void){
    1     Arv_BBusca* ABB;
    2     ABB = criar_arv0;
    3     ABB = arv_inserir (ABB, <valor a ser inserido>);
}
```

Figura 118. Função principal chamar a função criar\_arv e inserir\_nos na árvore binária de busca.

*Descrição das funcionalidades* apresentadas na figura 118:

1. Cria-se um ponteiro ABB, do tipo Arv\_BBusca, o qual referenciará a raiz da árvore.
2. Chamar a função **criar\_arv** para atribuir o primeiro valor a árvore, o qual é nulo. O ponteiro ABB declarado recebe o retorno da função **criar\_arv**.
3. Chamar a função **inserir\_nos** passando como parâmetros o ponteiro que referencia a árvore e o valor a ser inserido.

Para melhor entendimento, graficamente, realiza-se dois exemplos a seguir, utilizando o conceito *teste de mesa*.

### Exemplo

Inserir os elementos 5 e 1.

**1º PASSO:** Na função principal deve constar:

```

main(void){
    Arv_BBusca* ABB;
    ABB = criar_arv();
    ABB = arv_inserir(ABB, 5);
    ABB = arv_inserir(ABB, 1);
}

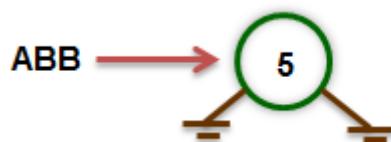
```

**2º PASSO:** O ponteiro ABB recebe o retorno da função **criar\_arv()**, o qual é NULL.



**3º PASSO:** A função **arv\_inserir (ABB, 5)** realiza os seguintes procedimentos:

- O ponteiro ABB aponta para NULL? SIM!!
- Aloca-se memória.
- O ponteiro ABB recebe o valor 5.
- Os ponteiros esquerda e direita apontam para NULL.



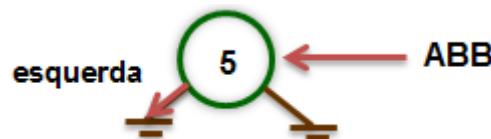
- Retorna o ponteiro ABB.

**4º PASSO:** A função **arv\_inserir (ABB, 1)** realiza os seguintes procedimentos:

- O ponteiro ABB aponta para NULL? NÃO!! Aponta para o nó de valor 5.
- O valor 1 é menor que o valor do nó (5) apontado pelo ponteiro ABB? SIM!! O ponteiro esquerda, do nó apontado pelo ponteiro ABB, aguarda o retorno da chamada recursiva da função **arv\_inserir(ABB -> esquerda, 1)**.

**5º PASSO:** A função **arv\_inserir(ABB -> esquerda, 1)** realiza os seguintes procedimentos:

- O ponteiro esquerda do nó de valor 5 aponta para NULL.



No parâmetro  $ABB = ABB \rightarrow esquerda$ :

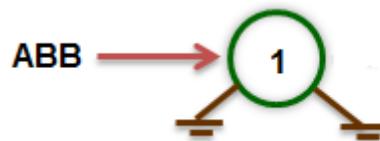
`arv_inserir(Arv_BBusca* ABB, 1)`



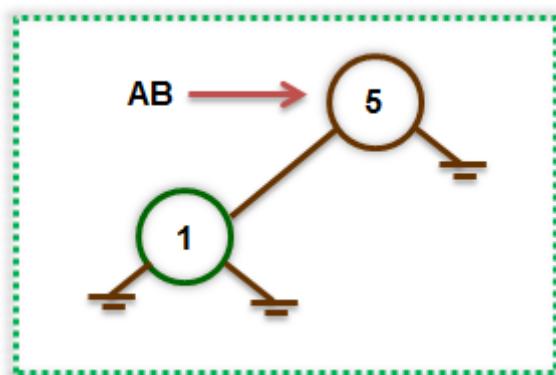
Assim, a função é definida por:

`arv_inserir(Arv_BBusca* ABB, 1) = arv_inserir(<NULL>, 1)`

- O ponteiro ABB aponta para NULL? SIM!!
- Aloca-se memória.
- O ponteiro ABB recebe o valor 1.
- Os ponteiros esquerda e direita apontam para NULL.



- Retorna o ponteiro ABB.
- No passo 4, o ponteiro esquerda, do nó (5) apontado pelo ponteiro ABB, aguarda o retorno da chamada recursiva da função `arv_inserir(ABB -> esquerda, 1)`.



#### 4.2.2.4. Averiguar se a árvore binária de busca está vazia

O que fazer? Dicas...

Verificar se a árvore aponta para o valor nulo (NULL).

A função denominada **arv\_vazia**, apresentada na figura 119, possui as seguintes características:

- A função possui como parâmetro o ponteiro ABB, o qual aponta para a raiz da árvore.
- O retorno é do tipo **booleano**, retornando valor 1 (verdadeiro) quando a árvore estiver vazia, caso contrário, retorna valor 0 (falso);

```
int arv_vazia(Arv_BBusca* ABB){
    return ABB == NULL;
}
```

Figura 119. Função que verifica se a árvore binária de busca está vazia.

#### 4.2.2.5. Imprimir os elementos

##### O que fazer? Dicas...

- Escolher qual tipo de percurso será utilizado para percorrer a árvore.

A função denominada **arv\_imprimir\_pre\_ordem**, apresentada na figura 120, possui as seguintes características:

- A função possui como parâmetro o ponteiro ABB, o qual aponta para a raiz árvore.
- O retorno é do tipo void, retornando na tela os valores contidos na árvore.

```
void arv_imprimir_pre_ordem(Arv_BBusca* ABB){

1   if(!arv_vazia(ABB)){
        printf("%d ", ABB -> info);      /* mostra raiz*/
        arv_imprimir(ABB -> esquerda);    /* mostra sae*/
        arv_imprimir(ABB -> direita);     /* mostra sad*/
    }
}
```

Figura 120. Função imprimir em pré-ordem.

*Descrição das funcionalidades* apresentadas na figura 120:

- Dentro da condicional chama-se a função arv\_vazia (a árvore não está vazia?), caso retorne o valor 1 a árvore não está vazia e, assim, considerando a impressão do tipo **pré-ordem**:

- Imprimi-se o valor contido no elemento.
- Chamar a função arv\_imprimir, recursivamente, passando como parâmetro a região **esquerda** da árvore.
- Chamar a função arv\_imprimir, recursivamente, passando como parâmetro a região **direita** da árvore.

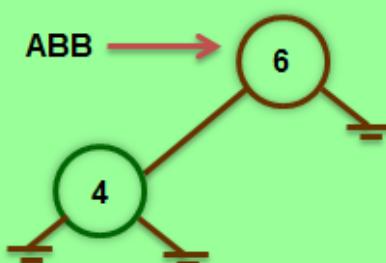
#### Observação

Na função principal (**main()**) a função para imprimir os valores contidos na árvore deve ser chamada como:

**imprimir(ABB);**

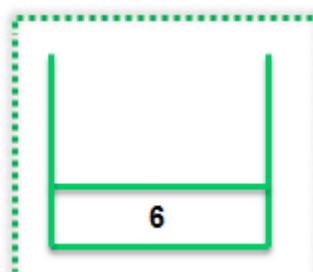
Para melhor entendimento, graficamente, realiza-se um exemplo a seguir, utilizando o conceito *teste de mesa*.

#### Exemplo



**1º PASSO:** Entra na condicional if. A função é definida por:

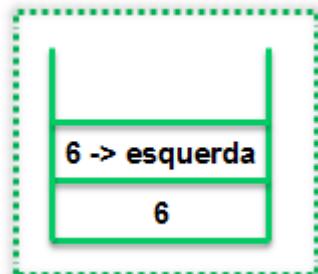
**arv\_imprimir(Arv\_BBusca\* ABB) = arv\_imprimir(<nó de valor 6>)**



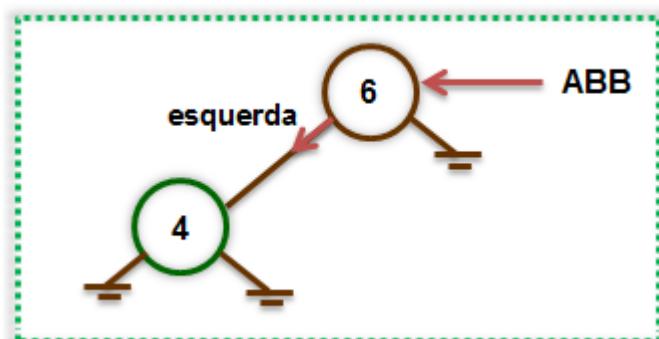
Empilha o 6.

- Chama-se a função arv\_vazia, a árvore não está vazia? SIM!! Pois o ponteiro AB, aponta para a raiz, a qual possui valor 6, ou seja, diferente de NULL.
- Imprimi-se o valor 6 contido no elemento.

**2º PASSO:** Chama a função arv\_imprimir, recursivamente, passando como parâmetro a **sub-árvore esquerda** do nó de valor 6. O ponteiro **ABB** aponta para este, sendo que o ponteiro **esquerda** deste aponta para o elemento de valor 4.



Empilha a sub-árvore esquerda do nó de valor 6.



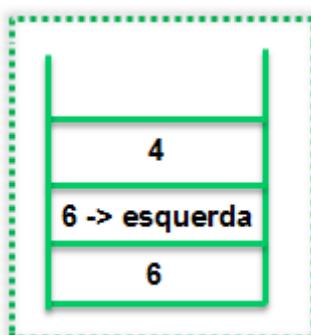
No parâmetro  $ABB = ABB \rightarrow \text{esquerda}$ :

$\text{arv\_imprimir}(\text{Arv\_BBusca}^* \text{ ABB})$

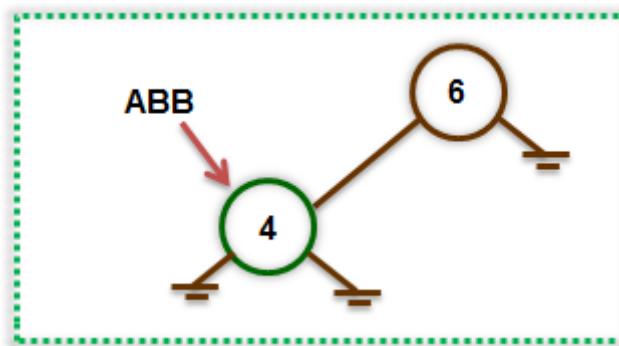
↑  
 $\text{arv\_imprimir}(\text{ABB} \rightarrow \text{esquerda})$

Assim, a função é definida por:

$\text{arv\_imprimir}(\text{Arv\_BBusca}^* \text{ ABB}) = \text{arv\_imprimir}(<\text{nó de valor } 4>)$



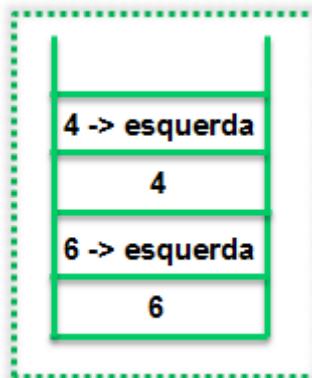
Empilha o nó de valor 4.



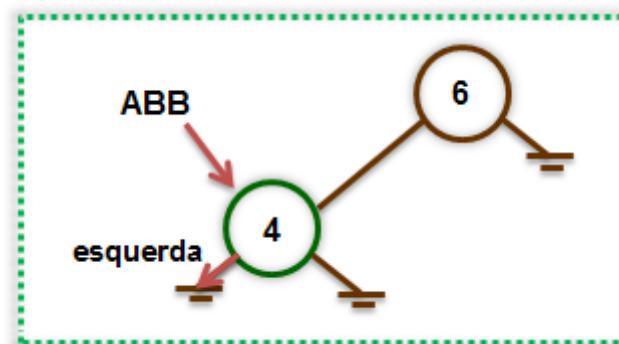
**3º PASSO:** Entra na condicional if.

- Chama-se a função `arv_vazia`, a árvore não está vazia? SIM!! Pois o ponteiro `AB`, aponta para a raiz, a qual possui valor 4, ou seja, diferente de `NULL`.
- Imprimi-se o valor 4 contido no elemento.

**4º PASSO:** Chamar a função `arv_imprimir`, recursivamente, passando como parâmetro a sub-árvore **esquerda** do nó de valor 4. O ponteiro **ABB** aponta para este, sendo que o ponteiro **esquerda** deste aponta para `NULL`.



Empilha a sub-árvore esquerda do nó de valor 4.



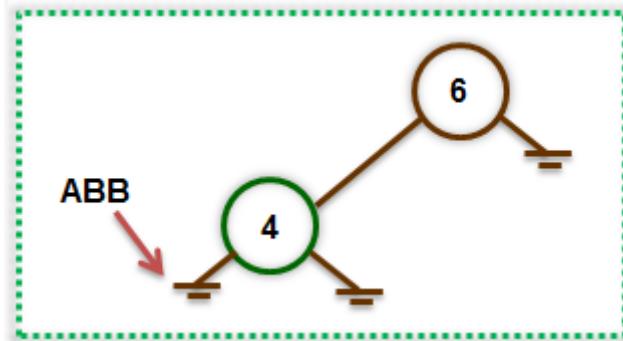
No parâmetro  $ABB = ABB \rightarrow esquerda$ :

`arv_imprimir(Arv_BBusca* ABB)`

$\overbrace{\quad\quad\quad}$   
`arv_imprimir(ABB -> esquerda)`

Assim, a função é definida por:

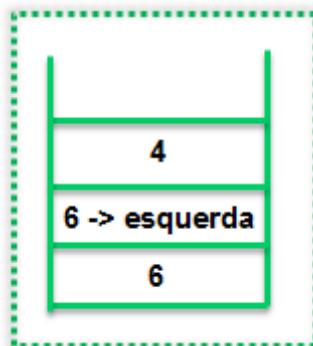
$$\text{arv_imprimir}(\text{Arv_BBusca}^* \text{ ABB}) = \text{arv_imprimir}(<\text{NULL}>)$$



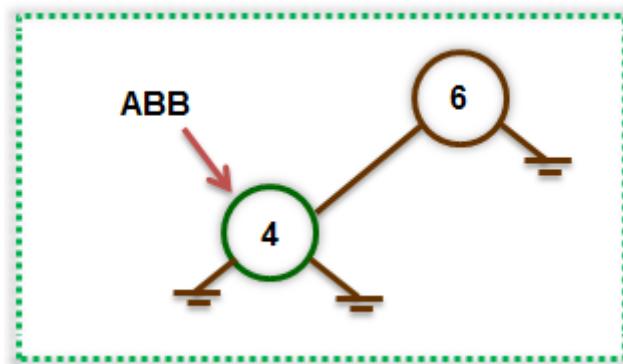
**5º PASSO:** Entra na condicional if.

- Chama-se a função `arv_vazia`, a árvore não está vazia? NÃO!! Pois o ponteiro `ABB` aponta para `NULL`.

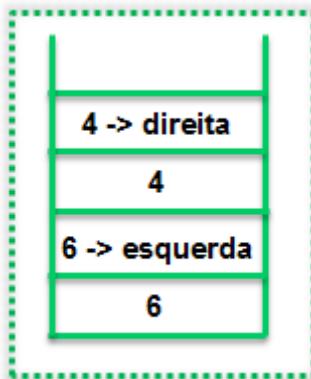
**6º PASSO:** Como encontrou o valor nulo para a função `arv_imprimir(ABB -> esquerda)` haverá o retorno para a função “pai” desta sub-árvore, assim, o ponteiro `ABB` aponta para o nó que o originou, o qual é o de valor 4.



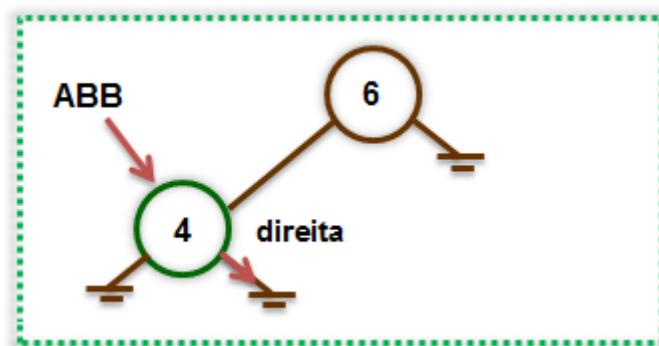
Sub-árvore esquerda do nó de valor 4 desempilhada, retorno `NULL`.



**7º PASSO:** A **sub-árvore esquerda** do nó de valor 4 foi analisada. O ponteiro `ABB` aponta para este, assim, chama-se a função `arv_imprimir`, recursivamente, passando como parâmetro a **sub-árvore direita** (`arv_imprimir(ABB -> direita)`) do nó de valor 4. O ponteiro **ABB** aponta para este, sendo que o ponteiro **direita** aponta para `NULL`.



Empilha a sub-árvore direita do nó de valor 4.

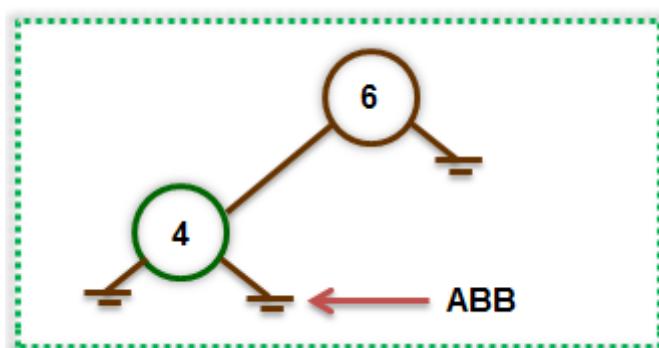


No parâmetro  $ABB = ABB \rightarrow direita$ :

$\text{arv\_imprimir}(\text{Arv\_BBusca}^* \text{ ABB})$   
 ↑  
 $\text{arv\_imprimir}(\text{ABB} \rightarrow \text{direita})$

Assim, a função é definida por:

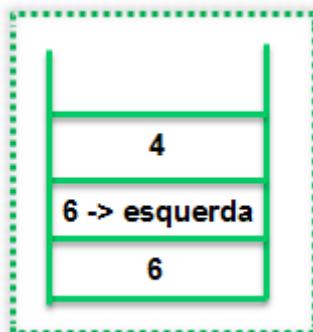
$\text{arv\_imprimir}(\text{Arv\_BBusca}^* \text{ ABB}) = \text{arv\_imprimir}(<\text{NULL}>)$



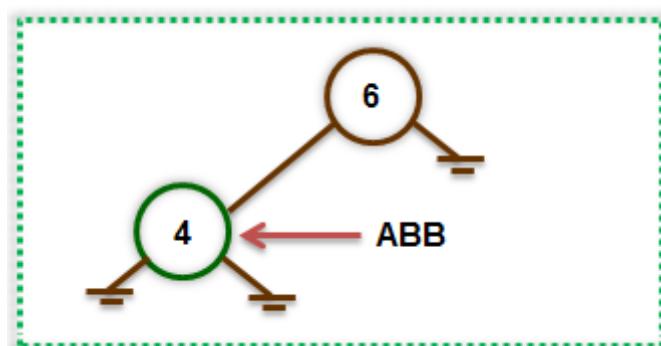
**8º PASSO:** Entra na condicional if.

- Chama-se a função `arv_vazia`, a árvore não está vazia? NÃO!! Pois o ponteiro `AB` aponta para `NULL`.

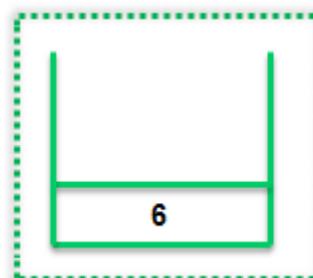
**9º PASSO:** Como encontrou o valor nulo para a função `arv_imprimir(ABB -> direita)` haverá o retorno para a função “pai” desta sub-árvore, assim, o ponteiro ABB aponta para o nó que o originou, o qual é o de valor 4.



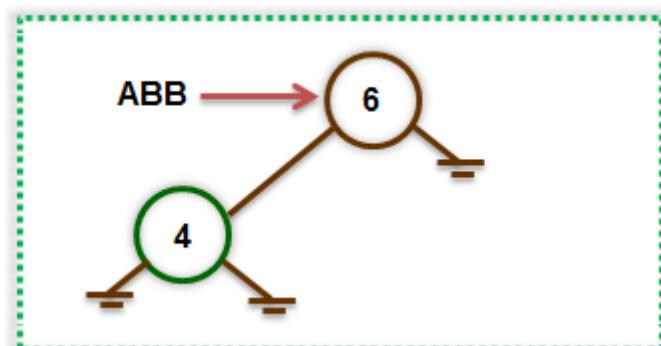
Sub-árvore direita do nó de valor 4 desempilhada, retorno NULL.



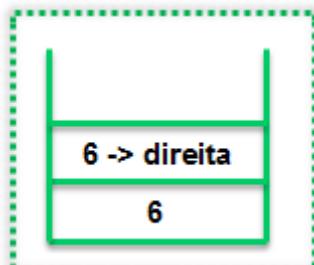
**10º PASSO:** As **sub-árvores esquerda e direita** do nó de valor **4** foram analisadas, assim, haverá o retorno para a função “pai” do elemento de valor 4, ou seja, a sub-árvore esquerda do nó de valor 6 foi analisada, assim, o ponteiro **ABB** aponta para o **6**.



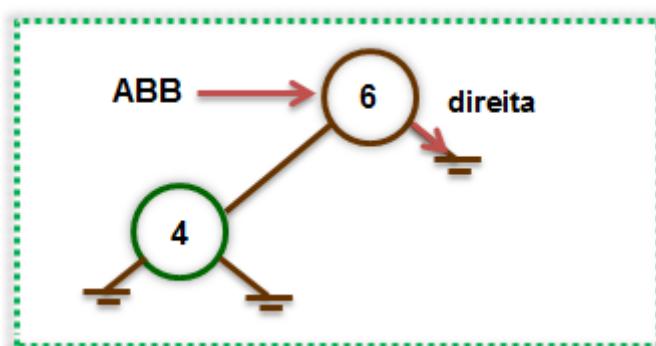
Desempilha o nó de valor 4 e a sub-árvore esquerda do elemento de valor 6.



**11º PASSO:** A **sub-árvore esquerda** do nó de valor 6 foi analisada. O ponteiro ABB aponta para este, assim, chama-se a função `arv_imprimir`, recursivamente, passando como parâmetro a **sub-árvore direita** (`arv_imprimir(ABB -> direita)`) do nó de valor **6**. O ponteiro **ABB** aponta para este, sendo que o ponteiro **direita** aponta para **NULL**.



Empilha a sub-árvore direita do nó de valor 6.



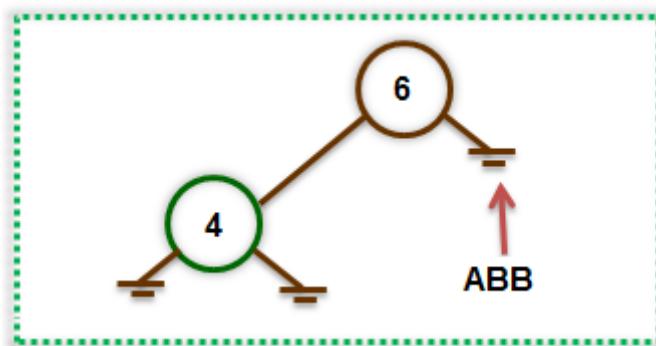
No parâmetro  $ABB = ABB \rightarrow direita$ :

$\text{arv\_imprimir}(\text{Arv\_BBusca}^* ABB)$

↑  
 $\text{arv\_imprimir}(ABB \rightarrow direita)$

Assim, a função é definida por:

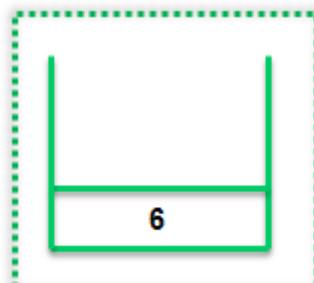
$\text{arv\_imprimir}(\text{Arv\_BBusca}^* ABB) = \text{arv\_imprimir}(<\text{NULL}>)$



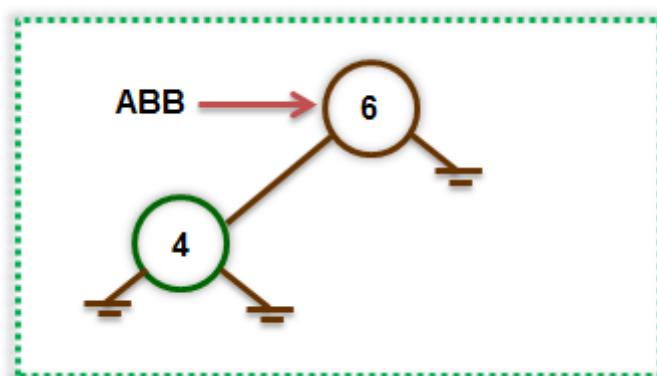
**12º PASSO:** Entra na condicional if.

- Chama-se a função `arv_vazia`, a árvore não está vazia? NÃO!! Pois o ponteiro AB aponta para **NULL**.

**13º PASSO:** Como encontrou o valor nulo para a função `arv_imprimir(ABB -> direita)` haverá o retorno para a função “pai” desta sub-árvore, assim, o ponteiro ABB aponta para o nó que o originou, o qual é o de valor 6.



Sub-árvore direita do nó de valor 6 desempilhada, retorno NULL.



**14º PASSO:** As **sub-árvores esquerda e direita** do nó de valor 6 foram analisadas, assim, terminou-se a impressão!

#### 4.2.2.6. Verificar se um determinado elemento pertence a árvore

##### O que fazer? Dicas...

- Percorrer a árvore comparando o valor que o nó possui com o valor a ser buscado.
- Caso encontre, retorna 1, ou seja, indica a ocorrência deste valor na árvore.

A função denominada `arv_pertence`, apresentada na figura 121, possui as seguintes características:

- A função possui como parâmetro o valor a ser procurado e o ponteiro ABB, o qual possui acesso a raiz.
- O retorno é do tipo **booleano**, retornando valor 1 indicando a **ocorrência** do valor, caso contrário, retorna valor 0 (falso);

