

Árvores Binárias de Busca

Prof. Luiz Gustavo Almeida Martins

Árvores binárias de busca (ABB)

Árvore binária ordenada na qual os dados são distribuídos pelos nós para **facilitar a pesquisa**

Estrutura mais adequada para a **busca binária**

Inserção e remoção é + eficiente que em estruturas lineares

Árvores binárias de busca (ABB)

Árvore binária ordenada na qual os dados são distribuídos pelos nós para **facilitar a pesquisa**

Estrutura mais adequada para a **busca binária**

Inserção e remoção é + eficiente que em estruturas lineares

Distribuição dos dados deve obedecer o critério de ordenação (**ex:** ordem crescente):

Elementos à esquerda (SAE) < raiz

Elementos à direita (SAD) > raiz

Árvores binárias de busca (ABB)

Árvore binária ordenada na qual os dados são distribuídos pelos nós para **facilitar a pesquisa**

Estrutura mais adequada para a **busca binária**

Inserção e remoção é + eficiente que em estruturas lineares

Distribuição dos dados deve obedecer o critério de ordenação (**ex:** ordem crescente):

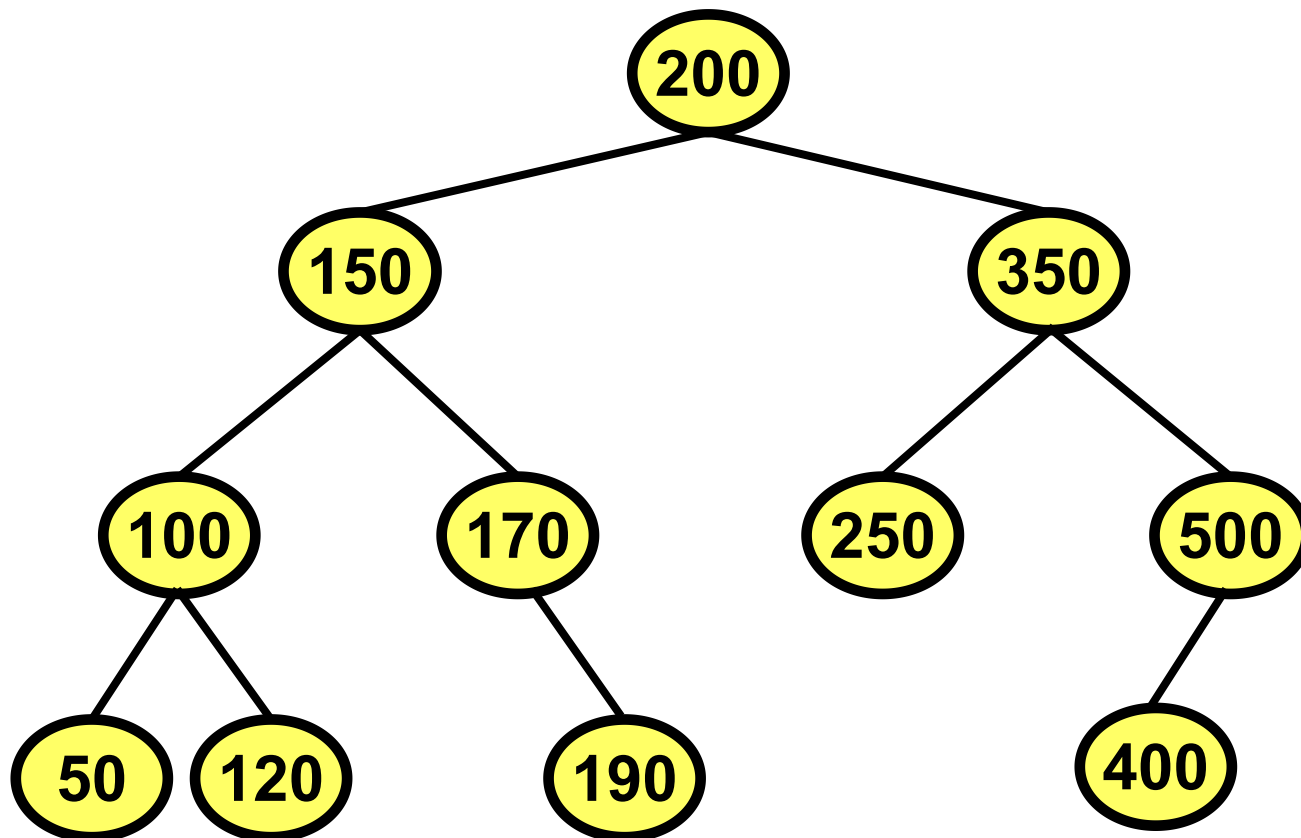
Elementos à esquerda (SAE) < raiz

Elementos à direita (SAD) > raiz

Eficiência está relacionada com a **altura da árvore**

Exemplo de árvore binária de busca

ABB representam uma **relação de ordem** definida pela **chave** existente no **campo informação** de cada nó



Especificação do TAD ABB

Cabeçalho:

Dados: registros de um dicionário (chave + outros campos)

Lista de operações:

cria_arvore: retorna uma ABB vazia

arvore_vazia: verifica se a ABB está vazia

insere_ord: insere um novo nó na ABB de modo a garantir a ordenação

remove_ord: remove um elemento da ABB, preservando a ordenação

busca_bin: busca um elemento na ABB, aproveitando-se da ordenação

exibe_arvore: percorre a ABB e imprime cada nó

exibe_ordenado: percorre a ABB e imprime os nós de forma ordenada

libera_arvore: libera todo o espaço de memória alocado pela ABB

Especificação do TAD ABB

Cabeçalho:

Dados: registros de um dicionário (chave + outros campos)

Lista de operações:

cria_arvore: retorna uma ABB vazia

arvore_vazia: verifica se a ABB está vazia

insere_ord: insere um novo nó na ABB de modo a garantir a **ordenação**

remove_ord: remove um elemento da ABB, preservando a **ordenação**

busca_bin: busca um elemento na ABB, aproveitando-se da **ordenação**

exibe_arvore: percorre a ABB e imprime cada nó

exibe_ordenado: percorre a ABB e imprime os nós de forma **ordenada**

libera_arvore: libera todo o espaço de memória alocado pela ABB

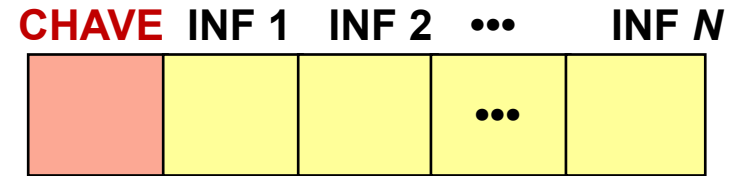
Várias operações vistas para a árvore binária continuam sendo válidas para a ABB, exceto **aquelas que são afetadas pela ordenação**

ABB: estrutura de representação

Estrutura do **dicionário**:

Campo **CHAVE**

Demais campos de informação



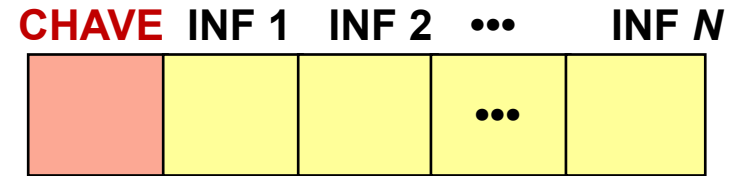
Estrutura Dicionário (DIC)

ABB: estrutura de representação

Estrutura do **dicionário**:

Campo **CHAVE**

Demais campos de informação



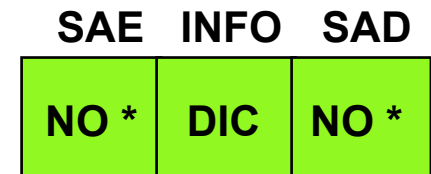
Estrutura Dicionário (DIC)

Estrutura do **nó**:

Campo **INFO**: informações do nó raiz (**do tipo dicionário**)

Campo **SAE**: endereço da subárvore à esquerda

Campo **SAD**: endereço da subárvore à direita



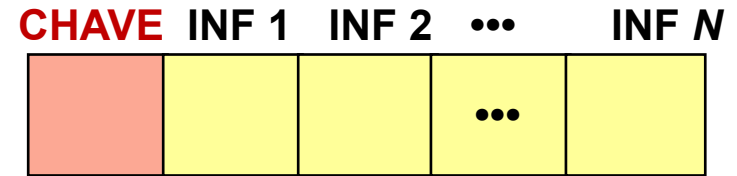
Estrutura Nó

ABB: estrutura de representação

Estrutura do **dicionário**:

Campo **CHAVE**

Demais campos de informação



Estrutura Dicionário (DIC)

Estrutura do **nó**:

Campo **INFO**: informações do nó raiz (**do tipo dicionário**)

Campo **SAE**: endereço da subárvore à esquerda

Campo **SAD**: endereço da subárvore à direita

Árvore: endereço do nó raiz

Ponteiro para o tipo nó



Estrutura Nó

ABB: estrutura de representação

Implementação em C:

// Registro do dicionário

```
struct registro {  
    int chave;  
    char nome[30];  
};
```

```
typedef struct registro reg;
```

ABB: estrutura de representação

Implementação em C:

// Registro do dicionário

```
struct registro {  
    int chave;  
    char nome[30];  
    int idade;  
};
```

```
typedef struct registro reg;
```

// Estrutura de um nó

```
struct no {  
    reg info;  
    struct no *sae,  
    struct no *sad;  
};
```

ABB: estrutura de representação

Implementação em C:

// Registro do dicionário

```
struct registro {  
    int chave;  
    char nome[30];  
    int idade;  
};
```

```
typedef struct registro reg;
```

// Estrutura de um nó

```
struct no {  
    reg info;  
    struct no *sae,  
    struct no *sad;  
};
```

// Árvore

```
typedef struct no * Arv;
```

Especificação do TAD ABB

Operação ***cria_arvore:***

Entrada: nenhuma

Pré-condição: nenhuma

Processo: cria uma árvore na **condição de vazia**

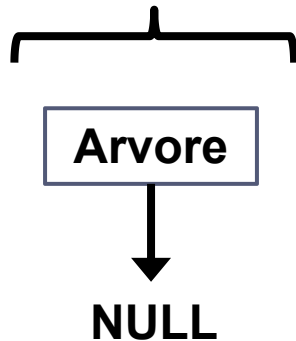
Saída: o endereço da árvore criada

Pós-condição: nenhuma

Similar à operação ***cria_vazia()***
do **TAD árvore binária**

ABB: implementação

árvore vazia



Arv ***cria_arvore*** ()
 retorna NULL;
FIM

Especificação do TAD ABB

Operação **arvore_vazia**:

Entrada: endereço da árvore

Pré-condição: nenhuma

Processo: verifica se a árvore binária está na **condição de vazia**

Saída: 1 - se vazia ou 0 - caso contrário

Pós-condição: nenhuma

Similar à operação **cria_vazia()**
do **TAD árvore binária**

ABB: implementação

arvore_vazia ($Arv * A$)

SE $A = NULL$ **ENTÃO**

retorna 1;

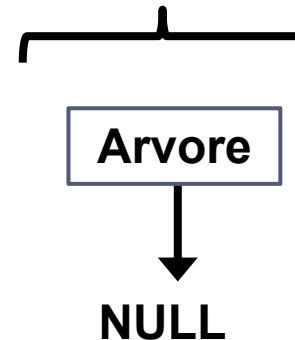
SENÃO

retorna 0;

FIM_SE

FIM

árvore vazia



Especificação do TAD ABB

Operação ***libera_arvore:***

Entrada: endereço do endereço da árvore (**referência**)

Pré-condição: a árvore existir e não estar vazia

Processo: percorre a árvore liberando o espaço alocado para cada nó, até que a árvore volte para o estado de vazia.

Saída: 1 - se operação bem sucedida ou 0 - caso contrário

Pós-condição: árvore no estado de vazia

Similar à operação **cria_vazia()**
do **TAD árvore binária**

ABB: implementação

libera_arvore (Arv * A)

SE árvore não estiver vazia **ENTÃO**

Libera subárvore à esquerda;

Libera subárvore à direita;

*Libera memória alocada para o nó raiz; // free(*A);*

FIM_SE

*Faz conteúdo de A = **NULL**; // *A = NULL;*

FIM

Especificação do TAD AB

Operação ***exibe_arvore:***

Entrada: endereço da árvore a ser exibida

Pré-condição: nenhuma

Processo: caminhe pela árvore em **pré-ordem**, apresentando o valor de cada nó.

Saída: nenhuma

Pós-condição: nenhuma

Similar à operação ***cria_vazia()***
do **TAD árvore binária**

Árvore binária: implementação

exibe_arvore (Arv A)

SE árvore vazia **ENTÃO**

 escreva("<>");

FIM_SE

escreva ("< "); // Abertura de contexto na notação textual

Exibe o campo **info** de A;

Exibe subárvore a esquerda de A;

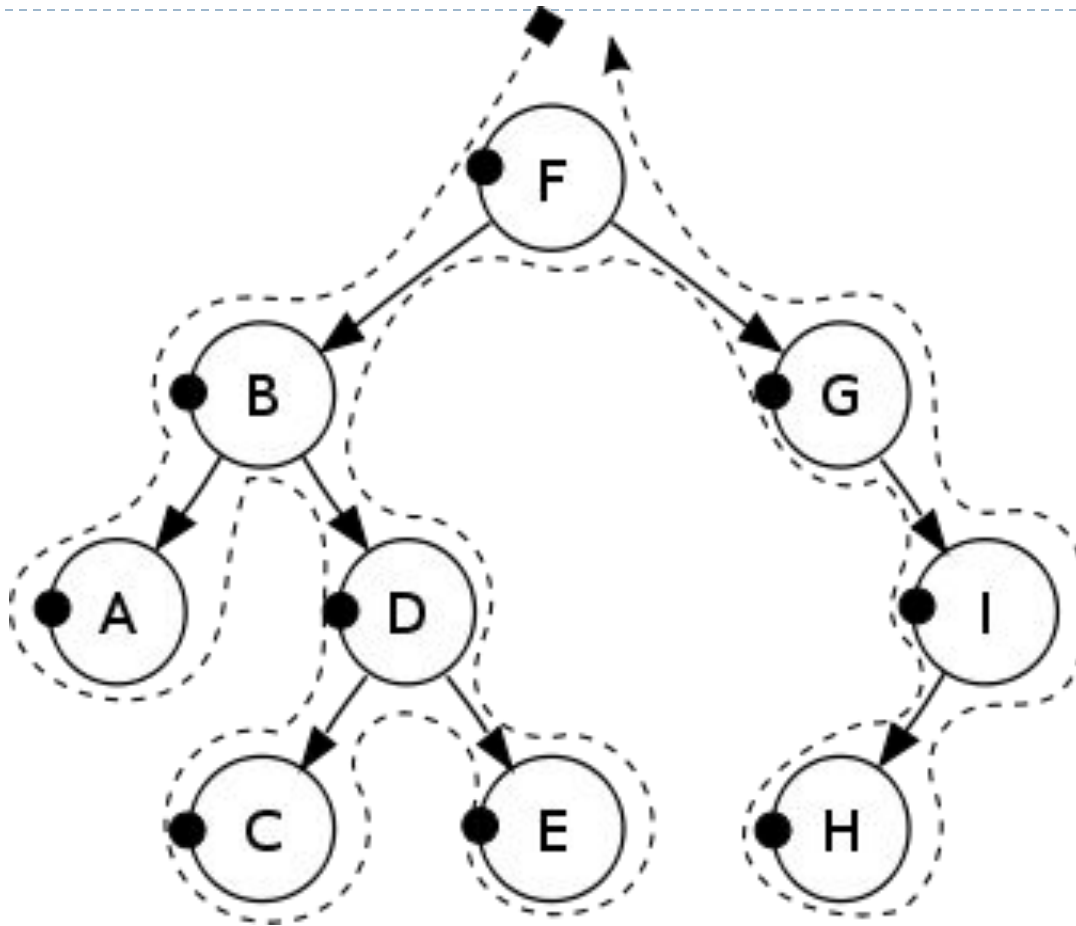
Exibe subárvore a direita de A;

} **recursão**

escreva(">"); // Fechamento de contexto na notação textual

FIM

Funcionamento da *exibe_arvore()*



saída: <F<B<A<><>>><D<C<><>>><E<><>>>>><G<><I<H<><>><>>>>>

Especificação do TAD ABB

Operação ***exibe_ordenado***:

Entrada: endereço da árvore

Pré-condição: nenhuma

Processo: caminhe pela árvore em **ordem simétrica (*in order*)**, apresentando o valor de cada nó.

Saída: nenhuma

Pós-condição: nenhuma

**Variação da operação anterior
(***exibe_arvore***)**

Árvore binária: implementação

exibe_ordenado (Arv A)

SE árvore **NÃO** vazia **ENTÃO**

Exibe subárvore a esquerda de A;

*Exibe o campo **info** de A;*

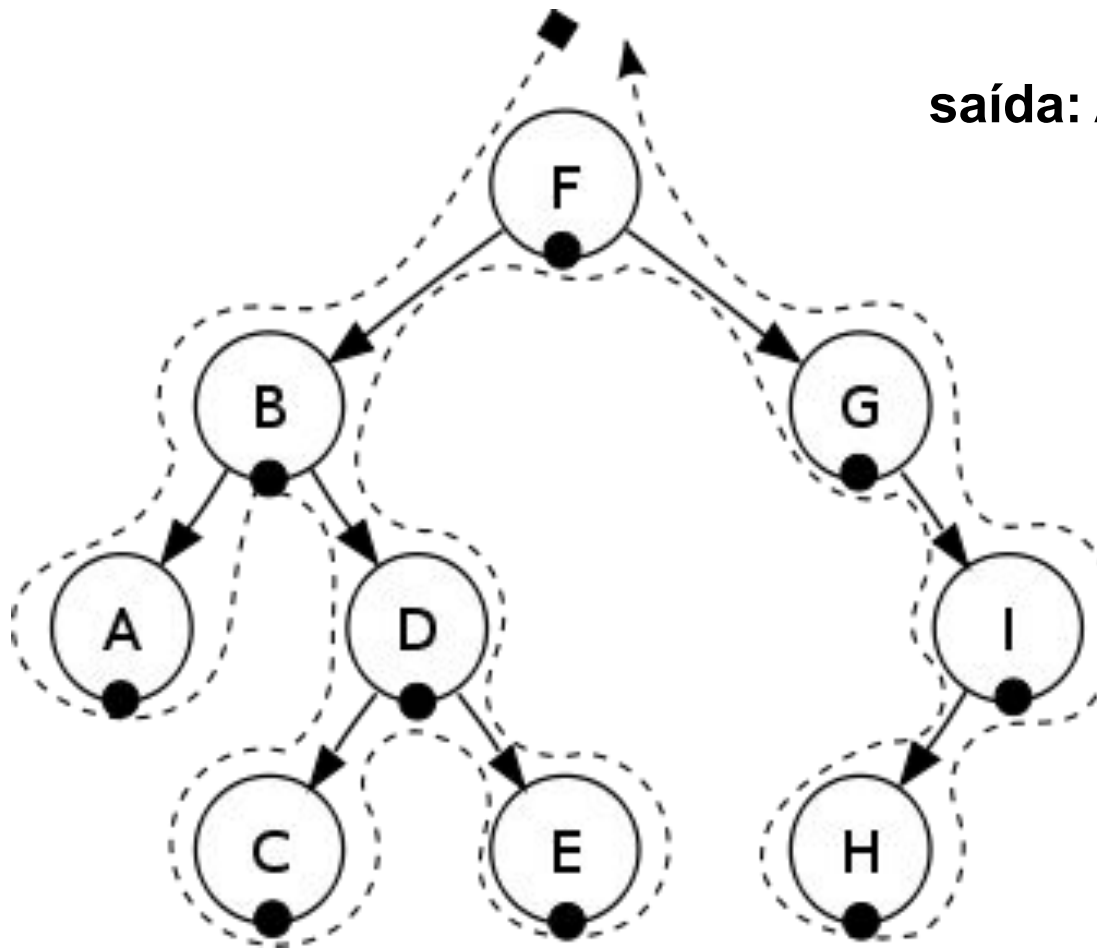
Exibe subárvore a direita de A;

 **recursão**

FIM_SE

FIM

Funcionamento da *exibe_ordenado()*



saída: A B C D E F G H I

Especificação do TAD ABB

Operação ***insere_ord***:

Entrada: endereço do endereço da árvore (**referência**) e o elemento a ser inserido

Pré-condição: a árvore existir

Processo: insere elemento de modo a manter a ordenação:

Se a árvore estiver vazia, inserir o novo elemento como raiz

Se novo elemento \leq raiz atual, inserir na subárvore à esquerda

Se novo elemento $>$ raiz atual, inserir na subárvore à direita

Saída: 1 - se operação bem sucedida ou 0 - caso contrário

Pós-condição: árvore de entrada com um nó a mais

ABB: inserção

Árvore vazia:

Arvore → NULL

ABB: inserção

Árvore vazia:

Arvore → NULL

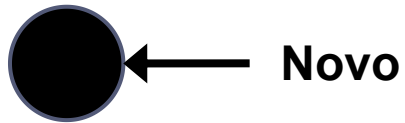


ABB: inserção

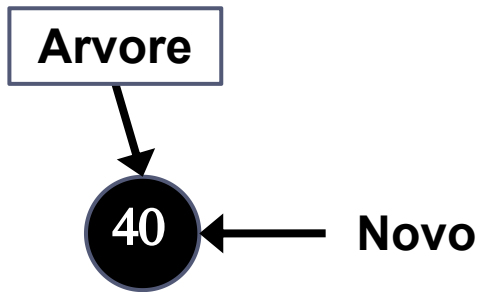
Árvore vazia:

Árvore → NULL

40 ← Novo

ABB: inserção

Árvore vazia:



***A = novo;**

ABB: inserção

Árvore com nós (não vazia):

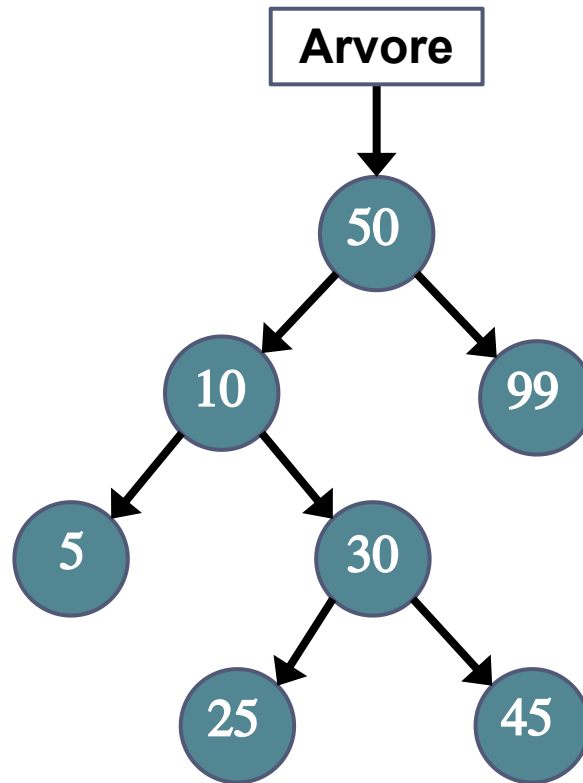
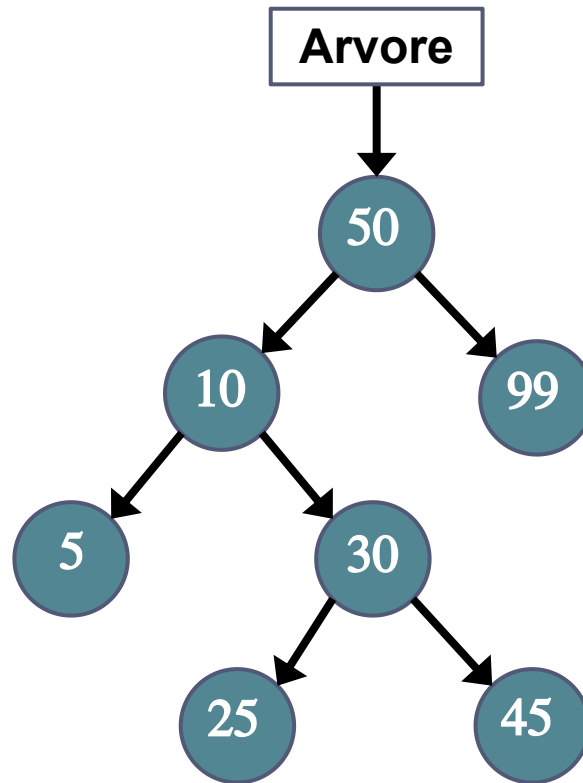


ABB: inserção

Árvore com nós (não vazia):