```

int arv_pertence(Arv_BBusca* ABB, int valor){

1    if(arv_vazia(ABB))
        return 0; /* árvore vazia: não encontrou*/
2    else
        return ABB -> info == valor ||
               arv_pertence(ABB -> esquerda, valor) ||
               arv_pertence(ABB -> direita, valor)
}

```

Figura 121. Função para verificar se um dado elemento pertence a árvore.

#### *Descrição das funcionalidades* apresentadas na figura 121:

1. Dentro da condicional chama-se a função arv\_vazia, caso retorne o valor 1 a árvore está vazia e, assim, retorna 0.
2. Senão:
  - Compara-se o valor contido no elemento apontado pelo ponteiro ABB com o valor a ser procurado, caso o encontre retorna 1.
  - Se a árvore não está vazia chama-se recursivamente a função arv\_pertence tanto para a região esquerda quanto para a região direita.

#### Observação

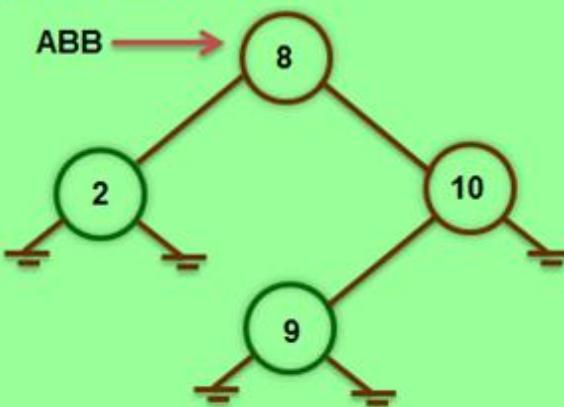
Na função principal (**main()**) a função para verificar a presença de um valor contido na árvore deve ser chamada como:

```
int resultado = arv_pertence(ABB, <valor a ser procurado>);
```

Para melhor entendimento, graficamente, realiza-se um exemplo a seguir, utilizando o conceito *teste de mesa*.

#### Exemplo

Verificar se o valor 2 se encontra na árvore:



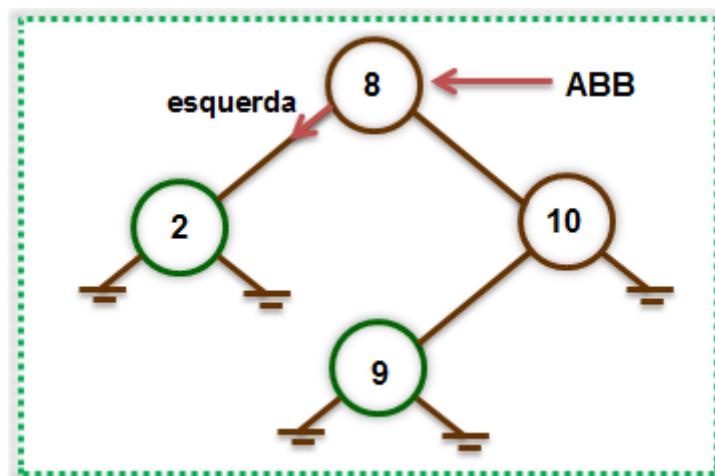
**1º PASSO:** Entra na condicional if. O parâmetro da função arv\_pertence é o **nó de valor 8**.

- Chama-se a função arv\_vazia, a árvore está vazia? NÃO!! Pois o ponteiro ABB aponta para a raiz, o qual possui valor 8, ou seja, diferente de NULL.

**2º PASSO:** Entra na condicional else.

- *Verifica a primeira condição:* o campo de informação do nó apontado pelo ponteiro ABB (ABB -> info) é igual ao valor a ser buscado, ou seja, 8 é igual à 2? NÃO!

**3º PASSO:** Chamar a função arv\_pertence, recursivamente, passando como parâmetro a **sub-árvore esquerda** do nó de valor 8. O ponteiro **ABB** aponta para este, sendo que o ponteiro **esquerda** deste aponta para o elemento de valor 2.



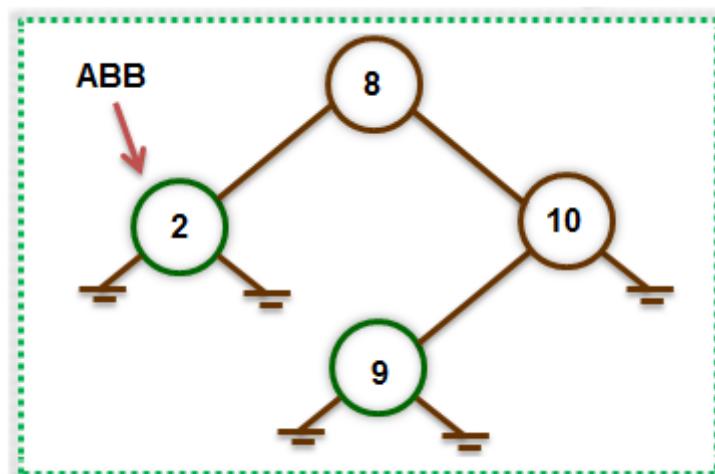
No parâmetro  $ABB = ABB \rightarrow esquerda$ :

$arv\_pertence(Arv\_BBusca^* ABB, 2)$

$\uparrow$   
 $arv\_pertence(ABB \rightarrow esquerda, 2)$

Assim, a função é definida por:

$arv\_pertence(Arv\_BBusca^* ABB, 2) = arv\_pertence(<\text{nó de valor } 2>, 2)$



**4º PASSO:** Entra na condicional if.

- Chama-se a função arv\_vazia, a árvore está vazia? NÃO!! Pois o ponteiro ABB, aponta para o nó de valor 2.

**57º PASSO:** Entra na condicional else.

- *Verifica a primeira condição:* o campo de informação do nó apontado pelo ponteiro ABB (ABB -> info) é igual ao valor a ser buscado, ou seja, 2 é igual à 2? SIM!
- Retorna 1, ou seja, o valor 2 se encontra na árvore, assim, termina-se a busca.

#### 4.2.2.7. Liberar todos os elementos

##### O que fazer? Dicas...

- Percorrer a árvore desalocando cada nó.
- Após desalocar todos os elementos retornar NULL para o ponteiro que referencia a árvore.

A função denominada **arv\_liberar**, apresentada na figura 122, possui as seguintes características:

- A função possui como parâmetro o ponteiro ABB.
- O retorno é do tipo criado.