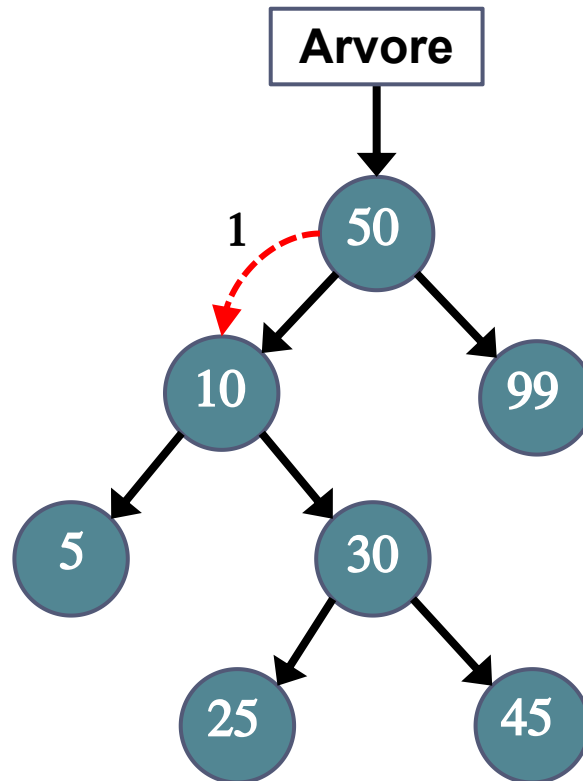
Novo → 

ABB: inserção

Árvore com nós (não vazia):

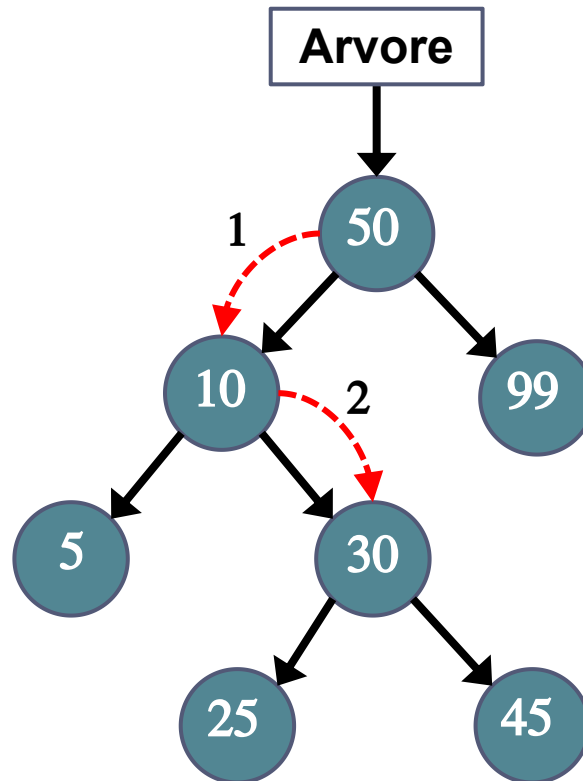


Novo → **40**

- 1 chave é menor do que 50:
visita filho da esquerda

ABB: inserção

Árvore com nós (não vazia):

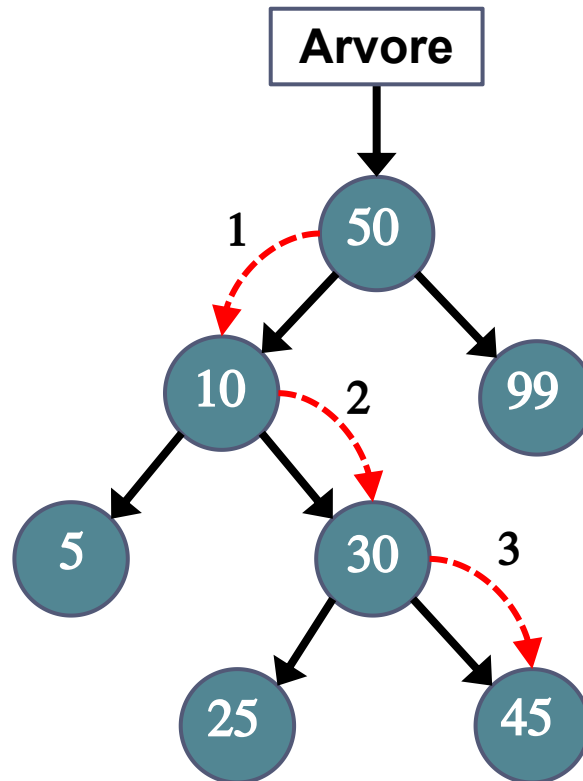


Novo → **40**

- | | |
|---|--|
| 1 | chave é menor do que 50:
visita filho da esquerda |
| 2 | chave é maior do que 10:
visita filho da direita |

ABB: inserção

Árvore com nós (não vazia):

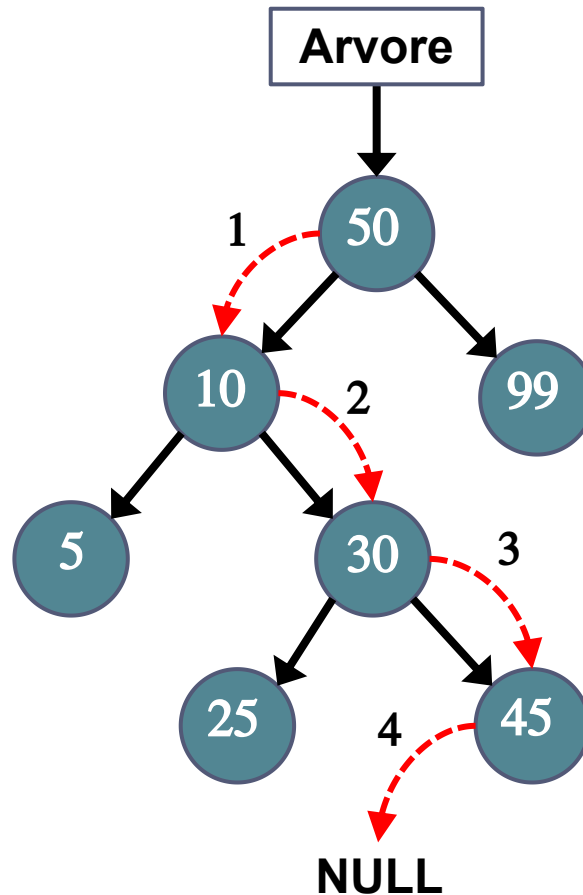


Novo → **40**

- | | |
|---|--|
| 1 | chave é menor do que 50:
visita filho da esquerda |
| 2 | chave é maior do que 10:
visita filho da direita |
| 3 | chave é maior do que 30:
visita filho da direita |

ABB: inserção

Árvore com nós (não vazia):

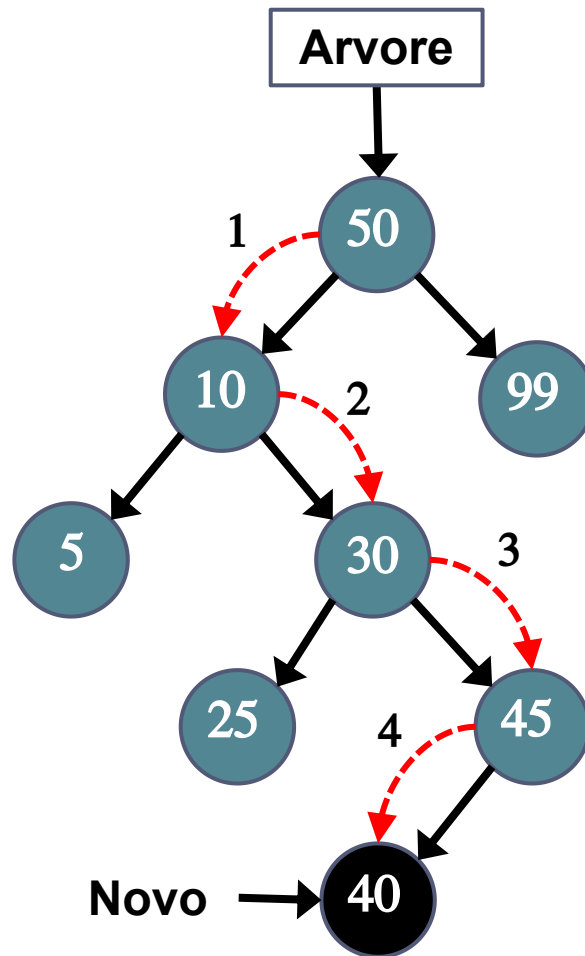


Novo → 40

- | | |
|---|--|
| 1 | chave é menor do que 50:
visita filho da esquerda |
| 2 | chave é maior do que 10:
visita filho da direita |
| 3 | chave é maior do que 30:
visita filho da direita |
| 4 | chave é menor do que 45:
visita filho da esquerda |

ABB: inserção

Árvore com nós (não vazia):



- | | |
|---|--|
| 1 | chave é menor do que 50:
visita filho da esquerda |
| 2 | chave é maior do que 10:
visita filho da direita |
| 3 | chave é maior do que 30:
visita filho da direita |
| 4 | chave é menor do que 45:
visita filho da esquerda |
| | Não existe filho da
esquerda. Novo nó passa
a ser o filho da esquerda
de 45 |

ABB: implementação

int **insere_ord** (Arv * A, **reg** elem)

SE \nexists árvore **ENTÃO** // A = **NULL**

 retorna 0;

FIM_SE

SE **arvore vazia** **ENTÃO** // Achou a posição

 Aloca um novo nó;

SE alocação falhou **ENTÃO**

 retorna 0;

FIM_SE

 Campo **INFO** do novo nó = elem;

 Campo **SAE** do novo nó = **NULL**;

 Campo **SAD** do novo nó = **NULL**;

...

ABB: implementação

...

conteúdo de A = novo nó;

retorna 1;

FIM_SE

SE *elem > chave da raiz* **ENTÃO**

retorna resultado da inserção do nó na subárvore a direita;

SENÃO

retorna resultado da inserção do nó na subárvore a esquerda;

FIM_SE

FIM

Especificação do TAD ABB

Operação ***remove_ord***:

Entrada: endereço do endereço da árvore (**referência**) e a chave a ser removida

Pré-condição: a árvore existir e não estar vazia

Processo: percorre a árvore até encontrar o nó ou não haver mais nós para visitar. Se encontrar o nó, remova-o mantendo a ordenação da ABB.

Saída: 1 - se operação bem sucedida ou 0 - caso contrário

Pós-condição: árvore de entrada com um nó a menos

ABB: remoção

Operação geralmente complexa

ABB: remoção

Operação geralmente complexa

Envolve 4 cenários:

Elemento **não está na árvore**

Elemento está em um **nó folha** (sem filhos)

Elemento está em um **nó de derivação com 1 filho**

Elemento está em um **nó de derivação com 2 filhos**

ABB: remoção

Operação geralmente complexa

Envolve 4 cenários:

Elemento **não está na árvore**

Elemento está em um **nó folha** (sem filhos)

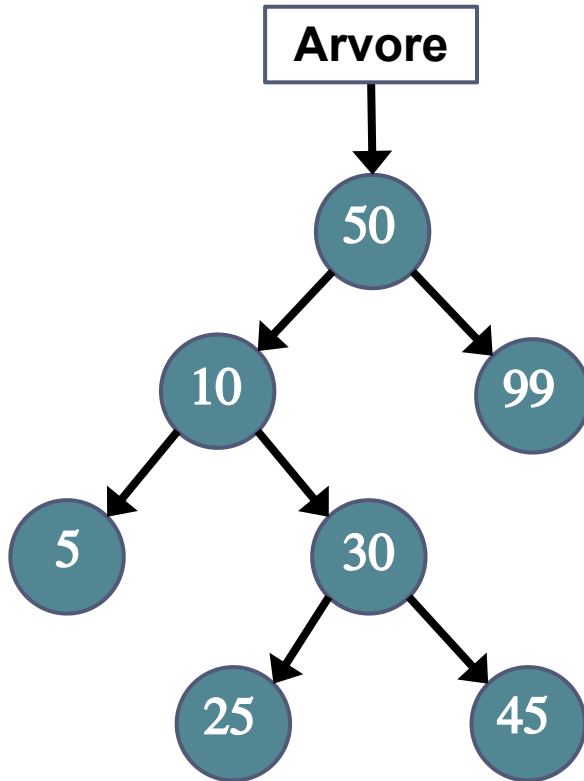
Elemento está em um **nó de derivação com 1 filho**

Elemento está em um **nó de derivação com 2 filhos**

Reorganização dos nós da árvore pode ser necessária **para manter a ordenação**

ABB: remoção

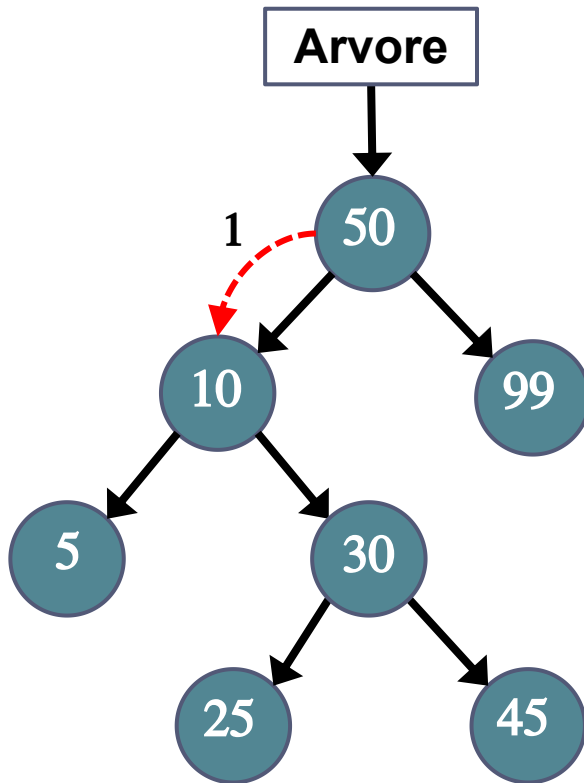
Elemento **NÃO** está na árvore:



Chave procurada: 28

ABB: remoção

Elemento **NÃO** está na árvore:

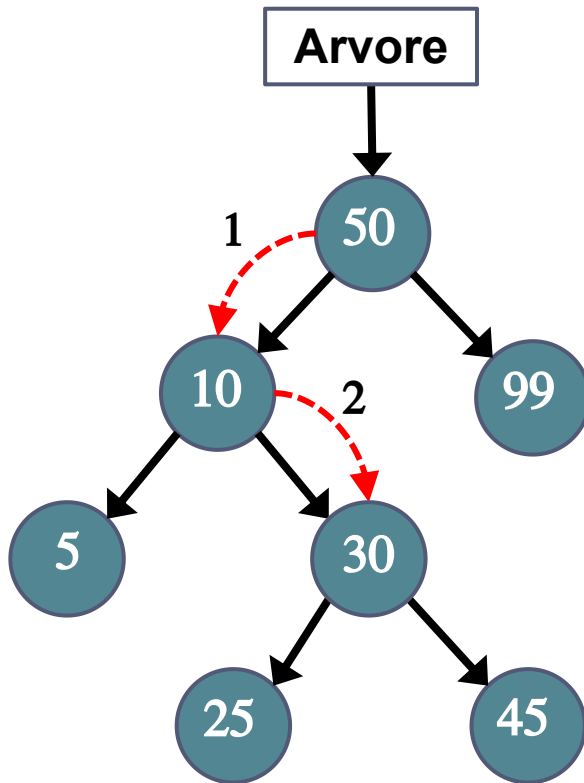


Chave procurada: 28

1	chave procurada é menor do que 50: visita filho da esquerda
---	---

ABB: remoção

Elemento **NÃO** está na árvore:

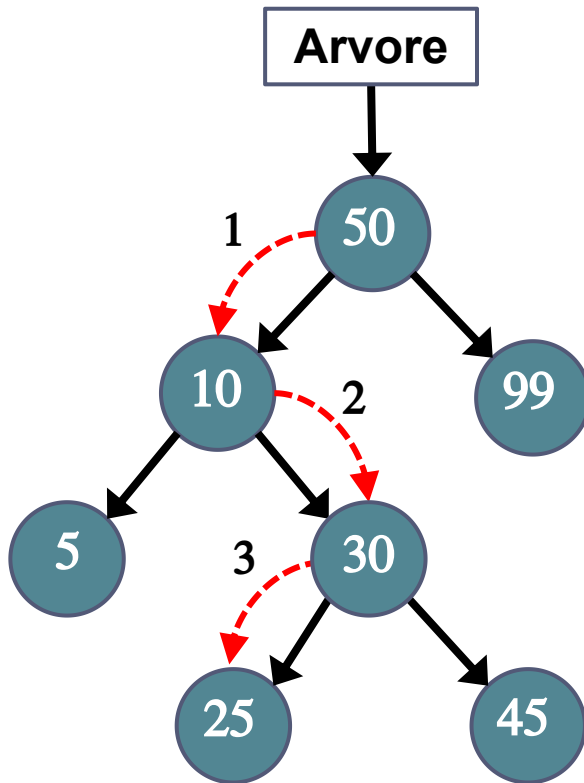


Chave procurada: 28

1	chave procurada é menor do que 50: visita filho da esquerda
2	chave procurada é maior do que 10: visita filho da direita

ABB: remoção

Elemento **NÃO** está na árvore:

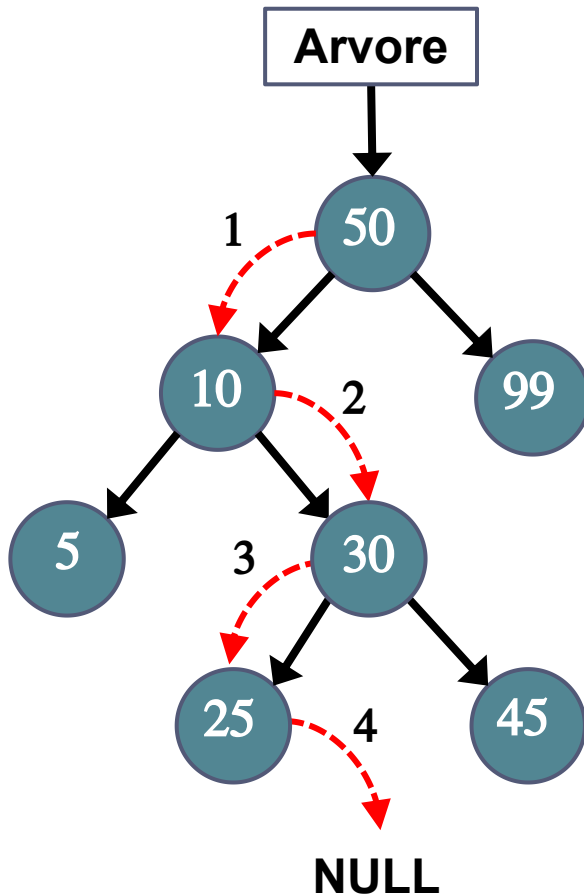


Chave procurada: 28

1	chave procurada é menor do que 50: visita filho da esquerda
2	chave procurada é maior do que 10: visita filho da direita
3	chave procurada é menor do que 30: visita filho da esquerda

ABB: remoção

Elemento **NÃO** está na árvore:

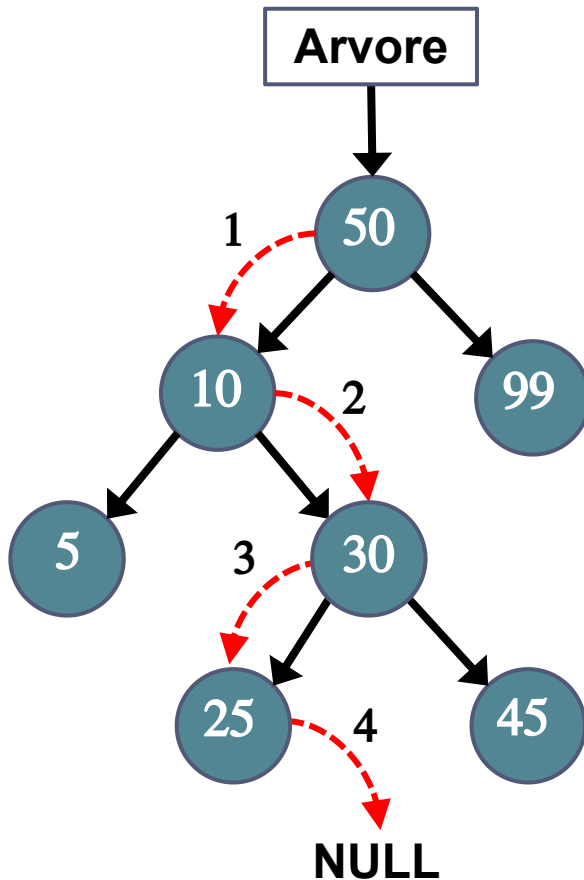


Chave procurada: 28

1	chave procurada é menor do que 50: visita filho da esquerda
2	chave procurada é maior do que 10: visita filho da direita
3	chave procurada é menor do que 30: visita filho da esquerda
4	chave procurada é maior do que 25: visita filho da direita

ABB: remoção

Elemento **NÃO** está na árvore:

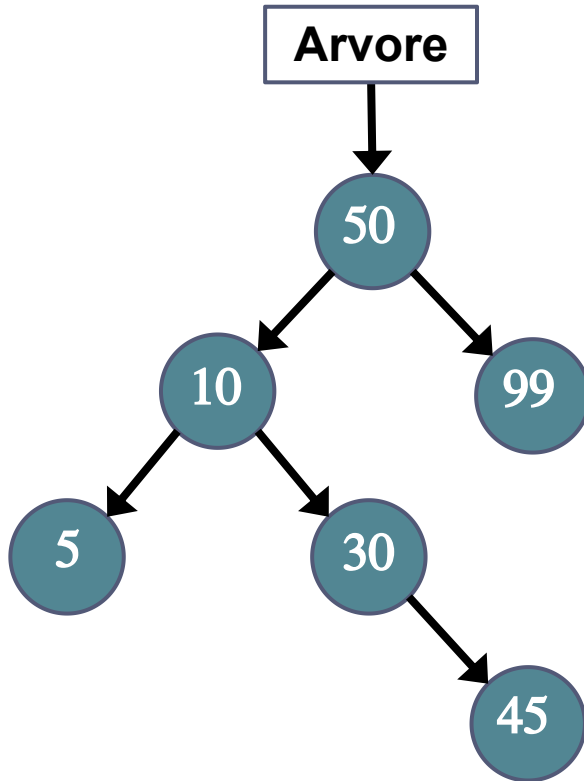


Chave procurada: 28

1	chave procurada é menor do que 50: visita filho da esquerda
2	chave procurada é maior do que 10: visita filho da direita
3	chave procurada é menor do que 30: visita filho da esquerda
4	chave procurada é maior do que 25: visita filho da direita
	filho a direita de 25 não existe: returnar 0 (busca falhou)

ABB: remoção

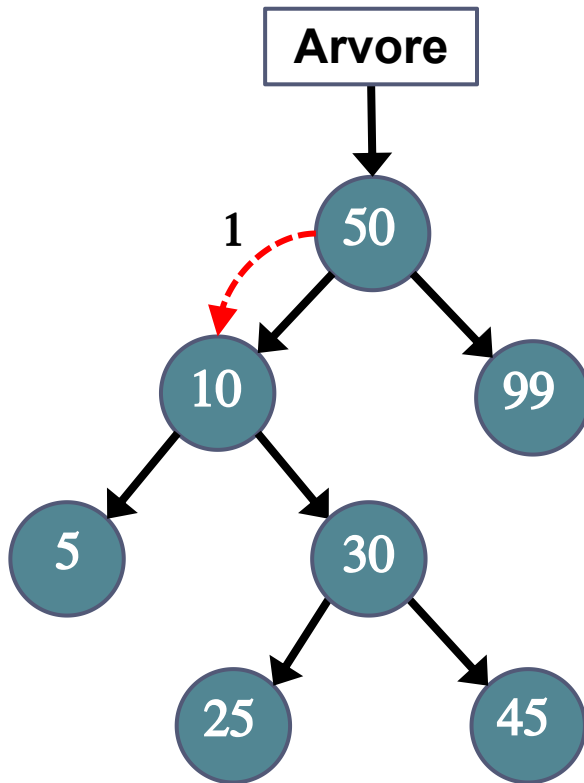
Remoção de um nó folha (**0 filhos**):



Chave procurada: 5

ABB: remoção

Remoção de um nó folha (**0 filhos**):