```
Arv_BBusca* arv_liberar(Arv_BBusca* ABB){
    1   if(!arv_vazia(ABB)){
        arv_liberar(ABB -> esquerda); /* liberar sae*/
        arv_liberar(ABB -> direita); /* liberar sad*/
        free(ABB);                  /* liberar raiz*/
    }
    2   return NULL;
}
```

Figura 122. Função liberar a árvore.

*Descrição das funcionalidades* apresentadas na figura 122:

1. Dentro da condicional chama-se a função `arv_vazia`, caso retorne o valor 1 a árvore está vazia e, assim, retorna 0.
2. Senão:

- Chamar a função `arv_liberar`, recursivamente, passando como parâmetro a região **esquerda** da árvore.
- Chamar a função `arv_liberar`, recursivamente, passando como parâmetro a região **direita** da árvore.
- Libera o nó apontado pelo ponteiro ABB.
- Após a liberação de toda a árvore o valor nulo é retornado para o ponteiro ABB.

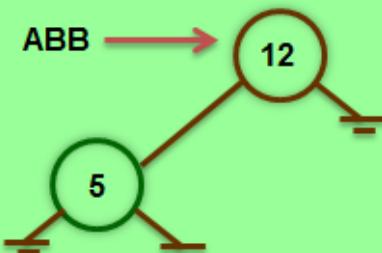
#### Observação

Na função principal (`main()`) a função para liberar a árvore deve ser chamada como:

`Arv_BBusca* ABB = arv_liberar(ABB);`

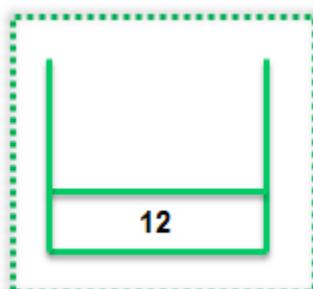
Para melhor entendimento, graficamente, realiza-se um exemplo a seguir, utilizando o conceito *teste de mesa*.

#### Exemplo



**1º PASSO:** Entra na condicional if. A função é definida por:

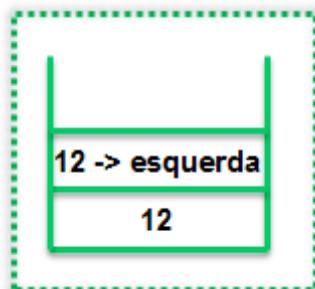
`arv_liberar(Arv_BBusca* ABB) = arv_liberar (<nó de valor 12>)`



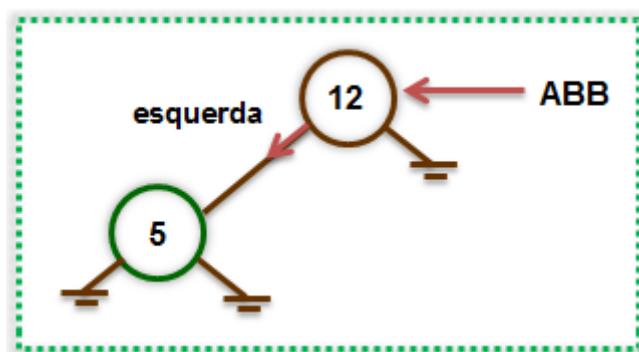
Empilha o nó de valor 12.

- Chama-se a função `arv_vazia`, a árvore não está vazia? SIM!! Pois o ponteiro `ABB`, aponta para a raiz, a qual possui valor 12, ou seja, diferente de `NONE`.

**2º PASSO:** Chama a função `arv_liberar`, recursivamente, passando como parâmetro a **sub-árvore esquerda** do nó de valor 12. O ponteiro `ABB` aponta para este, sendo que o ponteiro **esquerda** deste aponta para o elemento de valor 5.



Empilha a sub-árvore esquerda do nó de valor 12.



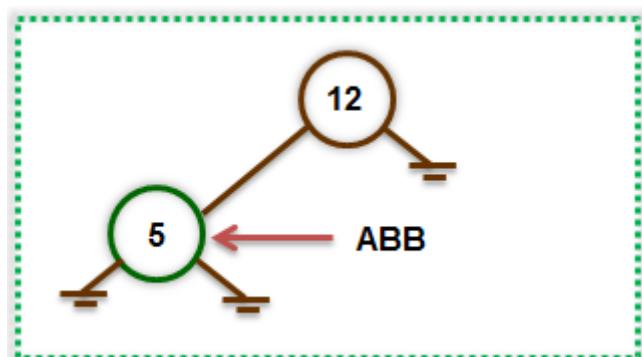
No parâmetro  $ABB = ABB \rightarrow \text{esquerda}$ :

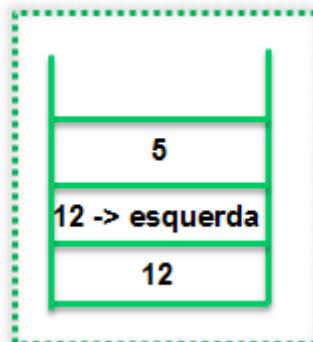
`arv_liberar(Arv_BBusca* ABB)`

↑  
`arv_liberar (ABB -> esquerda)`

Assim, a função é definida por:

`arv_liberar (Arv_BBusca* ABB) = arv_liberar(<nó de valor 5>)`



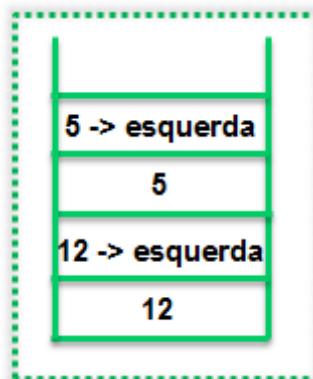


Empilha o nó de valor 5.

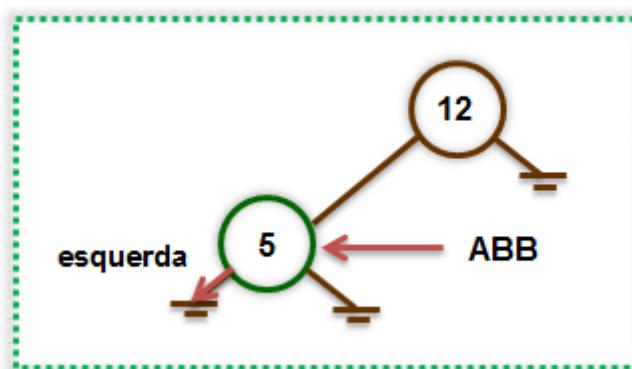
**3º PASSO:** Entra na condicional if.

- Chama-se a função `arv_vazia`, a árvore não está vazia? SIM!! Pois o ponteiro ABB, aponta para a raiz, a qual possui valor 5, ou seja, diferente de NULL.

**4º PASSO:** Chama a função `arv_liberar`, recursivamente, passando como parâmetro a sub-árvore **esquerda** do nó de valor 5. O ponteiro **ABB** aponta para este, sendo que o ponteiro **esquerda** deste aponta para NULL.



Empilha a sub-árvore esquerda do nó de valor 5.



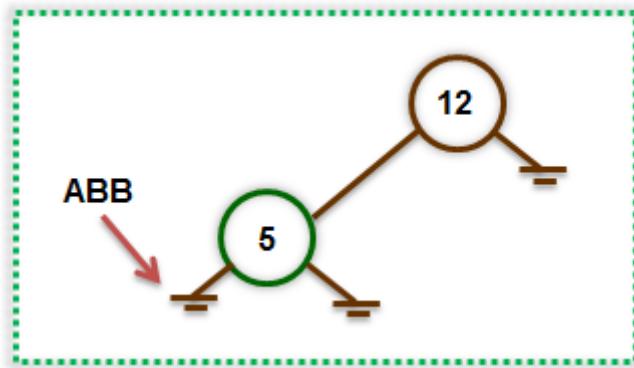
No parâmetro  $ABB = ABB \rightarrow esquerda$ :

`arv_liberar(Arv_BBusca* ABB)`

$\overbrace{\quad\quad\quad}$   
`arv_liberar(ABB -> esquerda)`

Assim, a função é definida por:

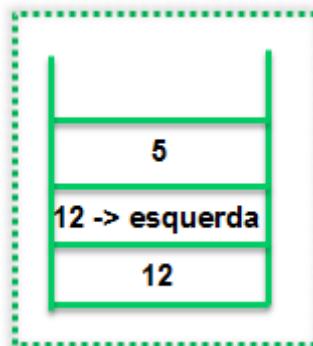
$$\text{arv\_liberar}(\text{Arv\_BBusca}^* \text{ ABB}) = \text{arv\_liberar}(<\text{NULL}>)$$



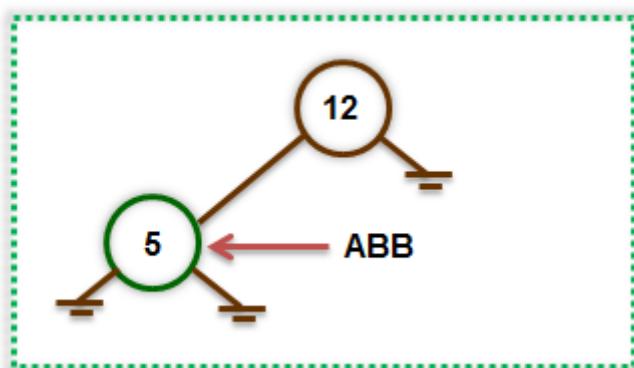
**4º PASSO:** Entra na condicional if.

- Chama-se a função arv\_vazia, a árvore não está vazia? NÃO!! Pois o ponteiro ABB, aponta para NULL.

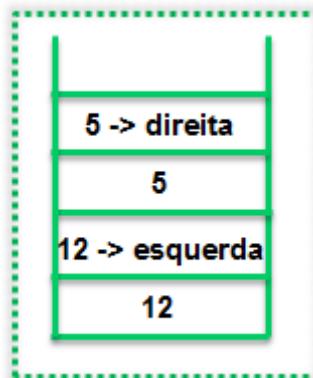
**5º PASSO:** Retorna NULL para a função arv\_liberar(ABB -> esquerda), haverá o retorno para a função “pai” desta sub-árvore, assim, o ponteiro ABB aponta para o nó que o originou, o qual é o de valor 5.



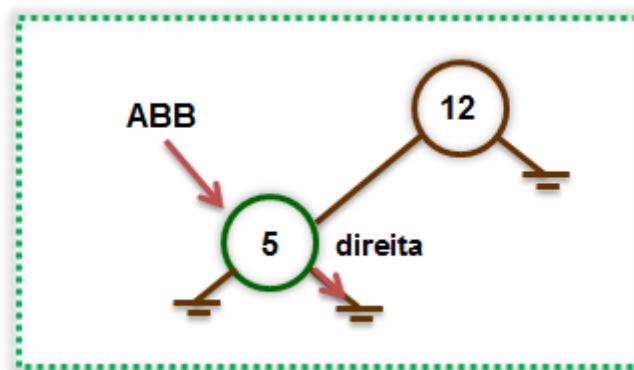
Sub-árvore esquerda do nó de valor 5 desempilhada, retorno NULL.



**6º PASSO:** A sub-árvore esquerda do nó de valor 5 foi analisada. O ponteiro ABB aponta para este, assim, chama-se a função arv\_liberar, recursivamente, passando como parâmetro a sub-árvore direita ( $\text{arv\_liberar}(ABB -> direita)$ ) do nó de valor 5. O ponteiro ABB aponta para este, sendo que o ponteiro direita aponta para NULL.



Empilha a sub-árvore direita do nó de valor 5.

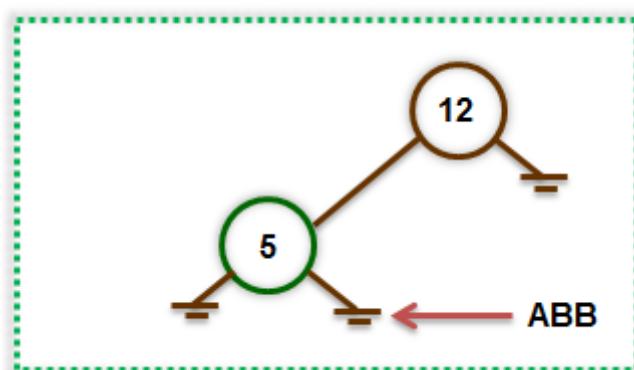


No parâmetro  $ABB = ABB \rightarrow direita$ :

```
arv_liberar(Arv_BBusca* ABB)
    |
    +-- arv_liberar(ABB -> direita)
```

Assim, a função é definida por:

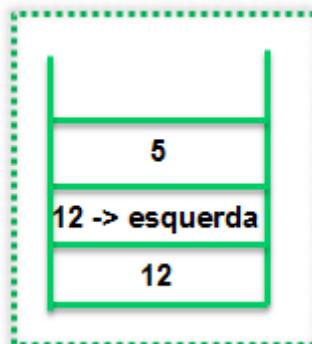
$$\text{arv\_liberar}(\text{Arv\_BBusca}^* \text{ ABB}) = \text{arv\_liberar}(<\text{NULL}>)$$



**7º PASSO:** Entra na condicional if.

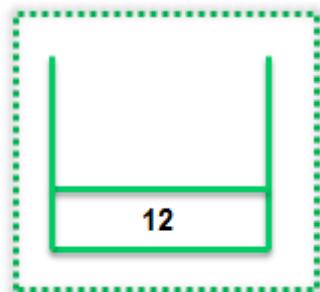
- Chama-se a função `arv_vazia`, a árvore não está vazia? NÃO!! Pois o ponteiro `ABB`, aponta para `NULL`.

**8º PASSO:** Retorna NULL para a função `arv_liberar(ABB -> direita)`, haverá o retorno para a função “pai” desta sub-árvore, assim, o ponteiro ABB aponta para o nó que o originou, o qual é o de valor 5.

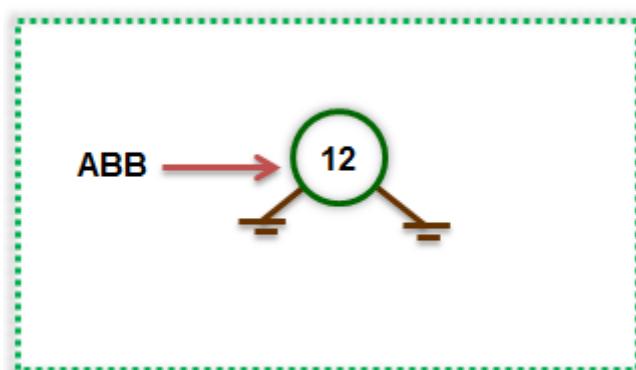


Sub-árvore direita do nó de valor 5 desempilhada, retorno NULL.

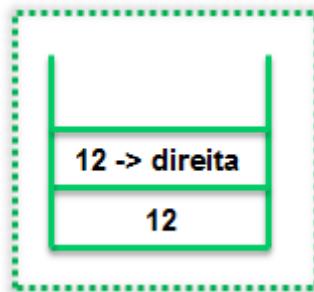
**9º PASSO:** As **sub-árvores esquerda e direita** do nó de valor 5 foram analisadas, assim, **desaloca-se** este, haverá o retorno NULL para a função “pai” do elemento de valor 5, ou seja, para sub-árvore esquerda do nó de valor 12, assim, o ponteiro **ABB** aponta para o **12**.



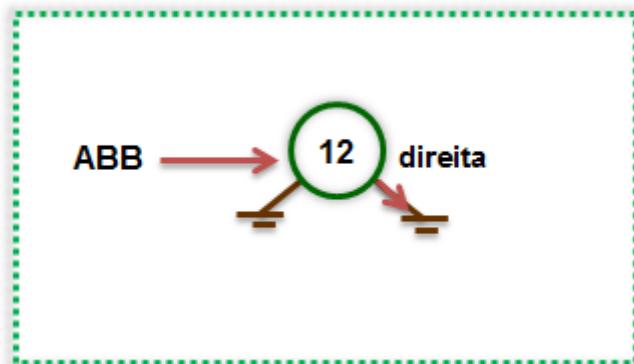
Desempilha o nó de valor 5 e a sub-árvore esquerda do elemento de valor 12.



**10º PASSO:** A **sub-árvore esquerda** do nó de valor 12 foi analisada. O ponteiro ABB aponta para este, assim, chama-se a função `arv_liberar`, recursivamente, passando como parâmetro a **sub-árvore direita** (`arv_liberar(ABB -> direita)`) do nó de valor **12**. O ponteiro **ABB** aponta para este, sendo que o ponteiro **direita** aponta para NULL.



Empilha a sub-árvore direita do nó de valor 12.



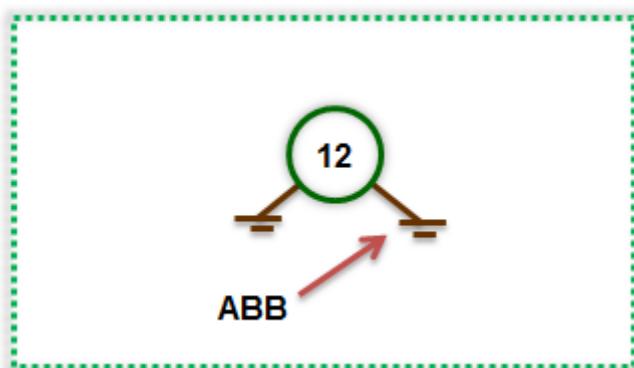
No parâmetro  $ABB = ABB \rightarrow direita$ :

$\text{arv\_liberar}(\text{Arv\_BBusca}^* \text{ ABB})$

↑  
 $\text{arv\_liberar}(\text{ABB} \rightarrow \text{direita})$

Assim, a função é definida por:

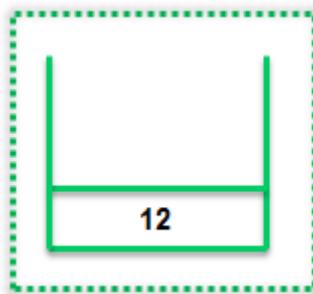
$\text{arv\_liberar}(\text{Arv\_BBusca}^* \text{ ABB}) = \text{arv\_liberar}(<\text{NULL}>)$



**11º PASSO:** Entra na condicional if.

- Chama-se a função  $\text{arv\_vazia}$ , a árvore não está vazia? NÃO!! Pois o ponteiro ABB, aponta para NULL.

**12º PASSO:** Retorna NULL para a função  $\text{arv\_liberar}(\text{ABB} \rightarrow \text{direita})$ , haverá o retorno para a função “pai” desta sub-árvore, assim, o ponteiro ABB aponta para o nó que o originou, o qual é o de valor 12.



Sub-árvore direita do nó de valor 12 desempilhada, retorno NULL.

**17º PASSO:** As **sub-árvores esquerda e direita** do nó de valor **12** foram analisadas, assim, **desaloca** o nó de valor 12.

**18º PASSO:** Retorna NULL para o ponteiro ABB.

#### 4.2.2.8. Retirar um elemento

##### O que fazer? Dicas...

- Percorrer a árvore, caso encontre o elemento a ser retirado este deverá ser removido da árvore. Caso contrário, retorna a árvore inalterada.

A função denominada **arv\_retirar**, apresentada na figura 123, possui as seguintes características:

- A função possui como parâmetro o ponteiro ABB e o valor a ser retirado.
- O retorno é do tipo criado.

*Descrição das funcionalidades* apresentadas na figura 123:

- Se a árvore estiver vazia retorna o valor nulo.
- Se o valor do nó, apontado pelo ponteiro ABB, for **maior** que o valor a ser retirado significa que este pode se encontrar na sub-árvore esquerda da árvore.
- Se o valor do nó, apontado pelo ponteiro ABB, for **menor** que o valor a ser retirado significa que este pode se encontrar na sub-árvore direita da árvore.
- Se o valor procurado **não** é maior e nem menor será igual assim se:
  - Se o nó que possui o valor a ser retirado **não** possui filhos desaloca-se este e retorna NULL para ABB.
  - Se o nó que possui o valor a ser retirado **possui filho** na sub-árvore **direita**:

- Um ponteiro auxiliar aponta para onde o ponteiro ABB aponta.
- O ponteiro ABB aponta para o filho, encontrado na sub-árvore direita, do nó a ser retirado.
- Desaloca-se o ponteiro auxiliar, ou seja, o nó com valor desejado.

4.3. Se o nó que possui o valor a ser retirado **possui filho** na **sub-árvore esquerda**:

- Um ponteiro auxiliar aponta para onde o ponteiro ABB aponta.
- O ponteiro ABB aponta para o filho, encontrado na sub-árvore esquerda, do nó a ser retirado.
- Desaloca-se o ponteiro auxiliar, ou seja, o nó com valor desejado.

4.4. Se o nó que possui o valor a ser retirado **possui filho** tanto na sub-árvore esquerda quanto na **direita**:

- Um ponteiro auxiliar aponta para onde o ponteiro esquerda do nó apontado pelo ponteiro ABB aponta.
- Enquanto o ponteiro direita do nó apontado pelo ponteiro auxiliar for diferente de NULL, o ponteiro auxiliar aponta para onde o ponteiro direita do ponteiro auxiliar apontar.
- Após o término do laço while, o campo de informação do nó apontado pelo ponteiro ABB recebe o valor do nó apontado pelo ponteiro auxiliar.
- O valor do nó apontado pelo ponteiro auxiliar recebe o valor a ser retirado.
- O ponteiro esquerda do nó apontado pelo ponteiro ABB recebe o retorno da chamada recursiva da função arv\_retirar, a qual possui como parâmetros ABB -> esquerda e o valor a ser retirado.

4.5. Retorna o ponteiro ABB.

#### Observação

Na função principal (**main()**) a função para retirar um nó da árvore deve ser chamada como:

**Arv\_BBusca\* ABB = arv\_retirar(ABB, <valor a ser retirado>);**

```

Arv_BBusca* arv_retirar(Arv_BBusca* ABB, valor){

    1      if(ABB == NULL)
            return NULL;

    2      else if(ABB -> info > valor)
            ABB -> esquerda = arv_retirar(ABB -> esquerda, valor);

    3      else if(ABB -> info < valor)
            ABB -> direita = arv_retirar(ABB -> direita, valor);

    4      else{ /* achou o nó a remover*/
            /* nó sem filhos*/
            4.1     if(ABB -> esquerda == NULL && ABB -> direita == NULL){
                    free(ABB);
                    ABB = NULL;
                }

            /* só tem filho à direita*/
            4.2     else if(ABB -> esquerda == NULL){
                    Arv_BBusca* aux1 = ABB;
                    ABB = ABB -> direita;
                    free(aux1);
                }