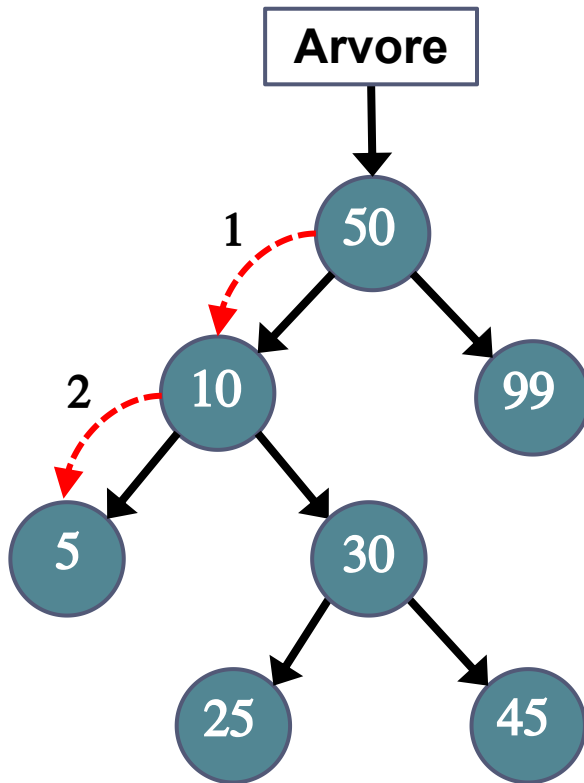


Chave procurada: 5

1	chave procurada é menor do que 50: visita filho da esquerda
---	---

ABB: remoção

Remoção de um nó folha (**0 filhos**):

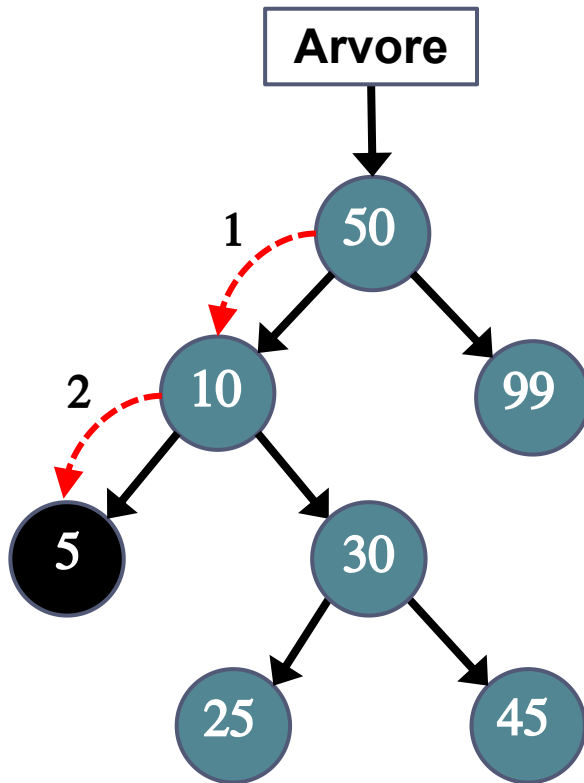


Chave procurada: 5

- | | |
|---|---|
| 1 | chave procurada é menor do que 50: visita filho da esquerda |
| 2 | chave procurada é menor do que 10: visita filho da esquerda |

ABB: remoção

Remoção de um nó folha (**0 filhos**):

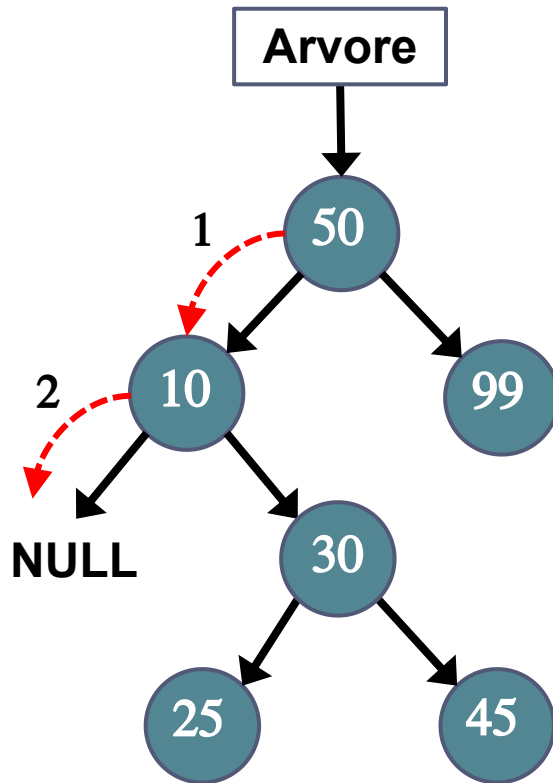


Chave procurada: 5

1	chave procurada é menor do que 50: visita filho da esquerda
2	chave procurada é menor do que 10: visita filho da esquerda
	chave procurada é igual a do nó e ele não tem filhos (nó folha):

ABB: remoção

Remoção de um nó folha (**0 filhos**):

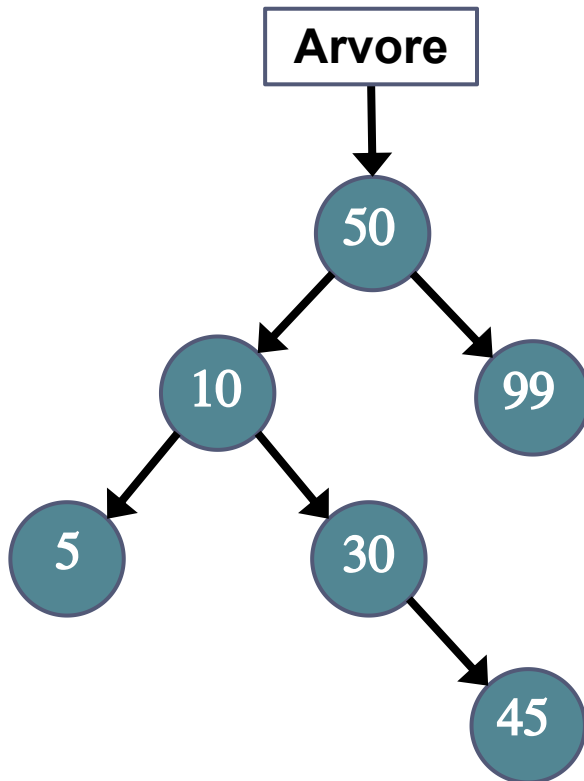


Chave procurada: 5

1	chave procurada é menor do que 50: visita filho da esquerda
2	chave procurada é menor do que 10: visita filho da esquerda
	chave procurada é igual a do nó e ele não tem filhos (nó folha): libera espaço alocado ao nó e faz o nó pai apontar para NULL

ABB: remoção

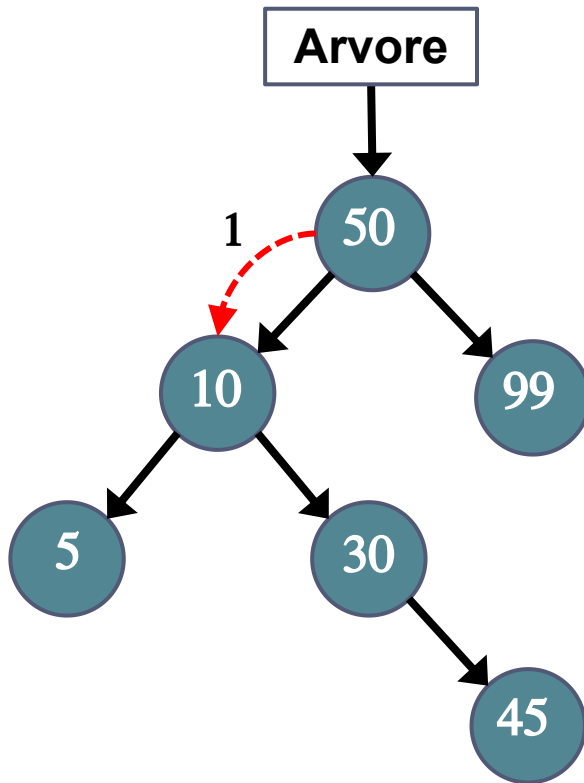
Remoção de um nó de derivação com **1 filho**:



Chave procurada: 30

ABB: remoção

Remoção de um nó de derivação com **1 filho**:

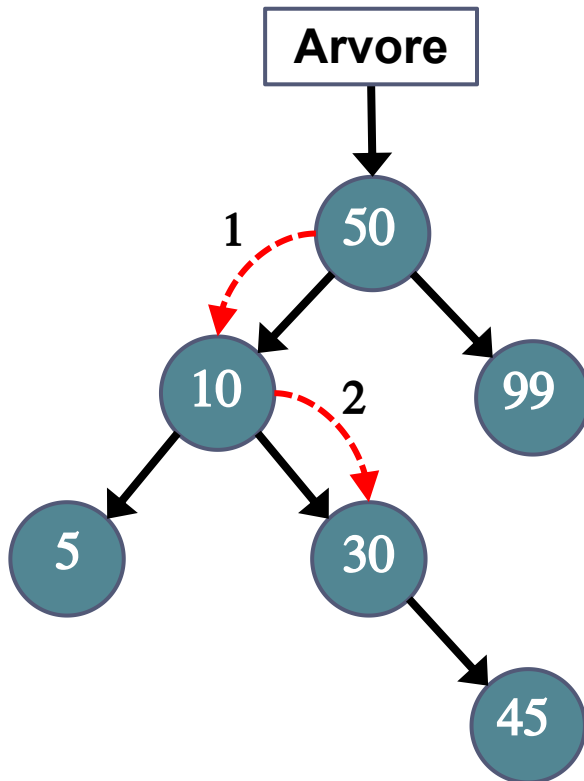


Chave procurada: 30

1	chave procurada é menor do que 50: visita filho da esquerda
---	---

ABB: remoção

Remoção de um nó de derivação com **1 filho**:

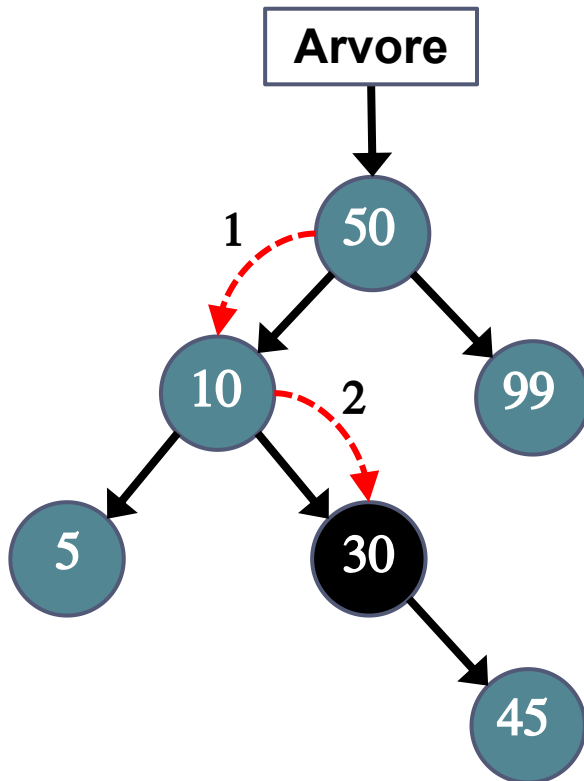


Chave procurada: 30

- | | |
|---|---|
| 1 | chave procurada é menor do que 50: visita filho da esquerda |
| 2 | chave procurada é maior do que 10: visita filho da direita |

ABB: remoção

Remoção de um nó de derivação com **1 filho**:

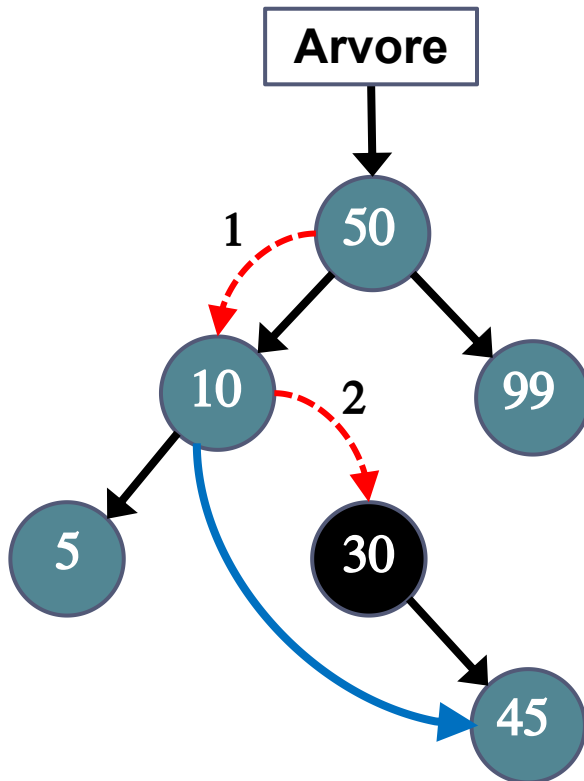


Chave procurada: 30

1	chave procurada é menor do que 50: visita filho da esquerda
2	chave procurada é maior do que 10: visita filho da direita
	chave procurada é igual a do nó e ele só tem um único filho :