            /* só tem filho à esquerda*/
            4.3     else if(ABB -> direita == NULL){
                    Arv_BBusca* aux1 = ABB;
                    ABB = ABB -> esquerda;
                    free(aux1);
                }

            /* tem os dois filhos */
            4.4     else{
                    Arv_BBusca* aux2 = ABB -> esquerda;
                    while(aux2 -> direita != NULL){}
                        aux2 = aux2 -> direita;
                    }
                    ABB -> info = aux2 -> info; /* troca as informações*/
                    aux2 -> info = valor;
                    ABB -> esquerda = arv_retirar(ABB -> esquerda, valor);
                }

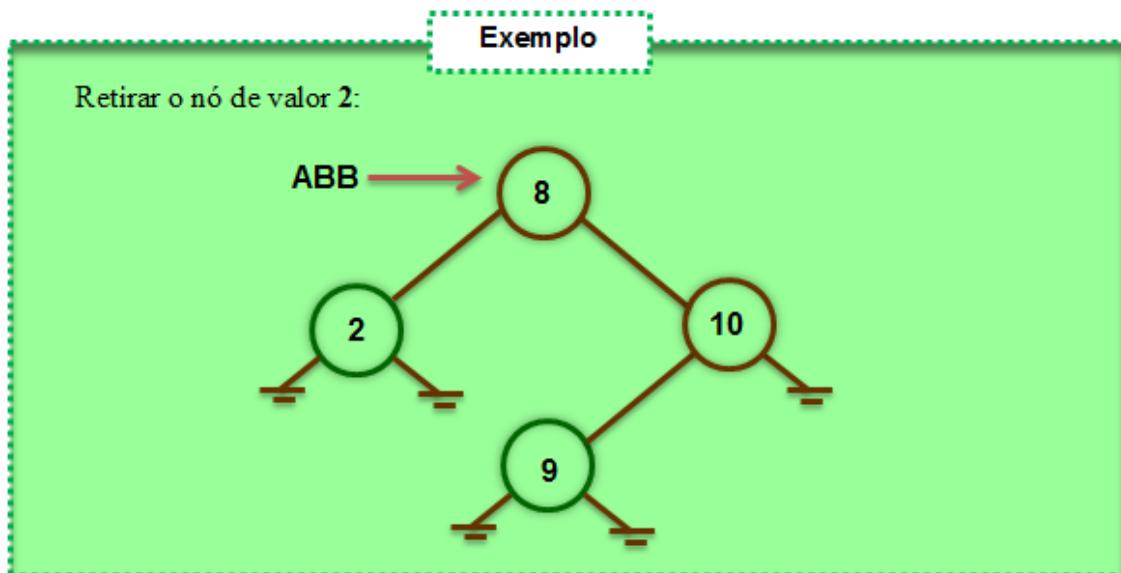
            }

        4.5     return ABB;
    }
}

```

Figura 123. Função para retirar um nó da árvore.

Para melhor entendimento, graficamente, realiza-se um exemplo a seguir, utilizando o conceito *teste de mesa*.



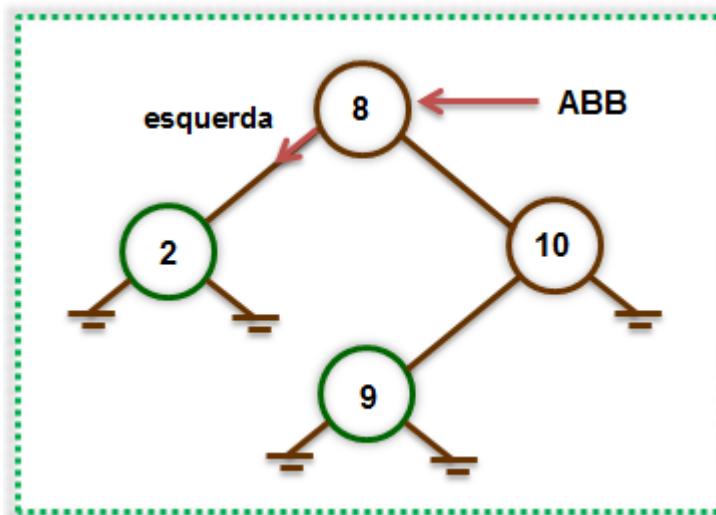
**1º PASSO:** Entra na condicional if. A função é definida por:

`arv_retirar(Arv_BBusca* ABB, 2) = arv_retirar(ABB, <nó de valor 8>)`

- O ponteiro ABB aponta para NULL? Não!! Pois o ponteiro ABB, aponta para a raiz, a qual possui valor 8, ou seja, diferente de NULL.

**2º PASSO:** Entra na primeira condicional else. O valor do nó apontado pelo ponteiro ABB, o qual é 8, é maior que 2? SIM!!

**3º PASSO:** O ponteiro esquerda, do nó apontado pelo ponteiro ABB, recebe o retorno da chamada recursiva da função `arv_retirar(ABB -> esquerda, valor)`.



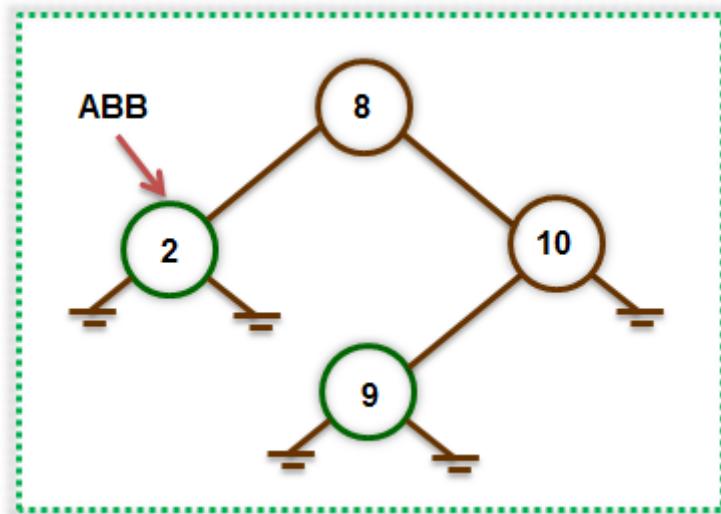
No parâmetro  $ABB = ABB \rightarrow esquerda$ :

`arv_retirar(Arv_BBusca* ABB, 2)`

$\overbrace{\quad\quad\quad}$   
`arv_retirar(ABB -> esquerda, 2)`

Assim, a função é definida por:

`arv_retirar(Arv_BBusca* ABB, 2) = arv_retirar(ABB, <nó de valor 2>)`



**4º PASSO:** Entra na condicional if.

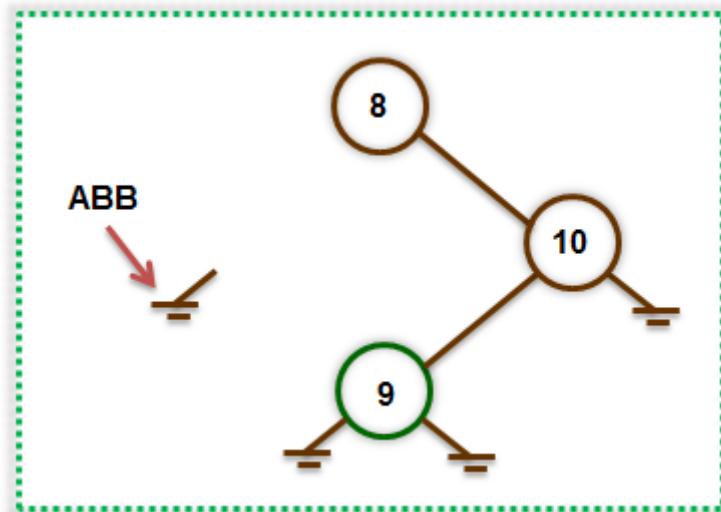
- O ponteiro ABB aponta para NULL? Não!! Pois o ponteiro ABB, aponta para a raiz, a qual possui valor 2, ou seja, diferente de NULL.

**5º PASSO:** Entra na primeira condicional else. O valor do nó apontado pelo ponteiro ABB, o qual é 2, é maior que 2? NÃO!!

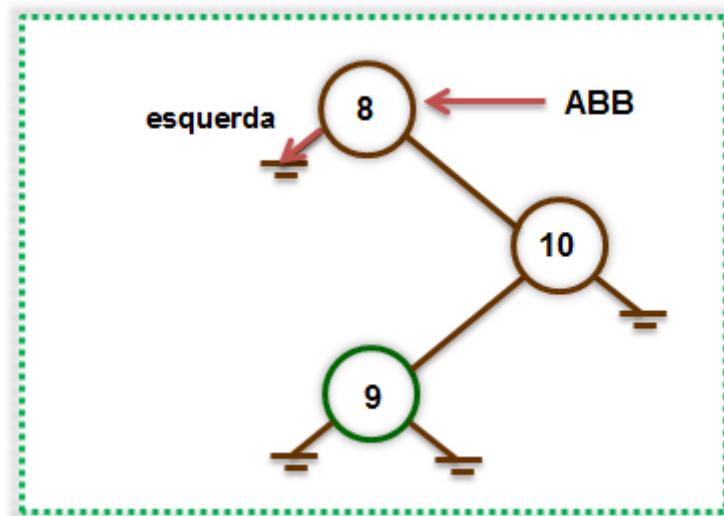
**6º PASSO:** Entra na segunda condicional else. O valor do nó apontado pelo ponteiro ABB, o qual é 2, é menor que 2? NÃO!!

**7º PASSO:** Entra na terceira condicional else.

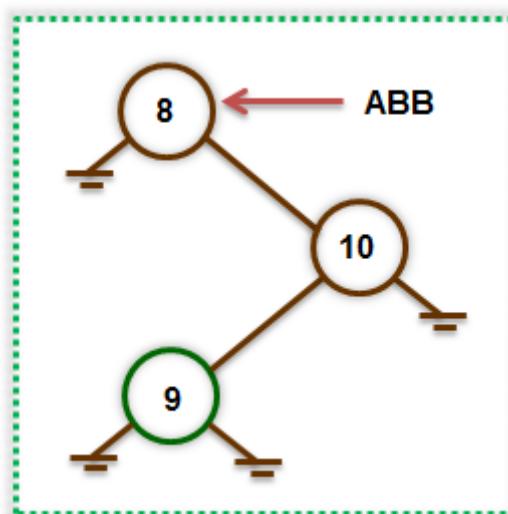
- O ponteiro esquerda e direita do nó apontado pelo ponteiro ABB apontam para NULL? SIM!!
- Desaloca-se o nó de valor 2 aponta pelo ponteiro ABB.



- O ponteiro ABB aponta para NULL.
- Retorna o nó que o ponteiro ABB aponta, no caso, para o valor nulo.
- No passo 3, o ponteiro esquerda do nó apontado pelo ponteiro ABB aguarda o retorno da função o que é NULL. Assim, ABB -> esquerda aponta para NULL.



**8º PASSO:** Retorna árvore alterada.



### 4.3. Árvore AVL

#### 4.3.2. Conceitos

Uma Árvore AVL é caracterizada por **três** fatores:

- Não há elementos duplicados.
- Ordenada.
- Balanceada.

#### 4.3.2.7. Fator de balanceamento

A **altura** de uma *árvore binária* (AB) é igual à **profundidade**, ou nível máximo, de suas folhas. A eficiência da busca em árvore depende do seu balanceamento. Algoritmos de inserção e remoção em ABB **não** garantem que a árvore gerada a cada passo seja balanceada.

Uma **Árvore AVL** é uma Árvore binária de Busca na qual as alturas das duas sub-árvores de todo nó nunca diferem em mais de **1**. Todo nó tem fator de balanceamento igual a **1, -1 ou 0**.

O problema das árvores AVL é como manter a estrutura balanceada após operações de inserção e remoção.

O **Fator de balanceamento** do nó (**FB**) é dado por:

$$\mathbf{FB(p)} = \mathbf{altura(sae)} - \mathbf{altura(sad)}$$

#### 4.3.2.8. Tipos de rotações

A transformação que promove uma árvore balanceada é chamada de **rotação**. Esta pode ser feita à esquerda ou à direita dependendo do desbalanceamento, porém, às vezes, uma única rotação pode não ser suficiente. A tabela 1 apresenta as condições para a ocorrência da rotação simples, enquanto, a tabela 2 se refere a rotação dupla.

Tabela 1. Condições para utilizar a rotação simples.

<b>Rotação Simples =</b> FB dos nós A e B com sinais iguais	
FB do nó A positivo	FB do nó A negativo
<b>Direita</b>	<b>Esquerda</b>

Tabela 2. Condições para utilizar a rotação dupla.

<b>Rotação Dupla =</b> FB dos nós A e B com sinais diferentes	
FB do nó A positivo	FB do nó A negativo
Esquerda/direita	Direita/esquerda

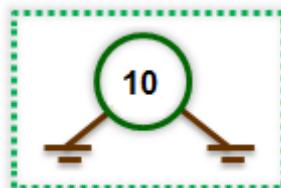
#### 4.3.2.8.1. Exemplo

Para melhor entendimento o exemplo a seguir, referente a inserção de nós e desbalanceamento, é desenvolvido passo-a-apasso.

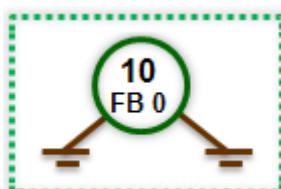
##### Exemplo

Inserir os elementos: 10, 7, 20, 15, 17, 25, 30, 5 e 1.

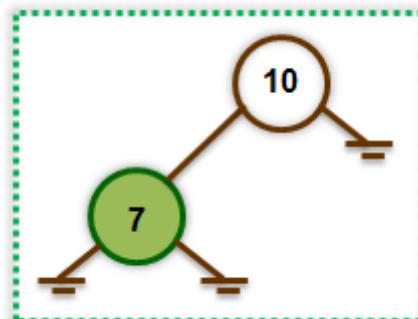
**1º PASSO:** Inserir o nó de valor 10 na árvore, o qual será a **raiz**.



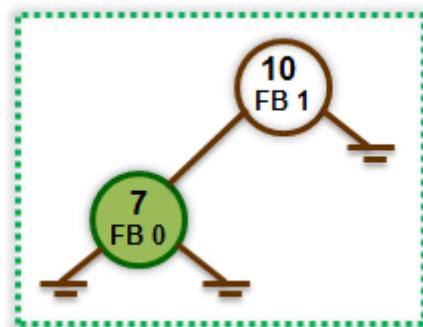
- Calculando o fator de balanceamento (FB) tem-se:  
 $\text{altura\_sae} - \text{altura\_sad} = 0 - 0 = 0$ .



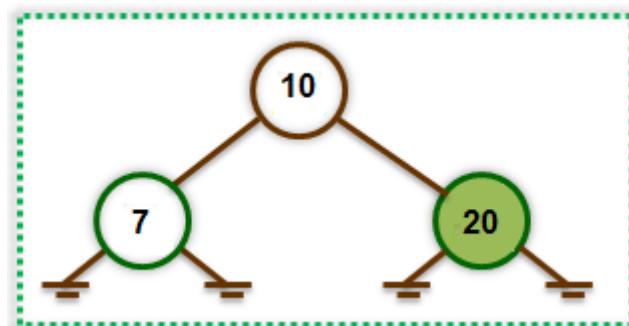
**2º PASSO:** Inserir o nó de valor 7 na árvore, o qual será filho do nó de valor 10 e colocado na **sub-árvore esquerda**, pois 7 é menor que 10.



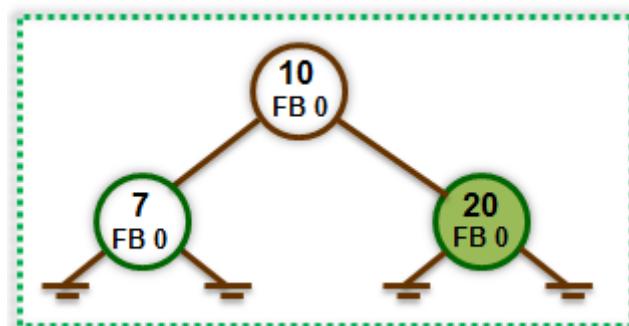
- Calculando o fator de balanceamento (FB) do:
  - Nó de valor **7** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 0 - 0 = \mathbf{0}$ .
  - Nó de valor **10** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 1 - 0 = \mathbf{1}$ .



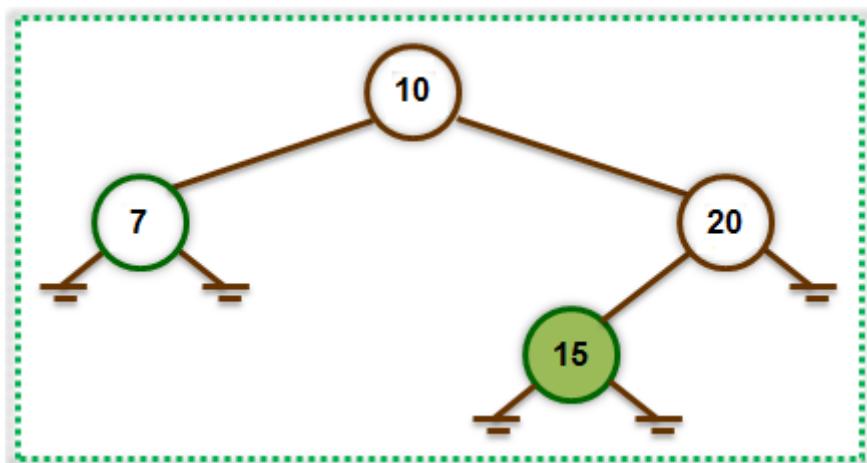
**3º PASSO:** Inserir o nó de valor **20** na árvore, o qual será filho do nó de valor 10 e colocado na **sub-árvore direita**, pois 20 é maior que 10.



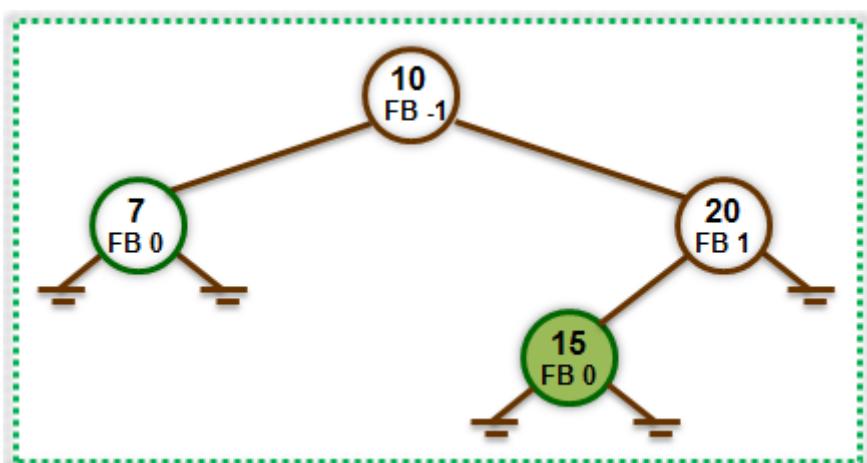
- Calculando o fator de balanceamento (FB) do:
  - Nó de valor **20** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 0 - 0 = \mathbf{0}$ .
  - Nó de valor **7** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 0 - 0 = \mathbf{0}$ .
  - Nó de valor **10** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 1 - 1 = \mathbf{0}$ .



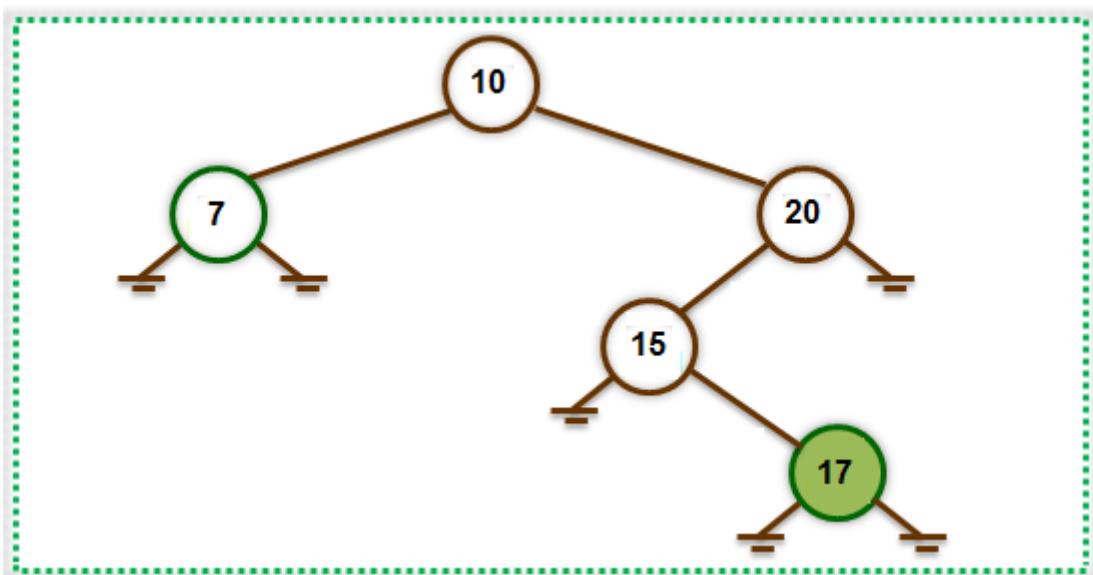
**4º PASSO:** Inserir o nó de valor **15** na árvore, o qual será filho do nó de valor 20, pois 15 é maior que 10 e menor que 20, assim, colocado na **sub-árvore esquerda**.



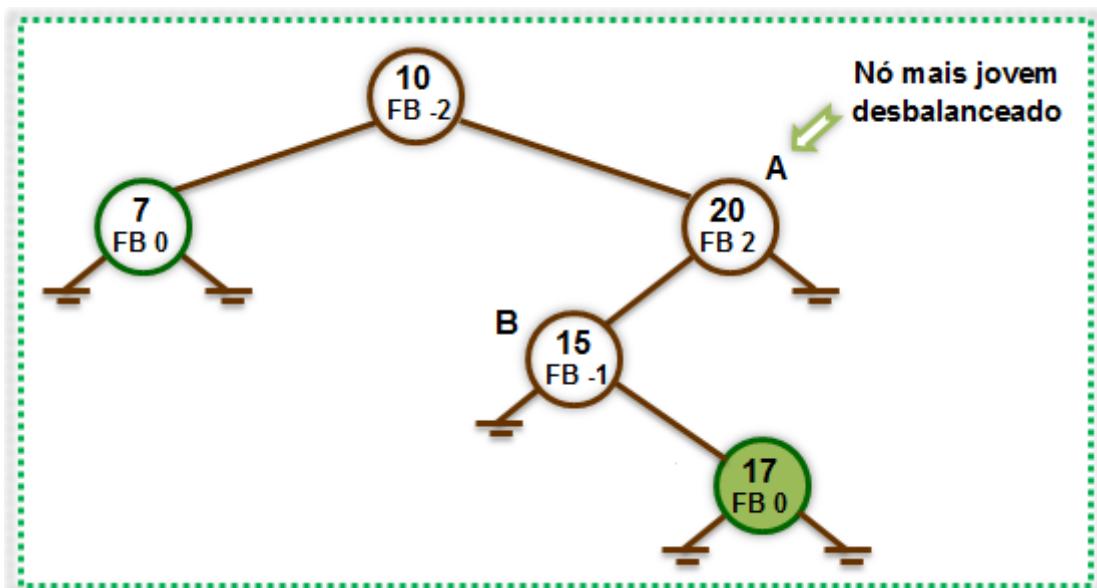
- Calculando o fator de balanceamento (FB) do:
  - Nó de valor **15** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 0 - 0 = \mathbf{0}$ .
  - Nó de valor **20** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 1 - 0 = \mathbf{1}$ .
  - Nó de valor **7** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 0 - 0 = \mathbf{0}$ .
  - Nó de valor **10** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 1 - 2 = \mathbf{-1}$ .



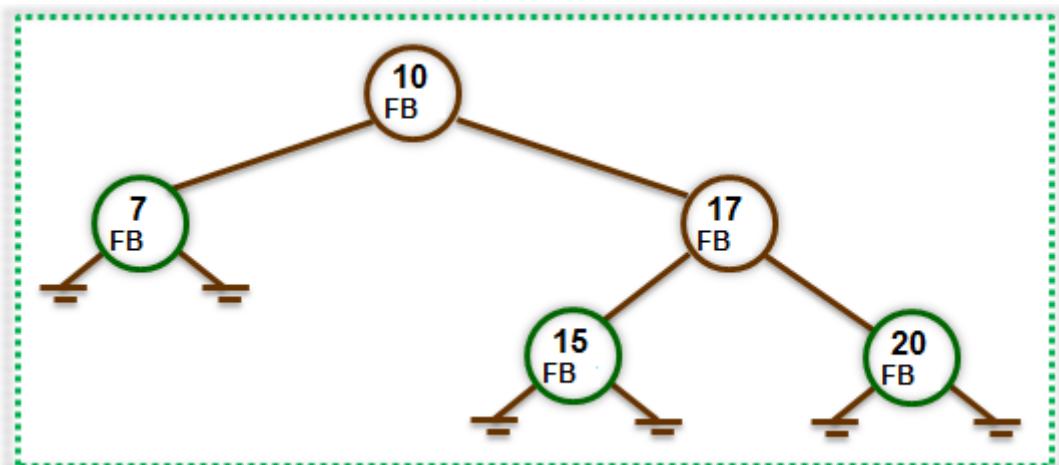
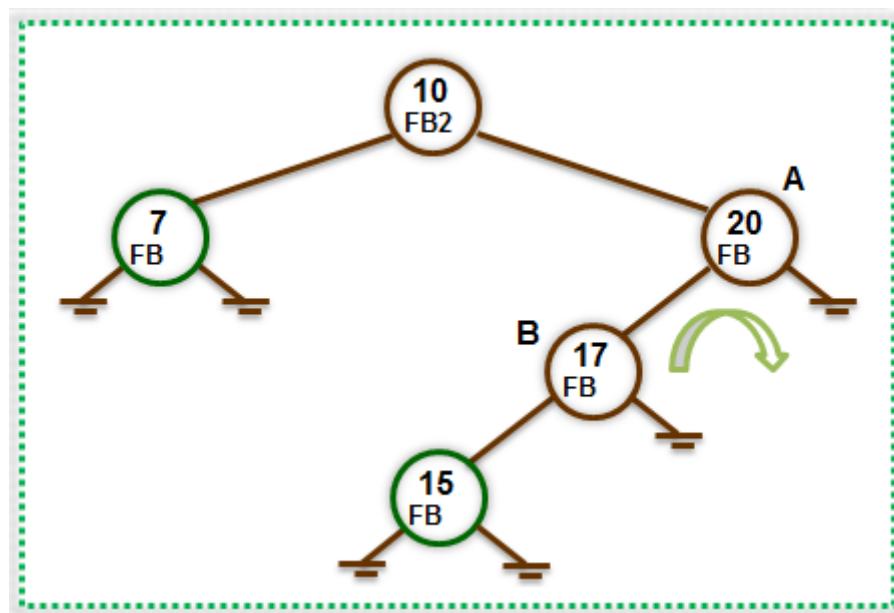
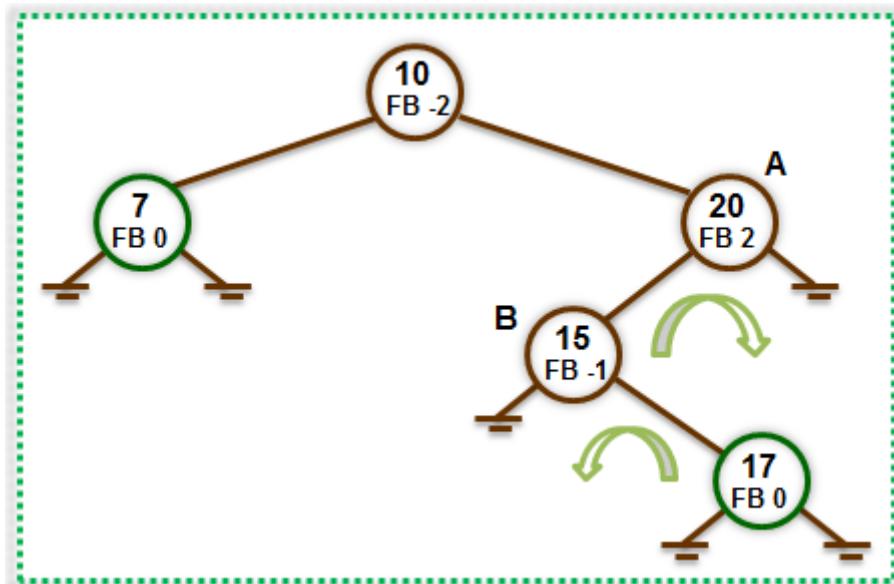
**5º PASSO:** Inserir o nó de valor **17** na árvore, o qual será filho do nó de valor 15, pois 17 é maior que 10, menor que 20 e maior que 15, assim, colocado na **sub-árvore direita**.



- Calculando o fator de balanceamento (FB) do:
  - Nó de valor **17** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 0 - 0 = \mathbf{0}$ .
  - Nó de valor **15** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 0 - 1 = \mathbf{-1}$ .
  - Nó de valor **20** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 2 - 0 = \mathbf{2}$ .
  - Nó de valor **7** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 0 - 0 = \mathbf{0}$ .
  - Nó de valor **10** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 1 - 3 = \mathbf{-2}$ .

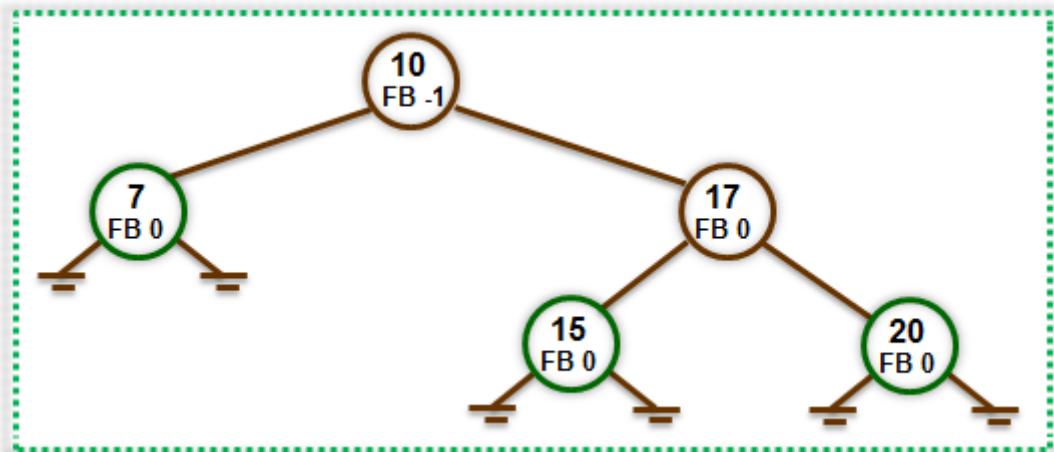


Os sinais referentes ao FB do nó B (-2) e do nó A (2) são **diferentes**, significa que deverá ser aplicada a **rotação dupla**. Como o sinal referente a A (2) é positivo a rotação será **esquerda/direita**.

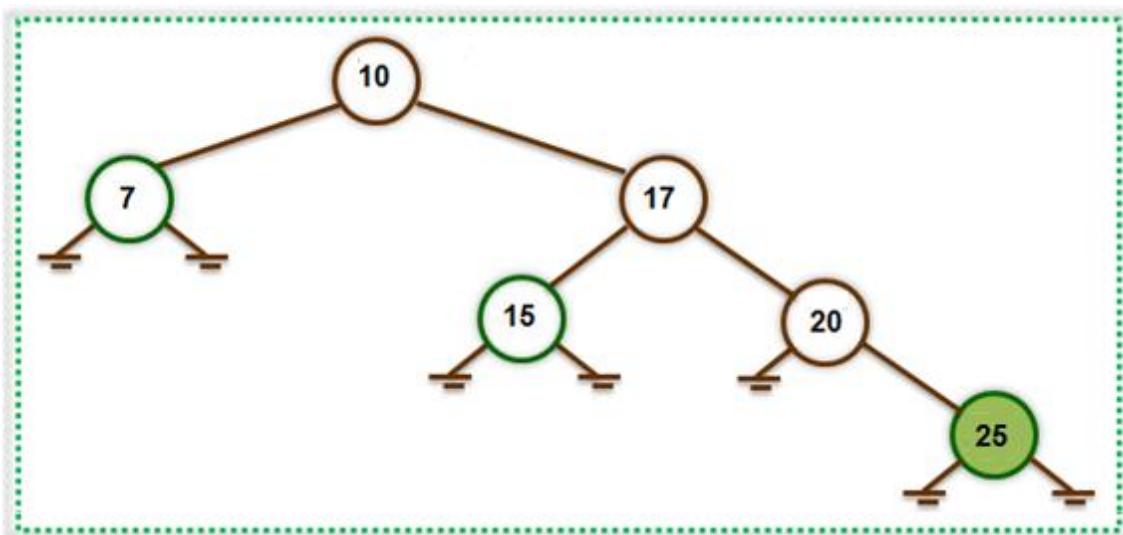


- Calculando o fator de平衡amento (FB) do:
  - Nó de valor **17** tem-se  $\text{altura\_sae} - \text{altura\_sad} = 1 - 1 = 0$ .

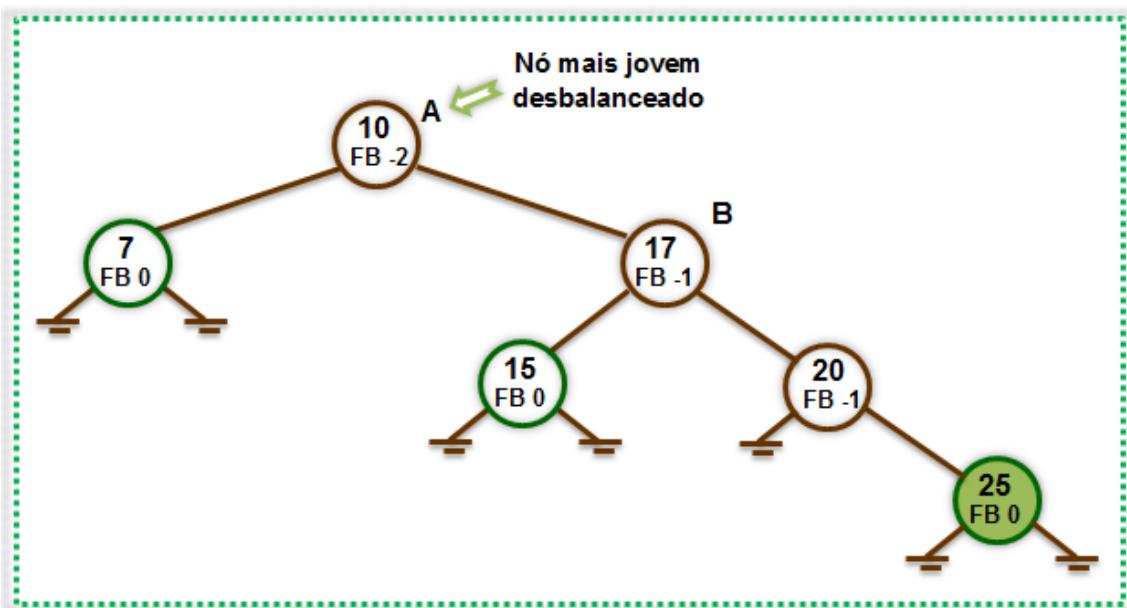
- Nó de valor **15** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 0 - 0 = \mathbf{0}$ .
- Nó de valor **20** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 0 - 0 = \mathbf{0}$ .
- Nó de valor **7** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 0 - 0 = \mathbf{0}$ .
- Nó de valor **10** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 1 - 2 = \mathbf{-1}$ .



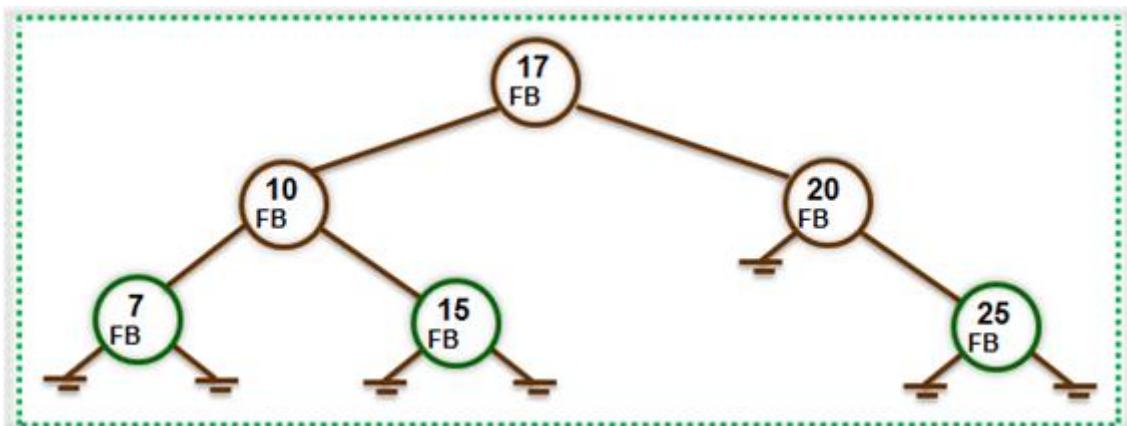
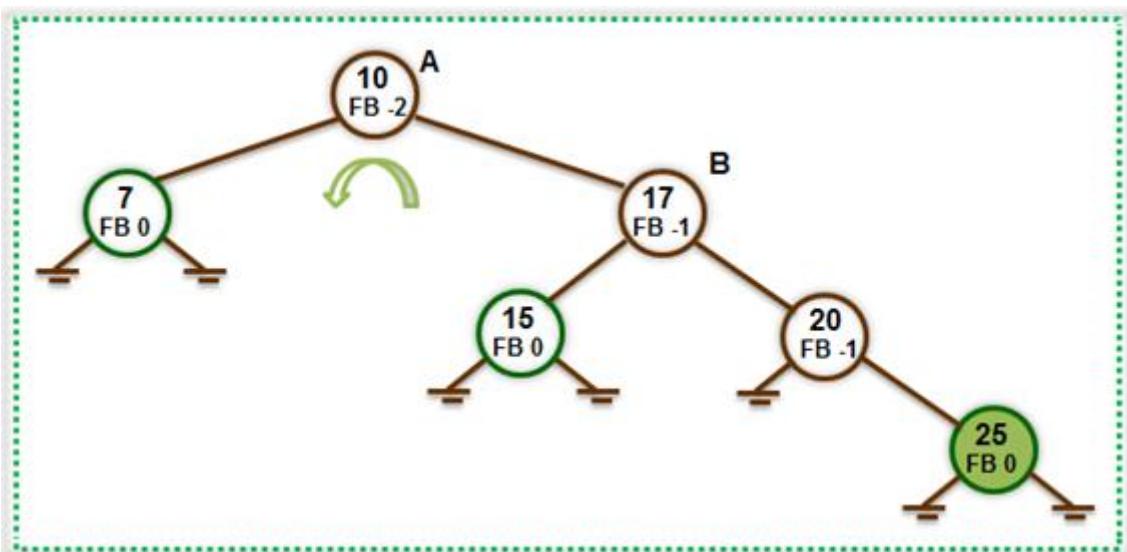
**6º PASSO:** Inserir o nó de valor **25** na árvore, o qual será filho do nó de valor 20, pois 25 é maior que 10, 17, e 20, assim, colocado na **sub-árvore direita**.



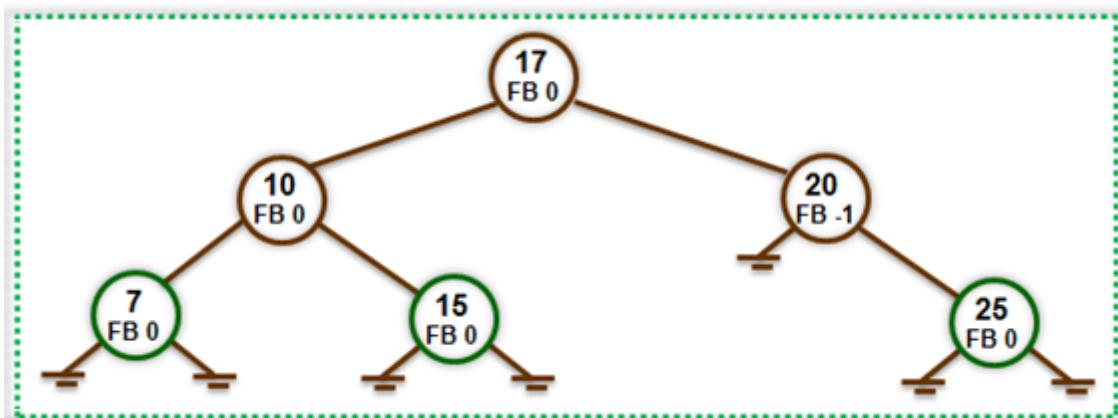
- Calculando o fator de平衡amento (FB) do:
  - Nó de valor **25** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 0 - 0 = \mathbf{0}$ .
  - Nó de valor **17** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 1 - 2 = \mathbf{-1}$ .
  - Nó de valor **15** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 0 - 0 = \mathbf{0}$ .
  - Nó de valor **20** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 0 - 1 = \mathbf{-1}$ .
  - Nó de valor **7** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 0 - 0 = \mathbf{0}$ .
  - Nó de valor **10** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 1 - 3 = \mathbf{-2}$ .



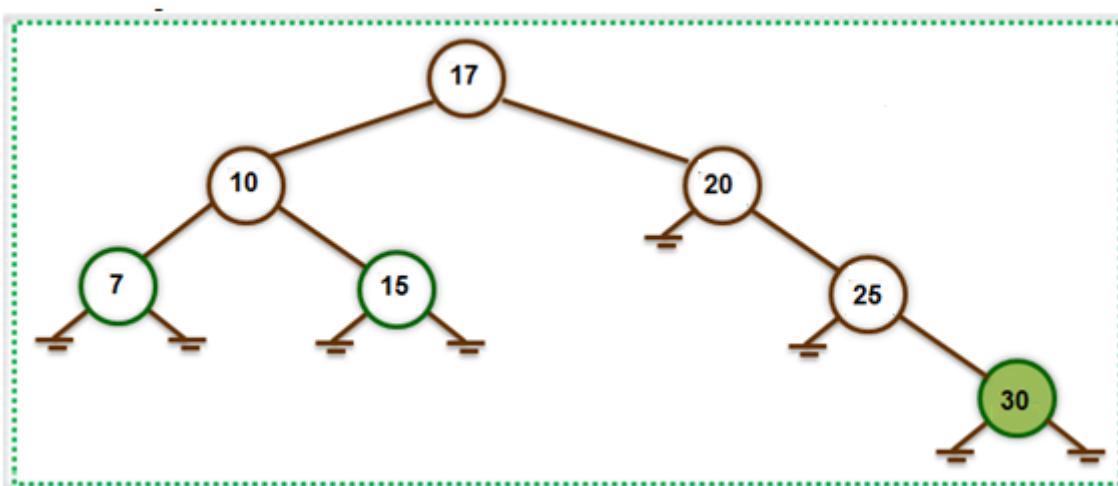
Os sinais referentes ao FB do nó B (-1) e do nó A (-2) são **iguais**, significa que deverá ser aplicada a **rotação simples**. Como o sinal referente a A (-2) é negativo a rotação será **esquerda**.



- Calculando o fator de balanceamento (FB) do:
  - Nó de valor **25** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 0 - 0 = \mathbf{0}$ .
  - Nó de valor **17** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 2 - 2 = \mathbf{0}$ .
  - Nó de valor **15** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 0 - 0 = \mathbf{0}$ .
  - Nó de valor **20** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 0 - 1 = \mathbf{-1}$ .
  - Nó de valor **7** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 0 - 0 = \mathbf{0}$ .
  - Nó de valor **10** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 1 - 1 = \mathbf{0}$ .

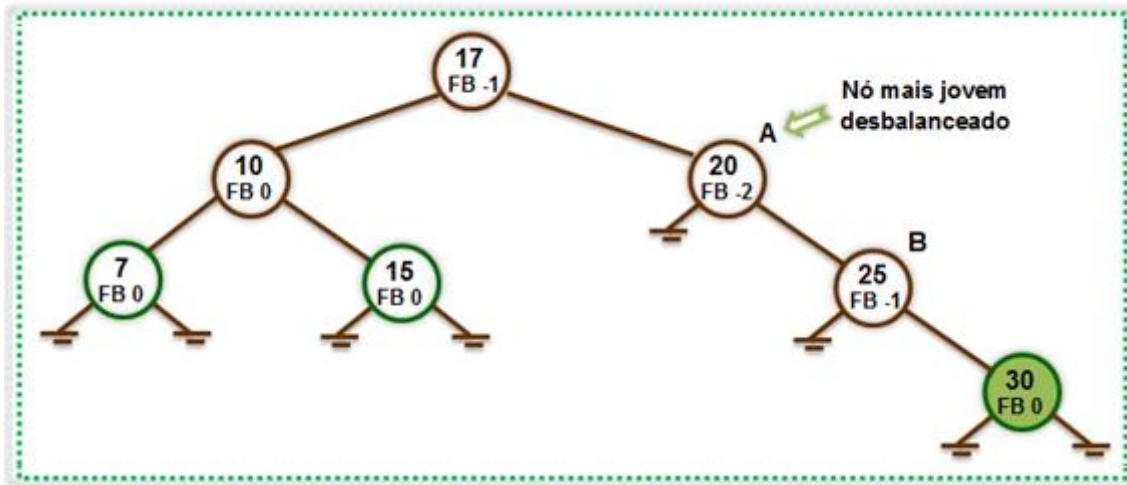


**7º PASSO:** Inserir o nó de valor **30** na árvore, o qual será filho do nó de valor 25, pois 30 é maior que 17, 20, e 25, assim, colocado na **sub-árvore direita**.

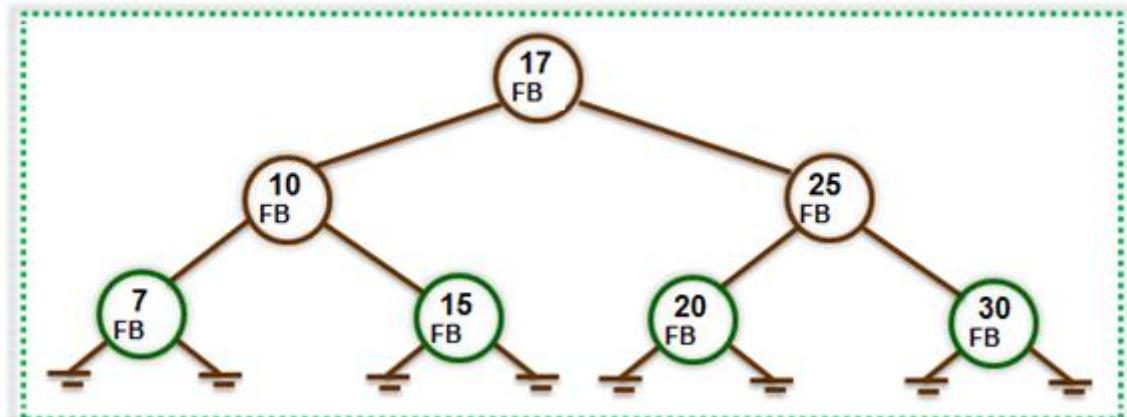