ABB: remoção

Remoção de um nó de derivação com **1 filho**:

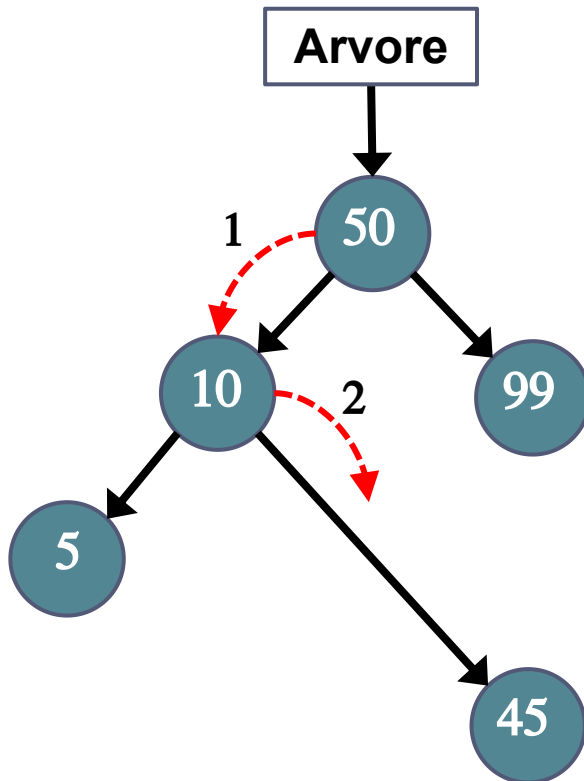


Chave procurada: 30

1	chave procurada é menor do que 50: visita filho da esquerda
2	chave procurada é maior do que 10: visita filho da direita
	chave procurada é igual a do nó e ele só tem um único filho : fazer o pai apontar para o filho

ABB: remoção

Remoção de um nó de derivação com **1 filho**:

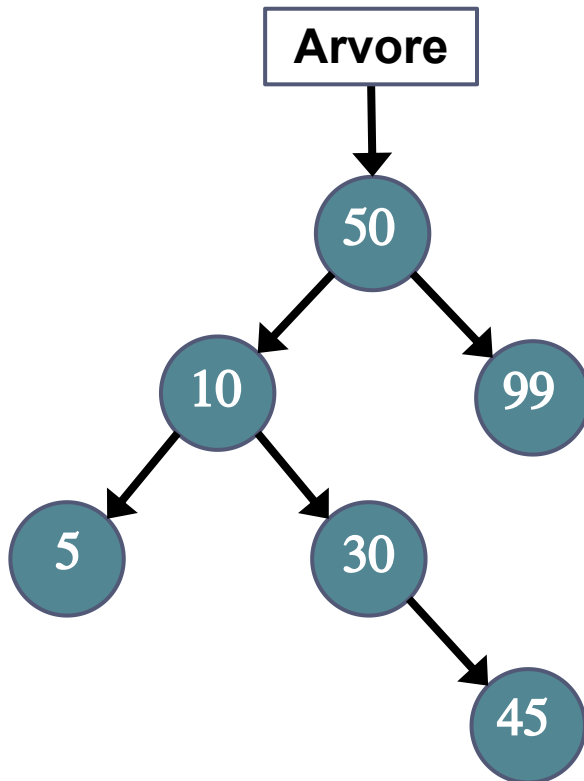


Chave procurada: 30

1	chave procurada é menor do que 50: visita filho da esquerda
2	chave procurada é maior do que 10: visita filho da direita
	chave procurada é igual a do nó e ele só tem um único filho : fazer o pai apontar para o filho e liberar memória alocada ao nó

ABB: remoção

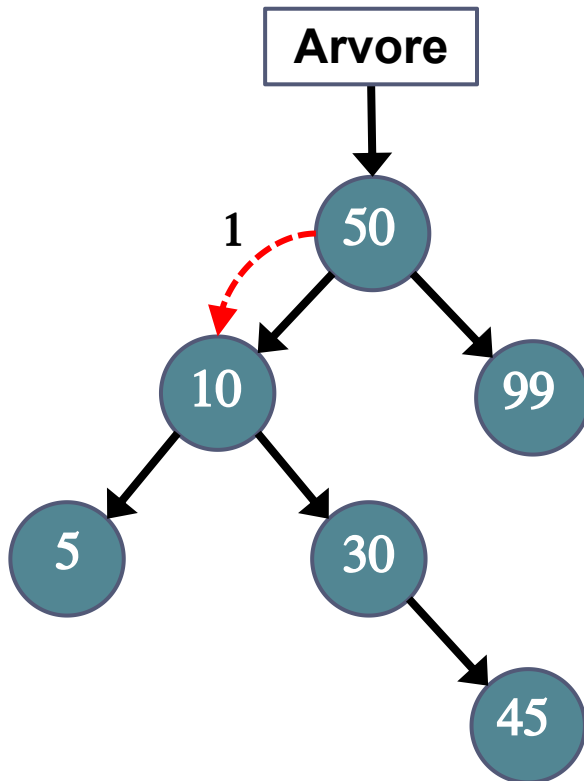
Remoção de um nó de derivação com **2 filhos**:



Chave procurada: 10

ABB: remoção

Remoção de um nó de derivação com **2 filhos**:

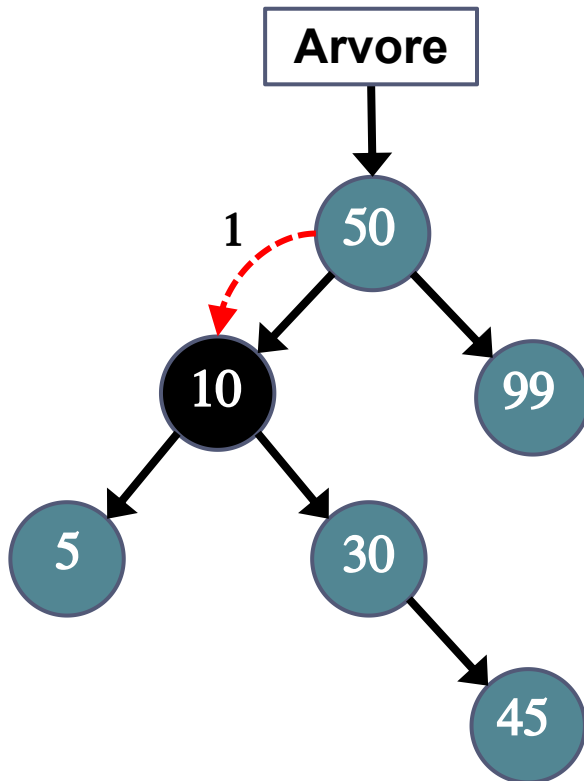


Chave procurada: 10

- | | |
|---|---|
| 1 | chave procurada é menor do que 50: visita filho da esquerda |
|---|---|

ABB: remoção

Remoção de um nó de derivação com **2 filhos**:

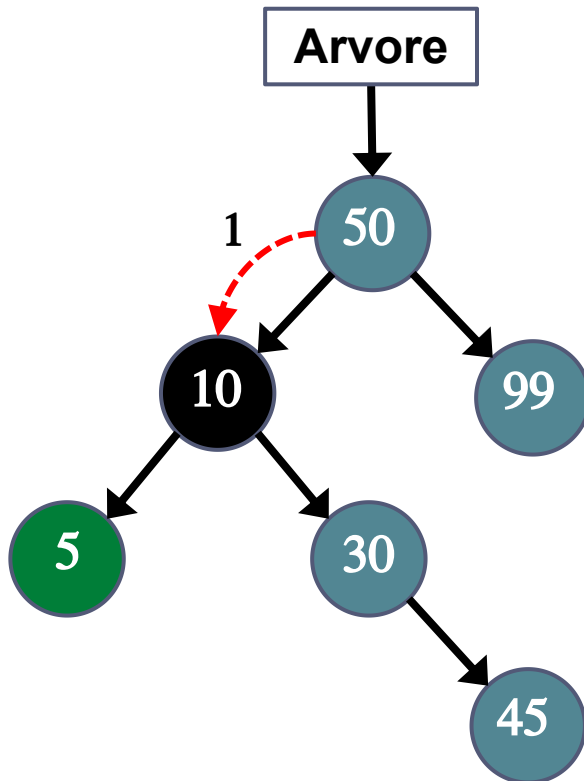


Chave procurada: 10

1	chave procurada é menor do que 50: visita filho da esquerda
	chave procurada é igual a do nó e ele tem 2 filhos :

ABB: remoção

Remoção de um nó de derivação com **2 filhos**:

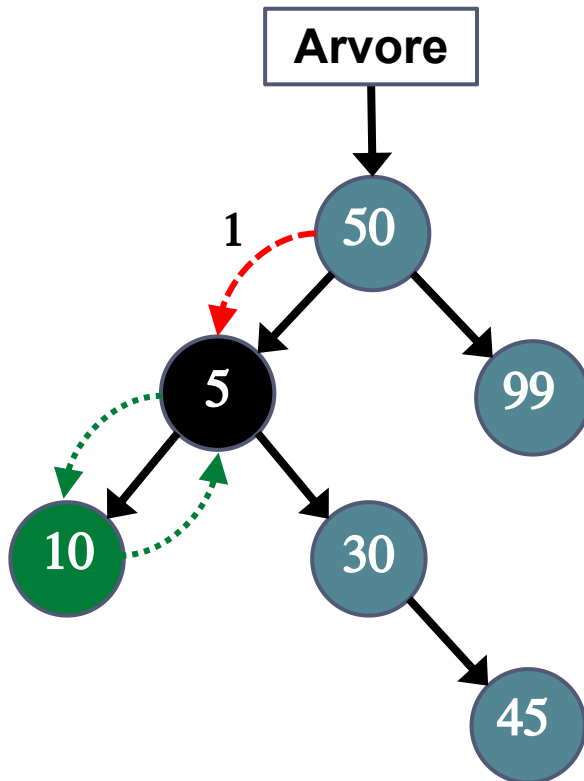


Chave procurada: 10

1	chave procurada é menor do que 50: visita filho da esquerda
	chave procurada é igual a do nó e ele tem 2 filhos : encontrar a maior chave (filho + à direita) da SAE

ABB: remoção

Remoção de um nó de derivação com **2 filhos**:

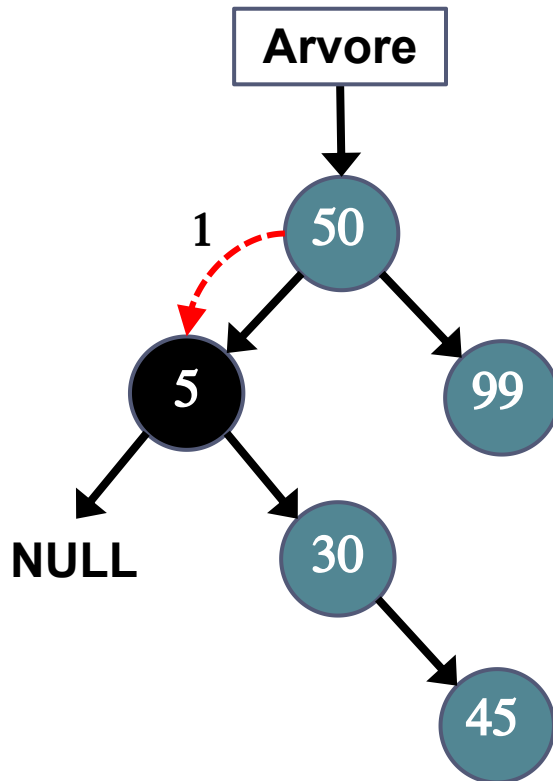


Chave procurada: 10

1	chave procurada é menor do que 50: visita filho da esquerda
	chave procurada é igual a do nó e ele tem 2 filhos : encontrar a maior chave (filho + à direita) da SAE; trocá-lo com o nó a remover

ABB: remoção

Remoção de um nó de derivação com **2 filhos**:

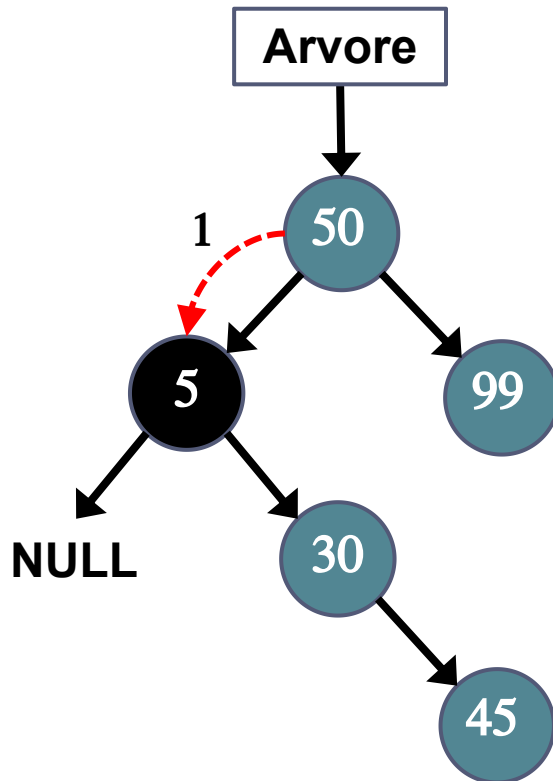


Chave procurada: 10

1	chave procurada é menor do que 50: visita filho da esquerda
	chave procurada é igual a do nó e ele tem 2 filhos : encontrar a maior chave (filho + à direita) da SAE; trocá-lo com o nó a remover; e aplicar remoção na SAE

ABB: remoção

Remoção de um nó de derivação com **2 filhos**:



Chave procurada: 10

1	chave procurada é menor do que 50: visita filho da esquerda
	chave procurada é igual a do nó e ele tem 2 filhos : encontrar a maior chave (filho + à direita) da SAE; trocá-lo com o nó a remover; e aplicar remoção na SAE

2a remoção será de um **nó folha** (sem filhos) ou **com um único filho**

ABB: implementação

int **remove_ord** (*Arv* * A, *int* chave)

SE \nexists árvore **OU** árvore vazia **ENTÃO**

retorna 0;

FIM_SE

SE *chave* > *chave da raiz* **ENTÃO**

retorna resultado da remoção do elem da subárvore à direita;

SENÃO SE *chave* < *chave da raiz* **ENTÃO**

retorna resultado da remoção do elem da subárvore à esquerda;

SENÃO // *Encontrou o nó (chave = chave da raiz)*

SE *SAE* = NULL **E** *SAD* = NULL **ENTÃO** // *Nó é folha*

*Libera a memória alocada para o nó; // free(*A);*

Atribui ao conteúdo de A o valor NULL;

retorna 1;

ABB: implementação

int **remove_ord** (Arv * A, int chave)

SE \nexists árvore **OU** árvore vazia **ENTÃO**

retorna 0;

FIM_SE

SE chave > chave da raiz **ENTÃO**

retorna resultado da **remoção do elem da subárvore à direita**;

SENÃO SE chave < chave da raiz **ENTÃO**

retorna resultado da **remoção do elem da subárvore à esquerda**;

SENÃO // Encontrou o nó (chave = chave da raiz)

SE SAE = NULL **E** SAD = NULL **ENTÃO** // Nó é folha

Libera a memória alocada para o nó; // free(*A);

Atribui ao conteúdo de A o valor NULL;

retorna 1;

**remoção sempre
será do nó raiz**

ABB: implementação

...

// Nó tem 1 filho à esquerda

SENÃO SE SAE \neq NULL E SAD = NULL ENTÃO

*Atribui a uma variável **aux** o conteúdo de A;*

Atribui ao conteúdo de A o endereço da subárvore à esquerda (SAE);

*Libera a memória endereçada por **aux**; // free(aux);*

retorna 1;

// Nó tem 1 filho à direita

SENÃO SE SAE = NULL E SAD \neq NULL ENTÃO

*Atribui a uma variável **aux** o conteúdo de A;*

Atribui ao conteúdo de A o endereço da subárvore à direita (SAD);

*Libera a memória endereçada por **aux**; // free(aux);*

retorna 1;

...

ABB: implementação

...

SENÃO // Nó tem 2 filhos

// Localiza a maior chave (nó + à direita) da subárvore a esquerda

*Atribui a uma variável **aux** o endereço de SAE;*

ENQUANTO SAD de **aux** \neq NULL **FAÇA**

***aux** = SAD de **aux**;*

FIM_ENQUANTO

*// Troca de conteúdo entre **aux** e o nó a remover*

*temp = Campo **info** da raiz;*

*Campo **info** da raiz = campo **info** de **aux**;*

*Campo **info** de **aux** = temp;*

*retorna resultado da **remoção do elem da subárvore à esquerda**;*

FIM_SE

FIM_SE

FIM



Especificação do TAD ABB

Operação ***busca_bin***:

Entrada: endereço da árvore e a chave a ser encontrada

Pré-condição: a árvore existir e não estar vazia

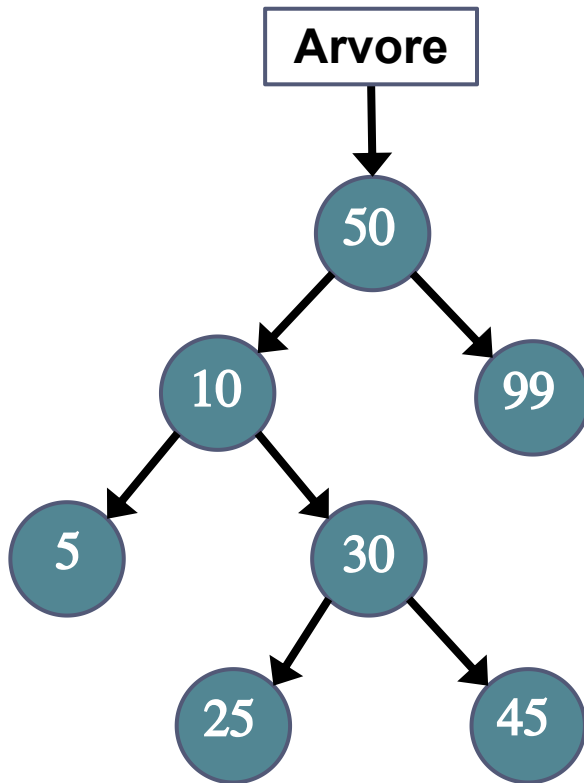
Processo: percorre a árvore, considerando a distribuição ordenada dos dados, até encontrar o nó ou atingir um nó folha. Se encontrar o nó, retorne o seu endereço.

Saída: o endereço do nó desejado ou ***NULL*** se ele não estiver na árvore

Pós-condição: nenhuma

ABB: busca binária

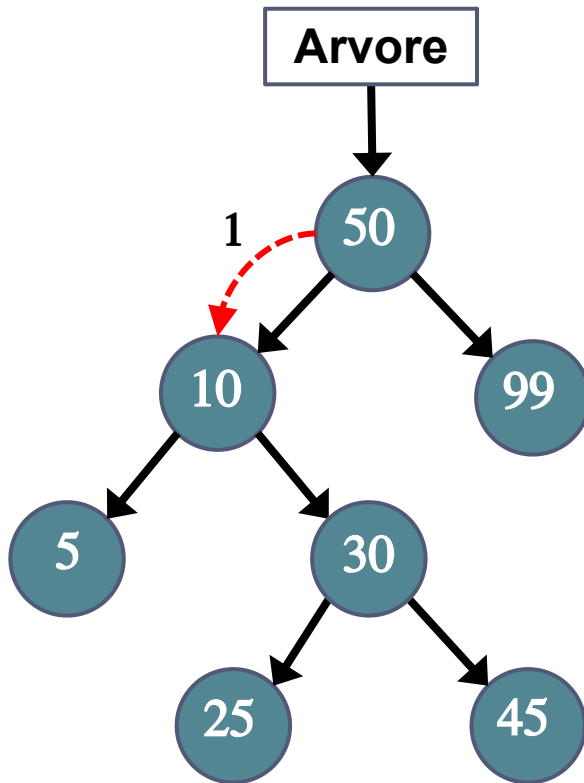
Elemento está na árvore:



Chave procurada: 30

ABB: busca binária

Elemento está na árvore:

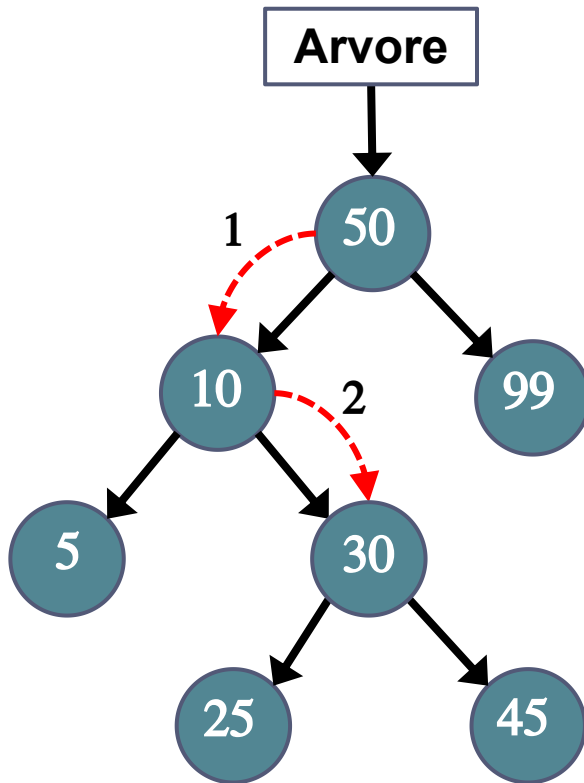


Chave procurada: 30

1	chave procurada é menor do que 50: visita filho da esquerda
---	---

ABB: busca binária

Elemento está na árvore:

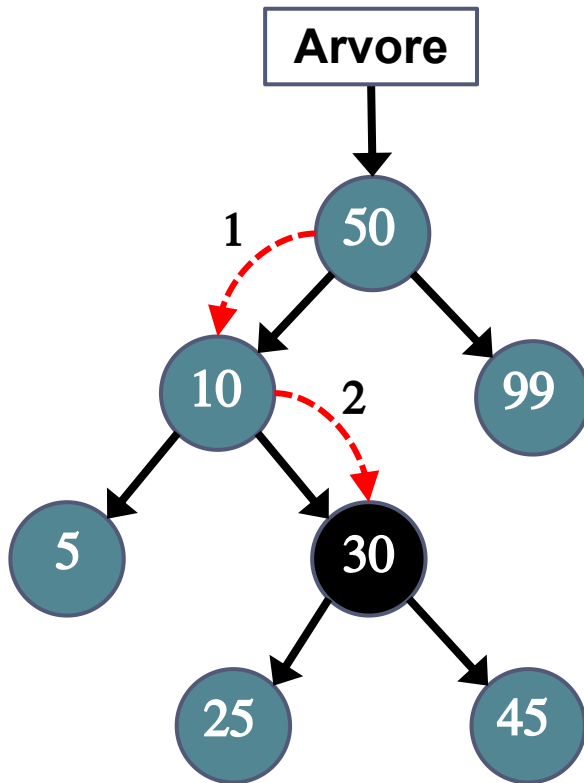


Chave procurada: 30

- | | |
|---|---|
| 1 | chave procurada é menor do que 50: visita filho da esquerda |
| 2 | chave procurada é maior do que 10: visita filho da direita |

ABB: busca binária

Elemento está na árvore:

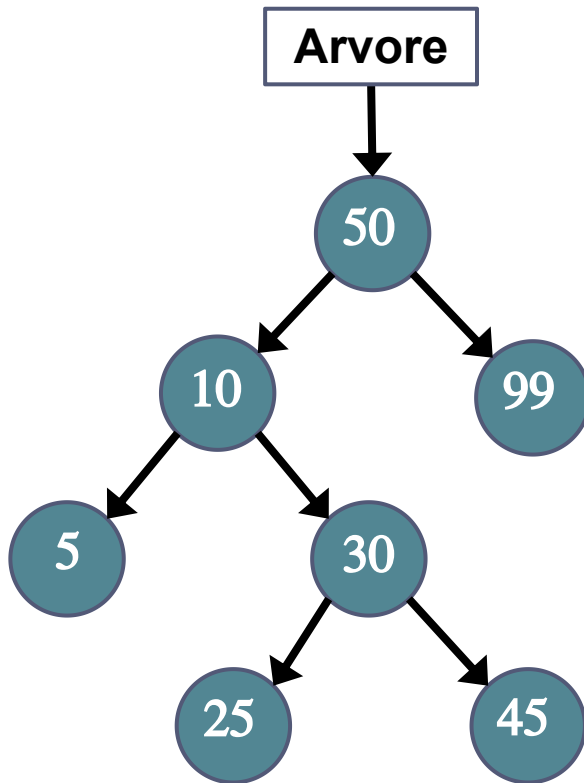


Chave procurada: 30

1	chave procurada é menor do que 50: visita filho da esquerda
2	chave procurada é maior do que 10: visita filho da direita
	chave procurada é igual a do nó: retornar endereço do nó