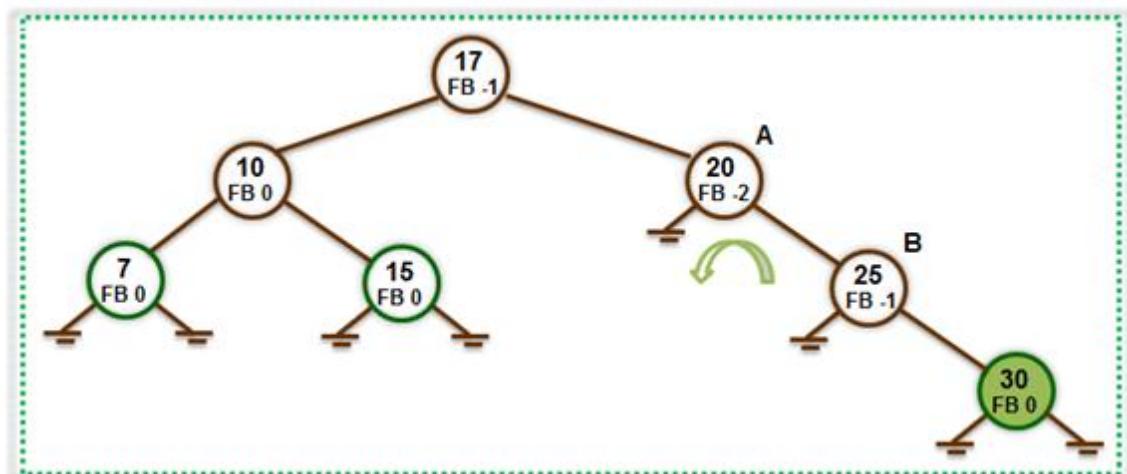


- Calculando o fator de balanceamento (FB) do:
  - Nó de valor **30** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 0 - 0 = \mathbf{0}$ .
  - Nó de valor **25** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 0 - 1 = \mathbf{-1}$ .
  - Nó de valor **17** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 2 - 3 = \mathbf{-1}$ .
  - Nó de valor **15** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 0 - 0 = \mathbf{0}$ .
  - Nó de valor **20** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 0 - 2 = \mathbf{-2}$ .

- Nó de valor **7** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 0 - 0 = 0$ .
- Nó de valor **10** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 1 - 1 = 0$ .

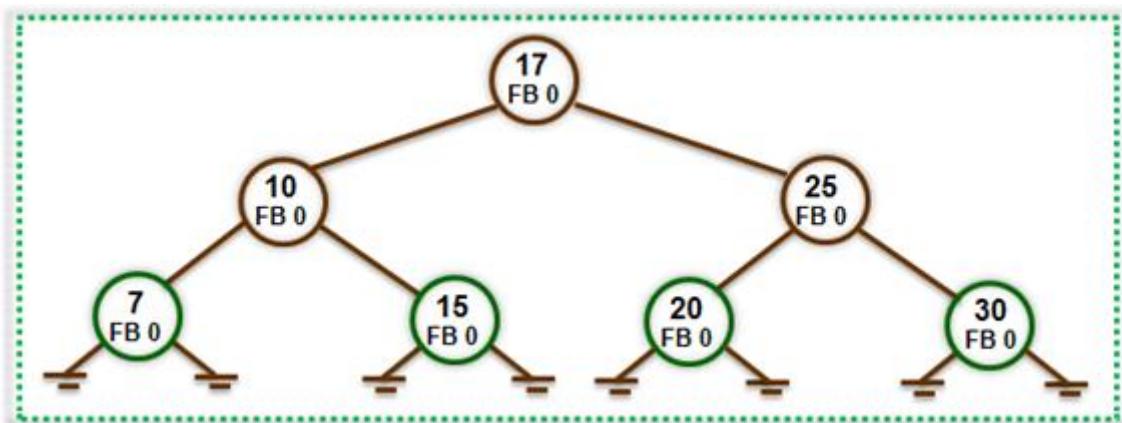


Os sinais referentes ao FB do nó B (-1) e do nó A (-2) são **iguais**, significa que deverá ser aplicada a **rotação simples**. Como o sinal referente a A (-2) é negativo a rotação será **esquerda**.

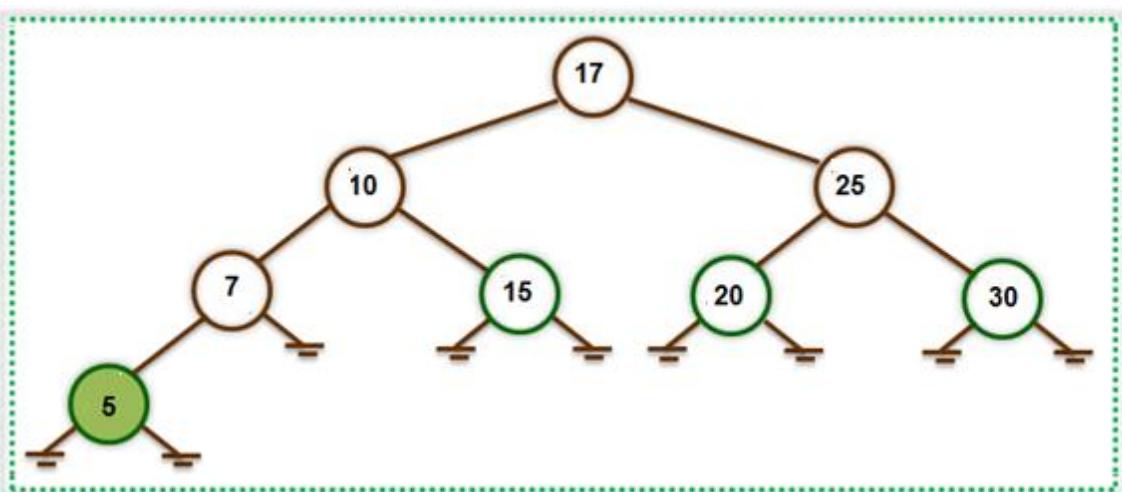


- Calculando o fator de balanceamento (FB) do:

- Nó de valor **30** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 0 - 0 = \mathbf{0}$ .
- Nó de valor **25** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 1 - 1 = \mathbf{0}$ .
- Nó de valor **17** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 2 - 2 = \mathbf{0}$ .
- Nó de valor **15** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 0 - 0 = \mathbf{0}$ .
- Nó de valor **20** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 0 - 2 = \mathbf{-2}$ .
- Nó de valor **7** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 0 - 0 = \mathbf{0}$ .
- Nó de valor **10** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 1 - 1 = \mathbf{0}$ .

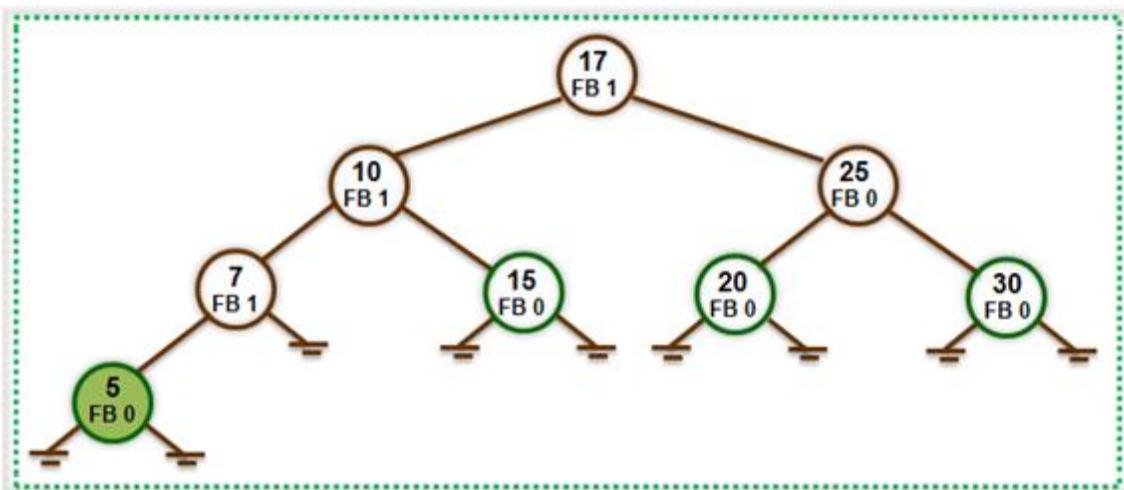


**8º PASSO:** Inserir o nó de valor **5** na árvore, o qual será filho do nó de valor 7, pois 5 é menor que 17, 10 e 7, assim, colocado na **sub-árvore esquerda**.

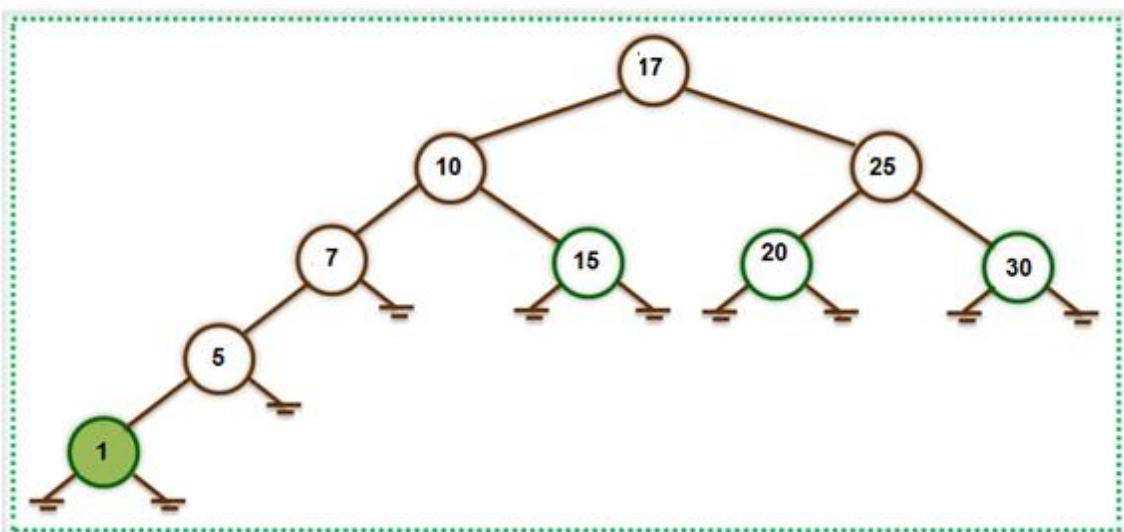


- Calculando o fator de balanceamento (FB) do:
  - Nó de valor **5** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 0 - 0 = \mathbf{0}$ .
  - Nó de valor **30** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 0 - 0 = \mathbf{0}$ .
  - Nó de valor **25** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 0 - 0 = \mathbf{0}$ .
  - Nó de valor **17** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 3 - 2 = \mathbf{1}$ .
  - Nó de valor **15** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 0 - 0 = \mathbf{0}$ .

- Nó de valor **20** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 0 - 0 = 0$ .
- Nó de valor **7** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 1 - 0 = 1$ .
- Nó de valor **10** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 2 - 1 = 1$ .

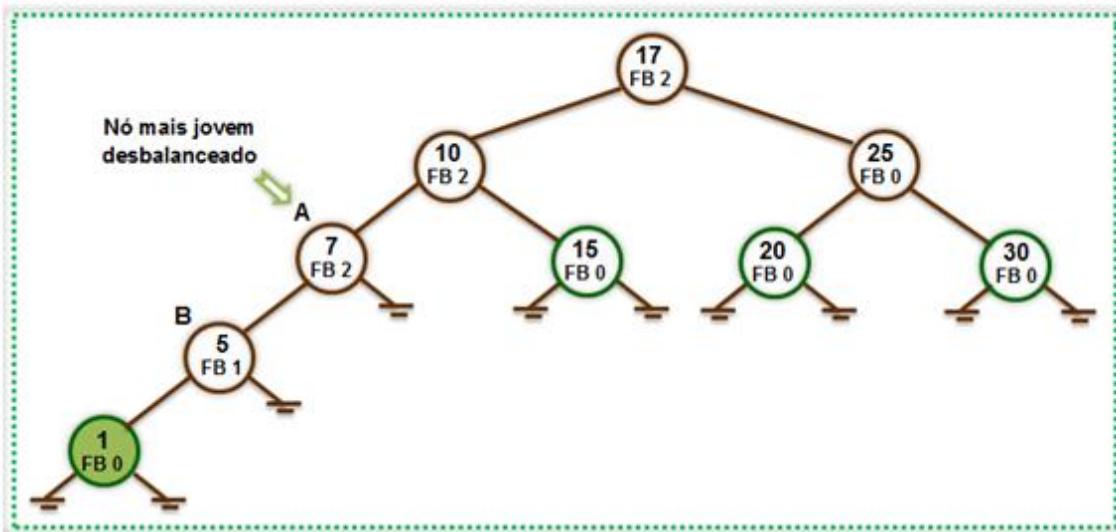


**9º PASSO:** Inserir o nó de valor **1** na árvore, o qual será filho do nó de valor 5, pois 1 é menor que 17, 10, 7 e 5, assim, colocado na **sub-árvore esquerda**.

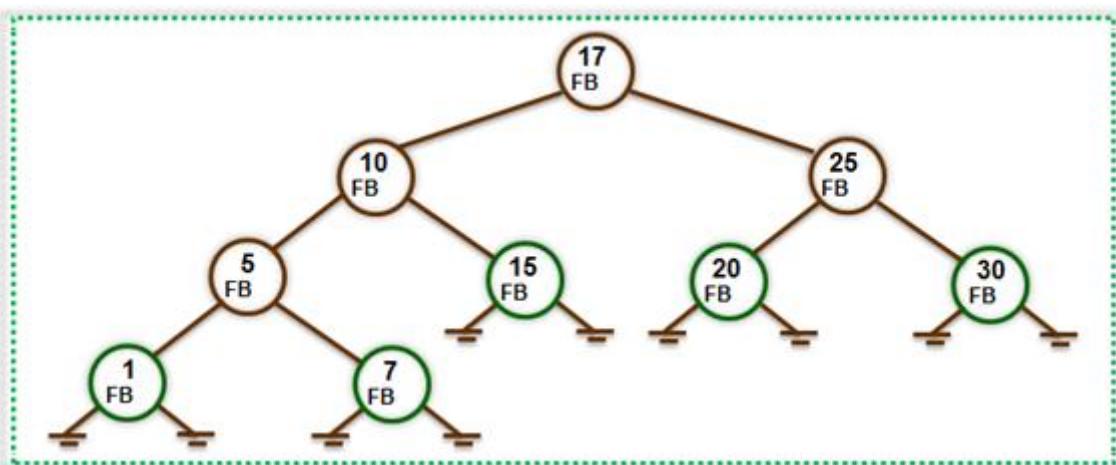
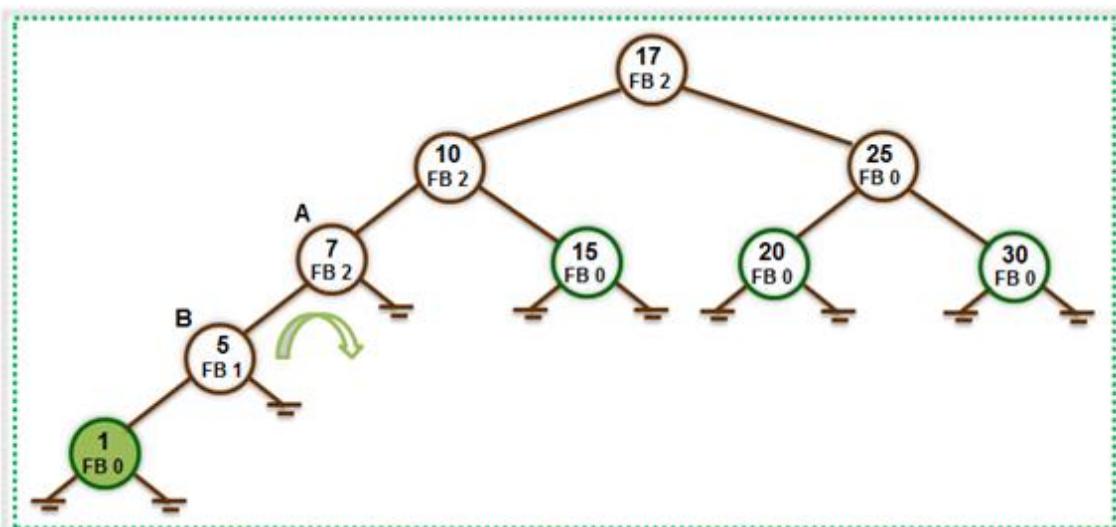


- Calculando o fator de平衡amento (FB) do:
  - Nó de valor **1** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 0 - 0 = 0$ .
  - Nó de valor **5** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 1 - 0 = 1$ .
  - Nó de valor **30** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 0 - 0 = 0$ .
  - Nó de valor **25** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 1 - 1 = 0$ .
  - Nó de valor **17** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 4 - 2 = 2$ .
  - Nó de valor **15** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 0 - 0 = 0$ .
  - Nó de valor **20** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 0 - 0 = 0$ .

- Nó de valor **7** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 1 - 0 = 1$ .
- Nó de valor **10** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 3 - 1 = 2$ .

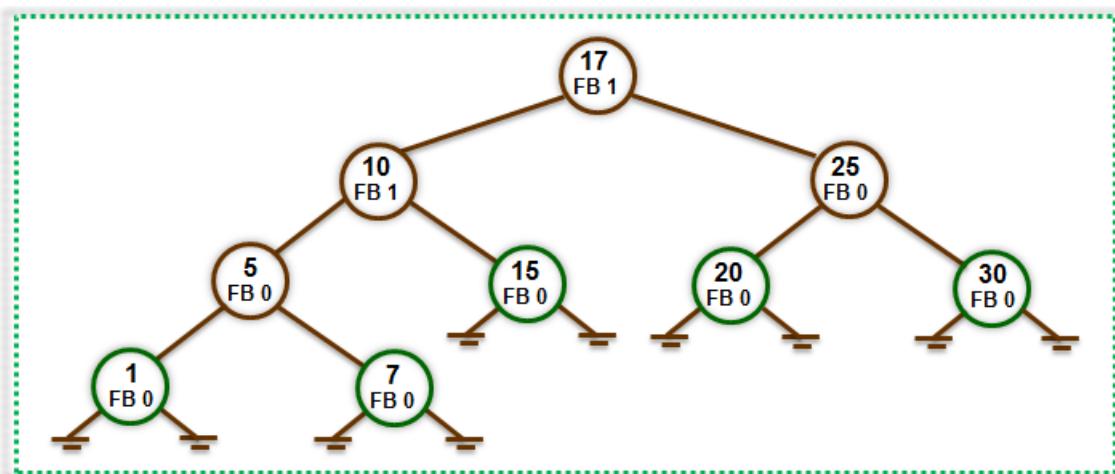


Os sinais referentes ao FB do nó B (1) e do nó A (2) são **iguais**, significa que deverá ser aplicada a **rotação simples**. Como o sinal referente a A (2) é positivo a rotação será **direita**.



- Calculando o fator de balanceamento (FB) do:

- Nó de valor **1** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 0 - 0 = \mathbf{0}$ .
- Nó de valor **5** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 1 - 1 = \mathbf{0}$ .
- Nó de valor **30** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 0 - 0 = \mathbf{0}$ .
- Nó de valor **25** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 1 - 1 = \mathbf{0}$ .
- Nó de valor **17** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 3 - 2 = \mathbf{1}$ .
- Nó de valor **15** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 0 - 0 = \mathbf{0}$ .
- Nó de valor **20** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 0 - 0 = \mathbf{0}$ .
- Nó de valor **7** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 0 - 0 = \mathbf{0}$ .
- Nó de valor **10** tem-se  $\text{altura}_{\text{sae}} - \text{altura}_{\text{sad}} = 2 - 1 = \mathbf{1}$ .



## 4.4. Árvore B

### 4.4.2. Conceitos

Uma Árvore B é caracterizada por:

- Não é necessário formar a árvore a partir de uma raiz como nas demais árvores apresentadas anteriormente. A figura 124 mostra a estrutura lógica de um elemento.