ABB: busca binária

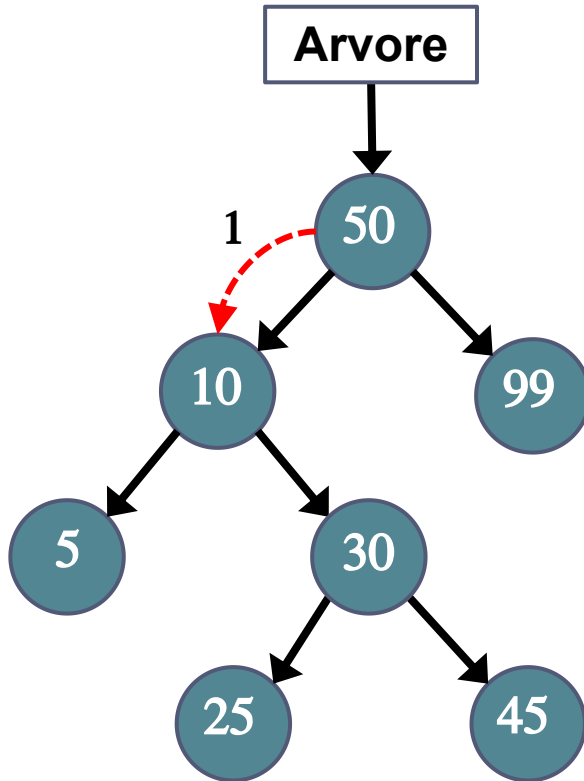
Elemento **NÃO** está na árvore:



Chave procurada: 28

ABB: busca binária

Elemento **NÃO** está na árvore:

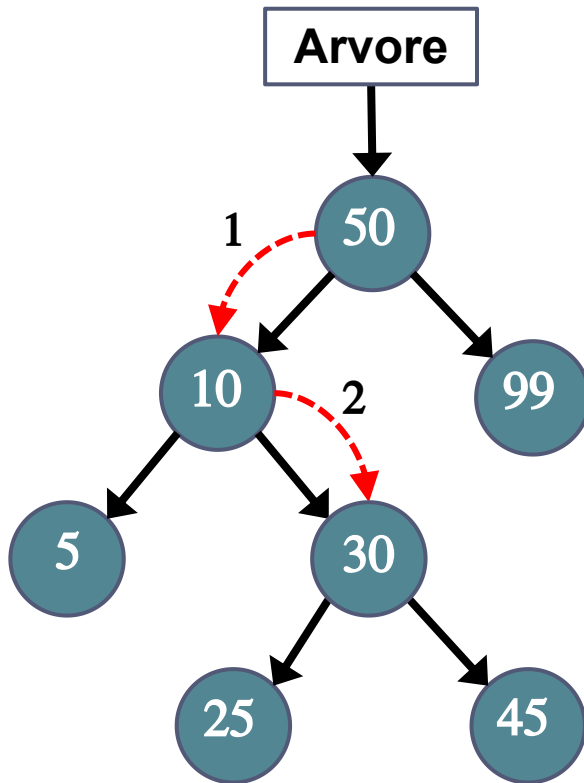


Chave procurada: 28

1	chave procurada é menor do que 50: visita filho da esquerda
---	---

ABB: busca binária

Elemento **NÃO** está na árvore:

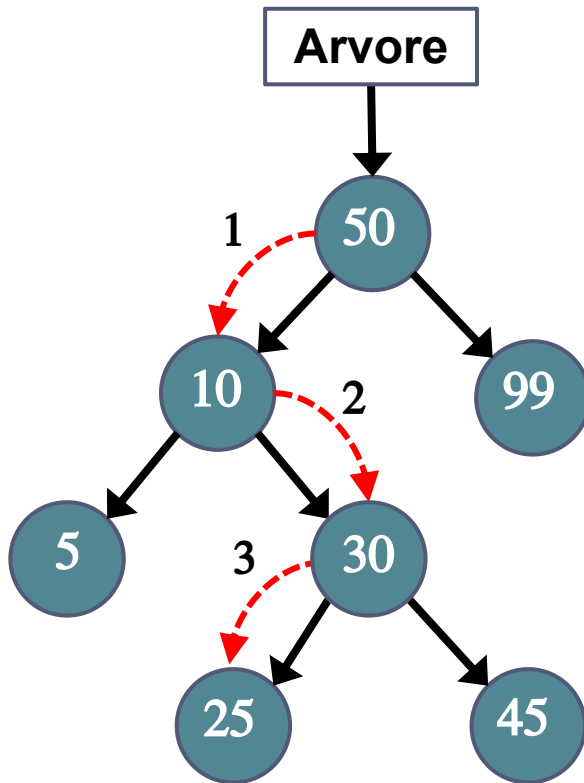


Chave procurada: 28

- | | |
|---|---|
| 1 | chave procurada é menor do que 50: visita filho da esquerda |
| 2 | chave procurada é maior do que 10: visita filho da direita |

ABB: busca binária

Elemento **NÃO** está na árvore:

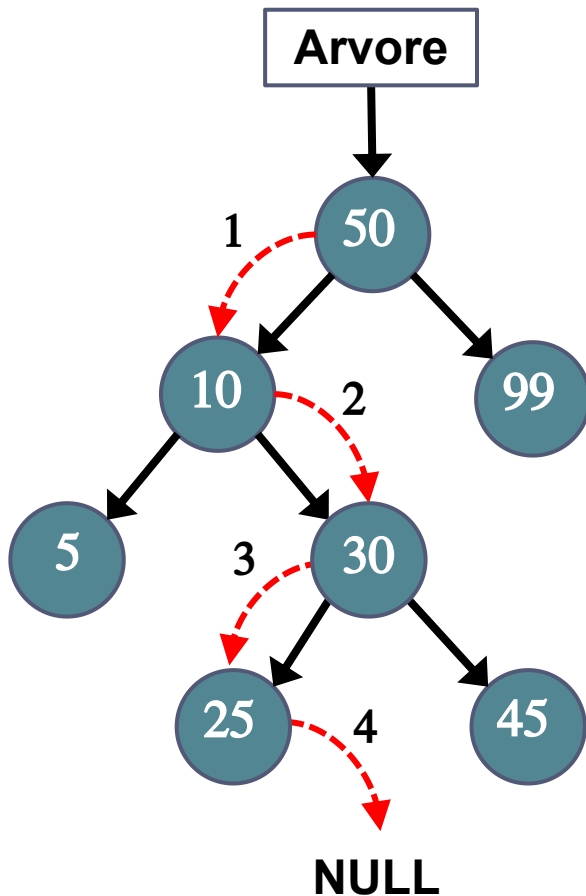


Chave procurada: 28

1	chave procurada é menor do que 50: visita filho da esquerda
2	chave procurada é maior do que 10: visita filho da direita
3	chave procurada é menor do que 30: visita filho da esquerda

ABB: busca binária

Elemento **NÃO** está na árvore:

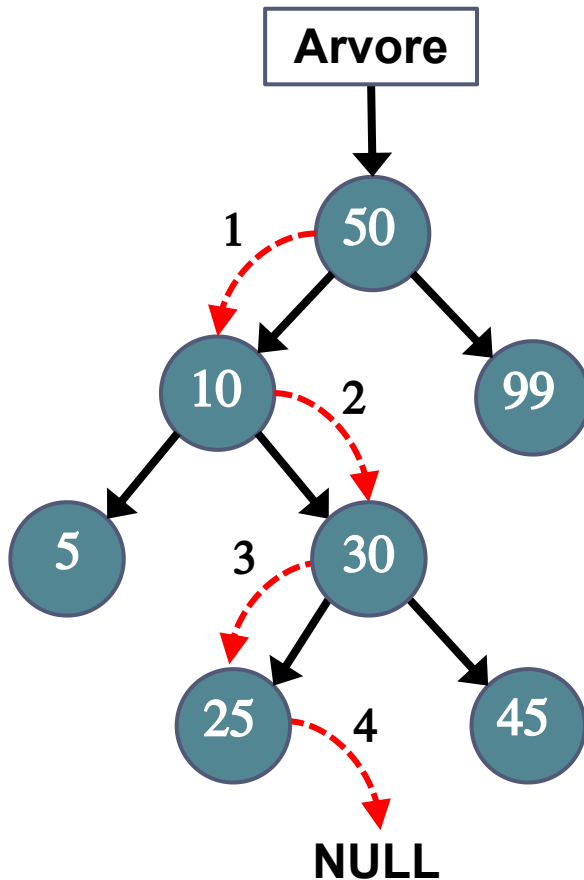


Chave procurada: 28

1	chave procurada é menor do que 50: visita filho da esquerda
2	chave procurada é maior do que 10: visita filho da direita
3	chave procurada é menor do que 30: visita filho da esquerda
4	chave procurada é maior do que 25: visita filho da direita

ABB: busca binária

Elemento **NÃO** está na árvore:



Chave procurada: 28

1	chave procurada é menor do que 50: visita filho da esquerda
2	chave procurada é maior do que 10: visita filho da direita
3	chave procurada é menor do que 30: visita filho da esquerda
4	chave procurada é maior do que 25: visita filho da direita
	filho a direita de 25 não existe: returnar <i>NULL</i> (busca falhou)

ABB: implementação

Arv **busca_bin** (Arv A, int chave)

SE árvore vazia **ENTÃO**

retorna NULL;

FIM_SE

SE chave da raiz = chave **ENTÃO** // Encontrou o nó

retornar A;

SENÃO SE chave > chave da raiz **ENTÃO**

retorna resultado da **busca na subárvore à direita**;

SENÃO // chave < chave da raiz

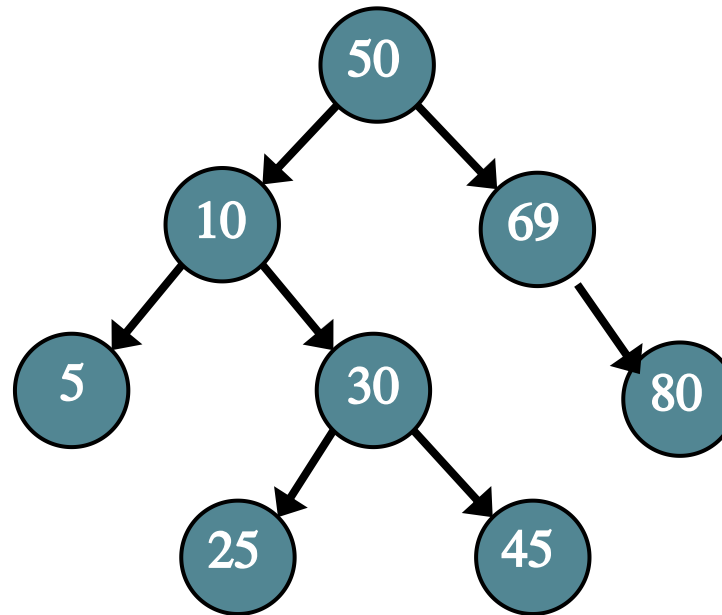
retorna resultado da **busca na subárvore à esquerda**;

FIM_SE

FIM

Exercícios

1. Considerando a ABB abaixo, apresente o processamento necessário para incluir o nó 40, a remoção dos nós 9 e 30 e a pesquisa dos nós 55 e 80.



2. Implemente as operações básicas de uma árvore binária de busca com a seguinte estrutura de registro: chave (inteiro), nome (string de 100 caracteres), idade (inteiro) e salario (real).

Bibliografia

Slides adaptados do material da Profa. Dra. Gina Maira Barbosa de Oliveira, da Profa. Dra. Denise Guliato e do Prof. Dr. Bruno Travençolo.

EDELWEISS, N; GALANTE, R. Estruturas de dados (Série Livros Didáticos Informática UFRGS, v. 18), Bookman, 2008.

CORMEN, T.H. et al. Algoritmos: Teoria e Prática, Campus, 2002

ZIVIANI, N. Projeto de algoritmos: com implementações em Pascal e C (2ª ed.), Thomson, 2004

CELES, W.; CERQUEIRA, R. & RANGEL, J. L. Introdução a Estruturas de Dados: com técnicas de programação em C, Elsevier, 2004