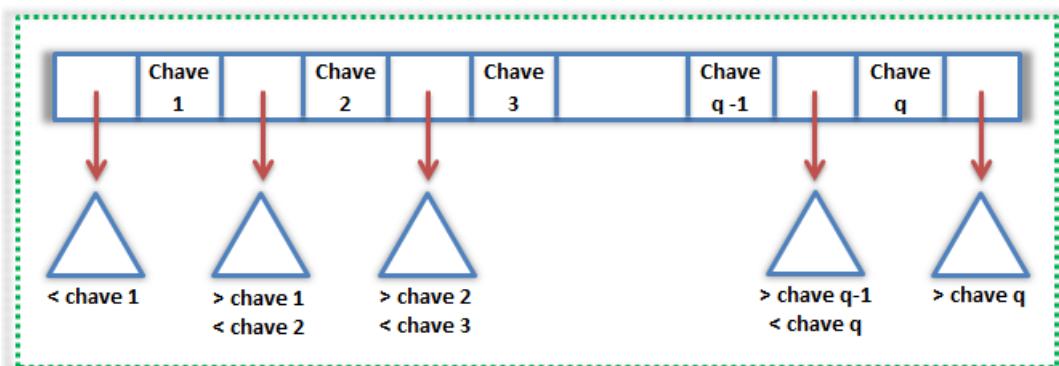


Figura 124. Estrutura lógica de um elemento.

- **Ordem.** Maior quantidade de ponteiros que pode ser armazenado em um nó. O número máximo de ponteiros é igual ao número máximo de descendentes de um elemento. **Exemplo:** Árvore B de ordem 4 possui no máximo 3 chaves e 4 ponteiros, como mostra a figura 125.

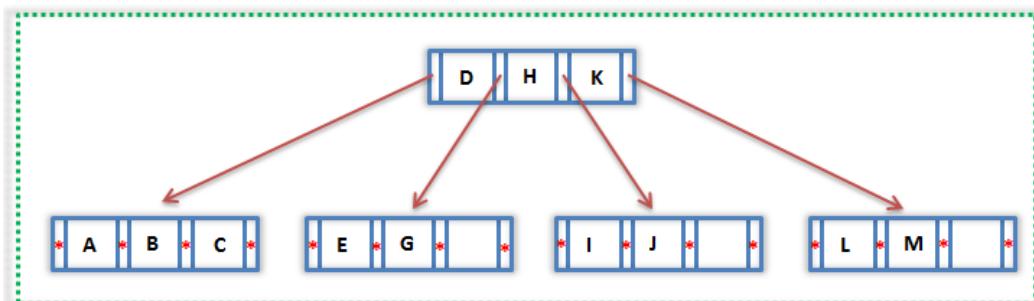


Figura 125. Exemplo de uma Árvore B de ordem 4.

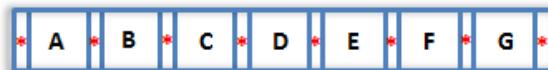
- **Botton-up** para a criação (as chaves na raiz da árvore surgem naturalmente). Assim, não há necessidade de tratamento do problema referente ao desbalanceamento utilizando algoritmos de reorganização da árvore.

### 4.4.2.7. Inserção de dados (chaves)

A inserção é sempre realizada nos elementos considerados folhas (nó sem filhos). Casos de árvore vazia e overflow no nó raiz devem ser verificados.

❖ **Caso de árvore vazia.** Para a criação e preenchimento do nó:

- Primeira chave se refere a criação do nó raiz. Exemplo:
  - Elemento com capacidade para 7 chaves, estas sendo letras do alfabeto e considerando a árvore B como vazia.
  - Chaves B C G E F D A. Matidas ordenadas no nó.
  - Ponteiros (\*). Nós folhas: fim da lista (NULL) ou -1.
  - Nó raiz é igual ao nó folha.



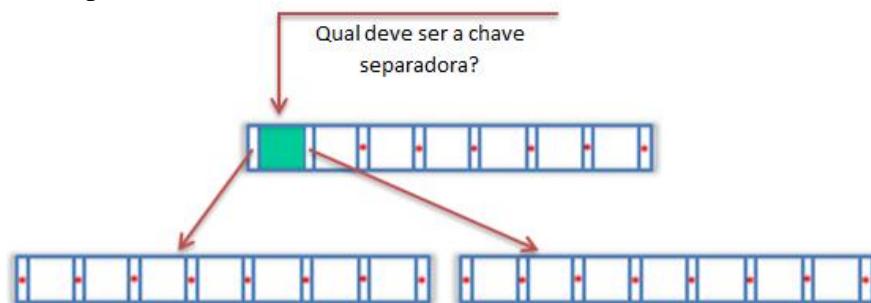
- Demais chaves se referem as inserções realizadas até a capacidade máxima do elemento.

❖ **Overflow na raiz.**

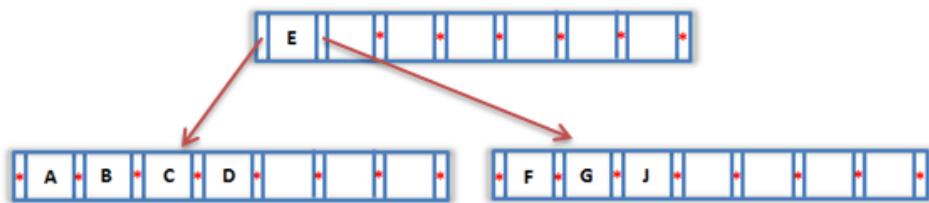
- O **primeiro** passo:
  - É partitionar o nó (split), assim, o nó original é igual a este mais o novo nó ( $nó\ original = nó\ original + novo\ nó$ ).
  - As chaves são distribuídas uniformemente nos dois nós. As chaves do nó original mais a nova chave ( $chaves\ do\ nó\ original + nova\ chave$ ).
  - Exemplo: inserção de J.



- O **segundo** passo:
  - Criação de uma nova raiz. A existência de um nível mais alto na árvore permite a escolha das folhas durante a pesquisa.
  - Exemplo:



- O **terceiro** passo:
  - Promoção de chave (*promotion*). A primeira chave do novo elemento resultante do partitionamento é promovida para o nó raiz.
  - Exemplo:



#### ❖ Inserção nos nós folhas.

- O **primeiro** passo – *Pesquisa*:
  - Percorre-se a árvore até encontrar o nó folha, na qual a nova chave será inserida.
- O **segundo** passo – *Inserção em nó com espaço*:
  - Ordenação da chave após a inserção.
  - Alteração dos valores nos campos de referência.
- O **terceiro** passo – *Inserção em nó cheio*:
  - **Particionamento:**
    - Criação de um novo nó (*nó original* = *nó original* + *novo nó*).
    - Distribuição uniforme das chaves nos dois nós.
  - **Promoção:**
    - Escolha da primeira chave do novo nó como chave separadora no elemento pai.
    - Ajusta-se o nó pai para apontar para o novo nó.
    - Propagação de overflow.

##### 4.4.2.7.1. Exemplo

Para melhor entendimento o exemplo a seguir, referente a inserção de nós, é desenvolvido passo-a-passo.

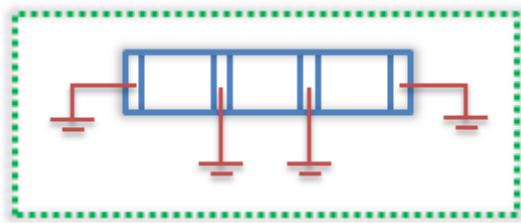
###### Exemplo

Inserir os elementos:

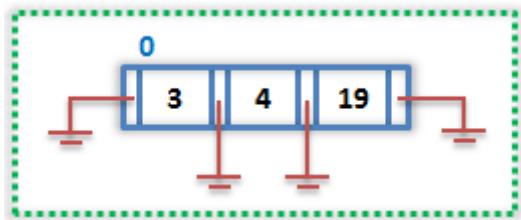
**3, 19, 4, 20, 1, 13, 16, 9, 2, 24 e 14.**

**Obs:** Construir uma árvore B de ordem .

**1º PASSO:** Uma árvore B de **ordem 4** possui **3 chaves**.

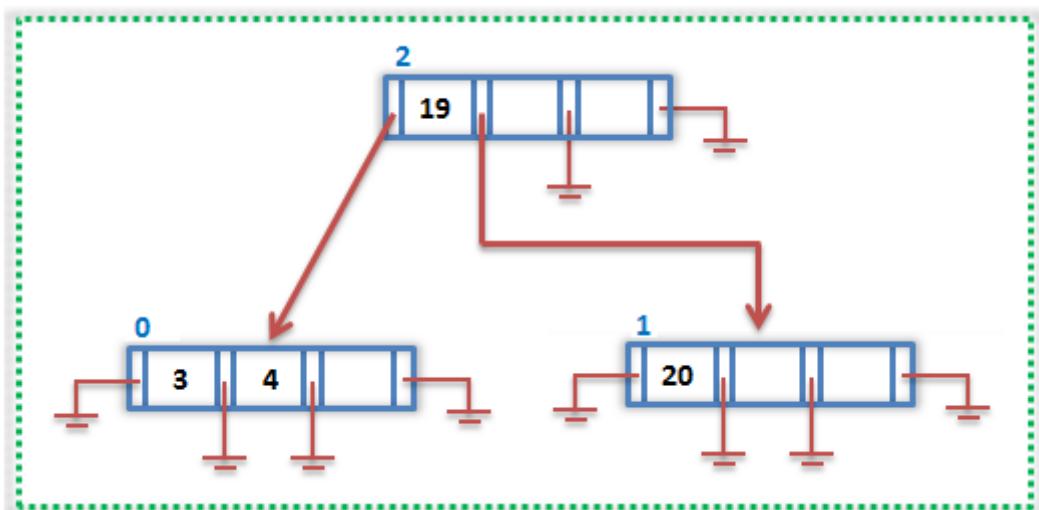


**2º PASSO:** Inserir os nós, ordenadamente, até atingir a capacidade máxima do nó, ou seja, inserir os nós 3, 4 e 19.

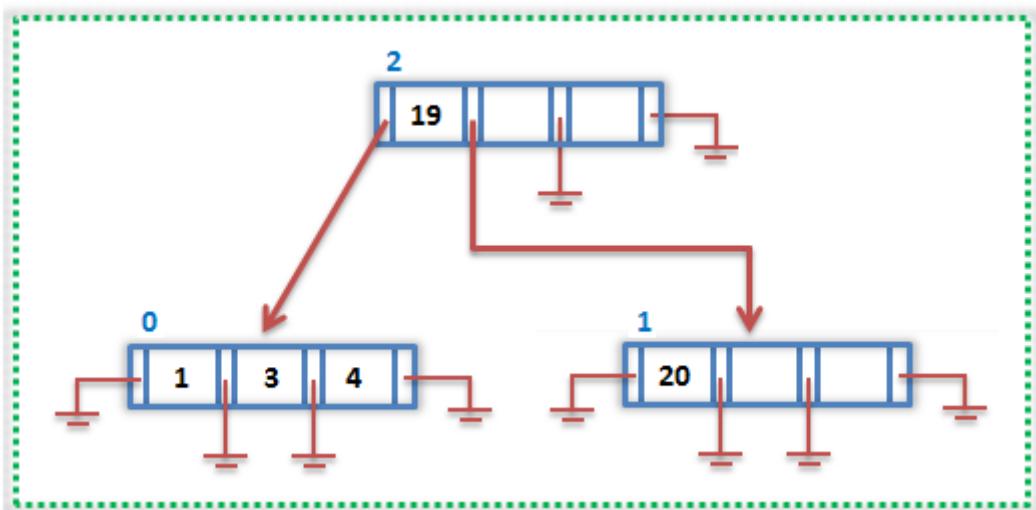


**3º PASSO:** Para inserir o 15 ficará: 3 4 19 20. Haverá:

- particionamento do nó: 3 4 | 19 20.
- Promoção do primeiro dado do novo nó criado, o qual é o 19.

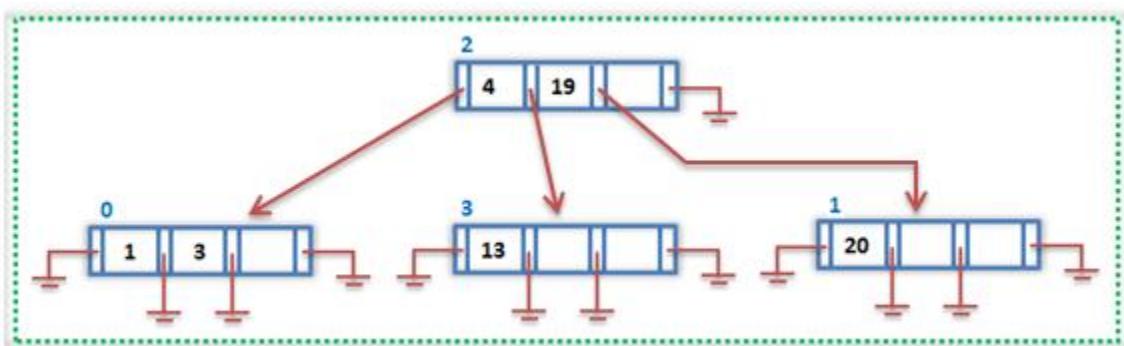


**4º PASSO:** O 1 deve ser inserido no nó que possui os valores **3 e 4**, pois 1 é menor que 19, assim, ficará. **1 3 4**.

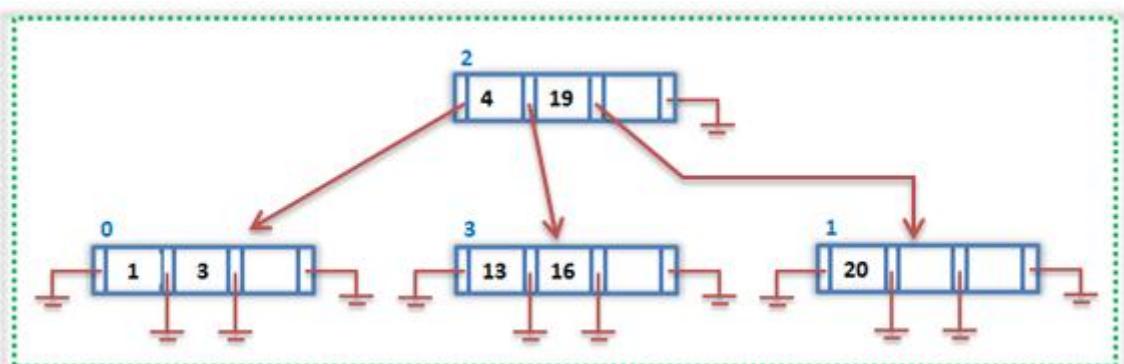


**5º PASSO:** O 13 é menor que 19, assim, ficará. **1 3 4 13**. Haverá:

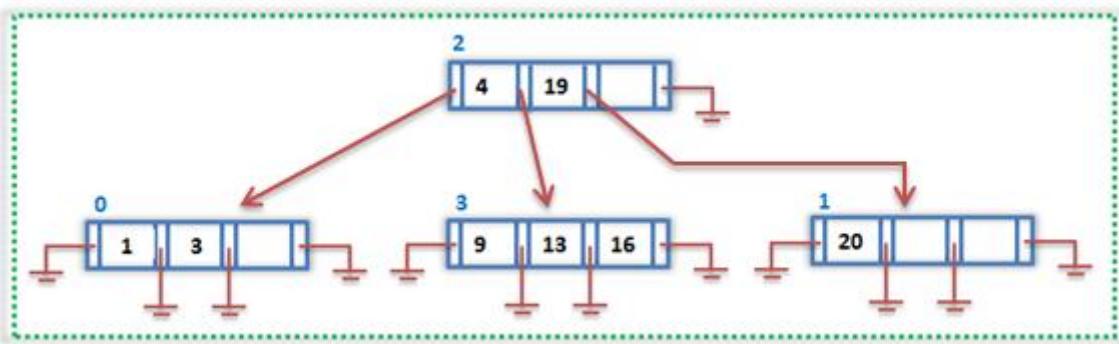
- Particionamento do nó: **1 3 | 4 13**.
- Promoção do primeiro dado do novo nó criado, o qual é o 4.



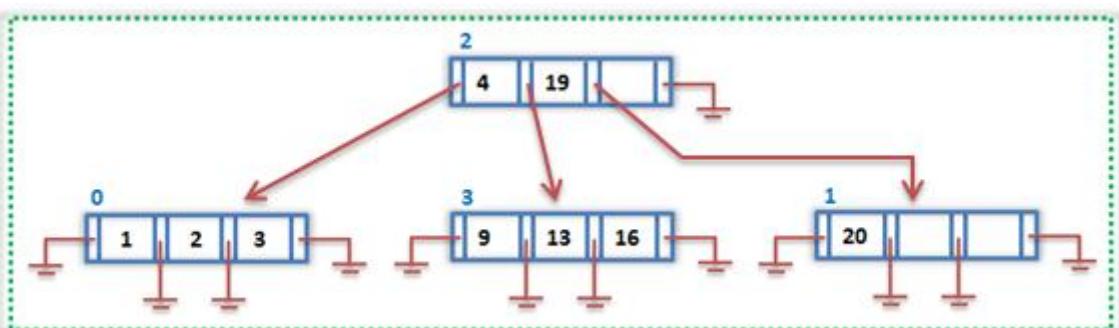
**6º PASSO:** O 16 deve ser inserido no nó que possui o valor **13**, pois 16 é maior que 4 e menor que 19, assim, ficará. **13 16**.



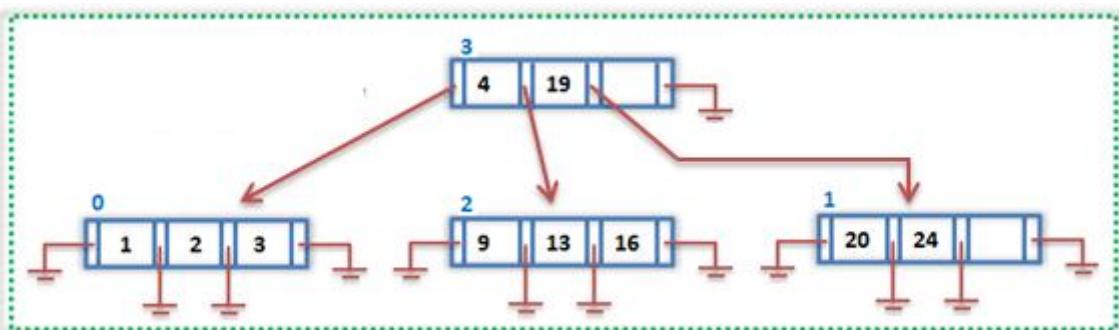
**7º PASSO:** O 9 deve ser inserido no nó que possui os valores **13** e **16**, pois 9 é maior que 4 e menor que 19, assim, ficará. **9 13 16**.



**8º PASSO:** O 2 deve ser inserido no nó que possui os valores 1 e 3, pois 2 é menor que 4, assim, ficará. **1 2 3.**

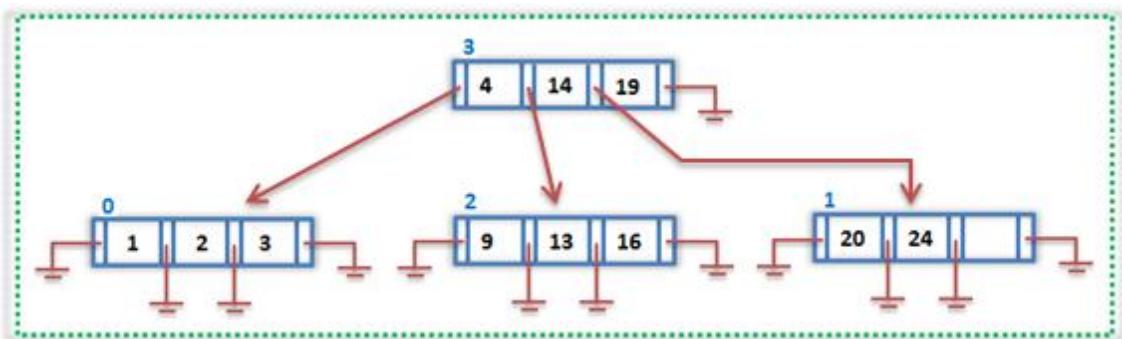


**9º PASSO:** O 24 deve ser inserido no nó que possui o valor 20, pois 24 é maior que 4 e 19, assim, ficará. **20 24.**



**10º PASSO:** O 14 é maior que 4 e menor que 19, assim, ficará. **9 13 14 16.** Haverá:

- Particionamento do nó: 9 13 | 14 16.
- Promoção do primeiro dado do novo nó criado, o qual é o 14.



## 4.5. Exercícios

- 1) Criação de algoritmos recursivos, considerando uma árvore binária, para determinar:
- A quantidade de nós que possui.
  - A soma dos valoresm contidos em todos os nós.
  - A profundidade.

- 3) Suponha que as notas de uma prova de uma turma de alunos estejam armazenadas numa “árvore binária de busca”, ordenada pelos números de matrícula dos alunos. A estrutura desta árvore é dada a seguir:

```
typedef struct aluno Aluno
struct aluno{
    int matricula;
    float nota;
    Aluno * esquerda, *direta;
};
```

Escreva uma função para dado o número de matrícula retornar a nota do aluno respectivo.

- 4) Considere uma **árvore binária de busca** que armazena os dados dos alunos de uma turma, usando a média de cada um como critério de ordenação. O tipo que representa um nó da árvore é dado por:

```
struct arv_bb{
    char nome[60];
    float media;
    struct arv_bb* esquerda, *direita;
};
typedef struct arv_bb arv_Bbusca;
```

Escreva uma função que receba como parâmetros uma árvore, esta definida pela estrutura acima, e os limites de um intervalo fechado de notas, e retorne quantos alunos têm médias dentro deste intervalo. O protótipo dessa função é dado por:

```
int arv_retira(Arv_BBusca* ABB, float menor_nota, float maior_nota);
```

- 5) Implemente uma função para verificar se uma **árvore binária** é uma **árvore binária de busca**. Se a árvore binária verificada não for uma árvore binária de busca, construa uma nova árvore binária, sendo que esta deverá ser binária de busca. Para isso, a construção dessa árvore binária de busca deverá ser realizada considerando o **percurso ordem simétrica** da árvore binária existente.

6) Considerando as seguintes declarações de uma árvore binária, implemente uma função que:

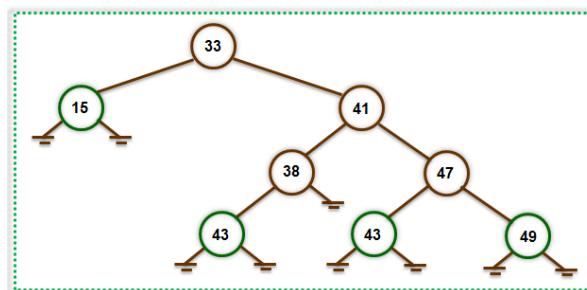
a) Retorne a quantidade de nós que possuem **apenas um filho** dessa árvore. O protótipo do função deverá ser:

```
int unico_filho(Arv* AB);
```

b) Retorne a quantidade de nós que **não** são folhas, isto é, nós que possuem pelo menos um filho. Essa função deve obedecer o protótipo:

```
int nos_intermediarios(Arv* ABB);
```

7) Dado a árvore binária de busca a seguir, desenvolver um programa que faça:



- a) Descrever as três ordens de percurso.
- b) Inserir o nó de valor 21.
- c) Remover o nó de valor 41.

8) Construa uma Árvore AVL, simule graficamente as seguintes inserções:

b z a y c x d w f v g

9) Construa uma Árvore B de ordem 4, simule graficamente as seguintes inserções:

a c g x n u s e b p i j

## Referências

CELES, Waldemar; CERQUEIRA, Renato; RANGEL, José Lucas. **Introdução a estruturas de dados**. Editora Campus, 2004.